Paralleles Lesen und Schreiben von farbigen 2D-Barcodes mit MPI

Seminar Wissenschaftliches Rechnen
ZHAW, Zürich

Florian Lüthi*

15. Juni 2013

*luethifl@students.zhaw.ch

Inhaltsverzeichnis

1 Einleitung

Diese Arbeit beschäftigt sich mit der von Md. Mashud Rana, M. E. Kawsar, M. E. Rabbani, S. M. M. Rashid und K. E. U. Ahmed vorgeschlagenen Methode des Lesens und Schreibens von farbigen zweidimensionalen Barcodes mit hoher Datenkapazität [2].

Sie beschreibt diese Methode, zeigt eine mögliche Umsetzung des Verfahrens in `C++` mit Parallelisierung mittels `MPI`, und macht eine quantitative Analyse der Implementation in Bezug auf die Performance. Zu guter Letzt zieht sie ein Fazit und überlegt sich den Sinn der angewandten Methode.

Zur Arbeit gehören dieses Dokument sowie die Beispielimplementation. Beides kann von `https://github.com/foyan/ColorizedBarCodec` bezogen werden.

Der Autor bedankt sich bei Dr. Alexander Herrigel für die Durchführung des Seminars.

2 Einführung in das Thema

2.1 Existierende Technologien

Seit den 1970er-Jahren [4] sind Barcodes aus Logistikbetrieben, Supermärkten etc. nicht mehr wegzudenken. Sie bilden das Rückgrat der automatisierten Warenbewirtschaftung. Das Scanning von eindimensionalen Barcodes wie in Abbildung 2.1 ist milliardenfach erprobt und in der Entwicklung eigentlich abgeschlossen.

Die Informationsdichte von eindimensionalen Barcodes ist aber sehr beschränkt. Beispielsweise erlaubt die eindimensionale Symbologie UPC-A maximal eine Billion eindeutiger Barcodes [8], was gemäss

$$\log_2 10^{12} = 39.863\ldots$$

einer maximalen Kapazität von weniger als 5 Bytes entspricht.

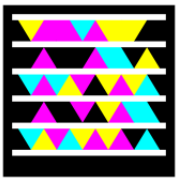
Aus diesem Grund wurden mit dem Aufkommen von billigen und sehr akkuraten Kameras in mobilen Devices zweidimensionale Symbologien entwickelt, welche es zumindest erlauben, ganze URLs von Webseiten oder ähnliches zu codieren. Google beispielsweise vermarktet den in Abbildung 2.2 ersichtlichen, ursprünglich von Toyota entwickelten *Quick Response* Code (QR) [7]. Dieser erlaubt es immerhin, maximal 2953 Bytes zu codieren [7].

Um die Kapazität weiter zu steigern, ist es nötig, eine dritte Dimension einzuführen. Weil aber Hologramme und 3D-Kameras noch nicht marktreif sind, liegt es nahe, die dritte Dimension als Farbe zu codieren. Microsoft geht in seiner Barcode-Technologie *High Capacity Color Barcode (HCCB)* seit 2007 [6] genau diesen Weg [3]. Ein Beispiel ist in Abbildung 2.3 abgebildet. Die maximale Kapazität von HCCB kann nicht genau genannt werden, weil die Barcodes keine definierte Grösse und Skalierung haben. Labortests scheinen aber gezeigt zu haben, dass es möglich ist, circa 2000 Bytes auf eine

Abbildung 2.1: Eindimensionaler Barcode (Quelle: Wikipedia)

Abbildung 2.2: QR Code (Quelle: Wikipedia)

Abbildung 2.3: High Capacity Color Barcode (Quelle: Wikipedia)



Fläche zu codieren, welche nicht viel grösser als eine Ein-Cent-Münze ist. Diese Fläche scheint sich mit einem 600-dpi-Laserdrucker herstellen zu lassen [3].

2.2 Vorgeschlagene Methode

Rana, Kawsar, Rabbani, Rashid und Achmed schlagen nun eine Methode vor, welche sich durch eine hohe Kapazität sowie ein sehr schnelles Leseverfahren auszeichnen soll [2]. Die grundlegenden Eckwerte sind die folgenden:

- Es werden 16 verschiedene Farben verwendet. Jede dieser Farben wird bijektiv einer Sequenz aus 4 Bits (Nibbles) zugeordnet (siehe Tabelle 2.1, [2]).
- Jedes Byte wird auf ein Zeichen abgebildet (siehe [2]).
- Die Fläche des Barcodes wird in ein 102×100 Zellen fassendes Raster aufgeteilt (im folgenden als Pixel bezeichnet). Bei einer angenommenen realen Grösse des Barcodes von 1.5×1.15 Zoll [2] ergibt sich daraus eine horizontale Auflösung von 68 DPI und eine vertikale Auflösung von 87 DPI.
- Jeweils 6 benachbarte Pixel in einem 3×2-Feld codieren nun ein Byte, wobei ein ein vertikal gespiegeltes `L` für den ersten Nibble und ein horizontal gespiegeltes `L` für den zweiten Nibble entstehen (Abbildung 2.4). Über die Reihenfolge der

Tabelle 2.1: Zuordnung der Farben zu den Nibbles

Wert	Nibble (binär)	RGB-Wert
0	0000	rgb(255, 000, 000)
1	0001	rgb(000, 128, 000)
2	0010	rgb(000, 000, 025)
3	0011	rgb(165, 042, 042)
4	0100	rgb(128, 000, 000)
5	0101	rgb(100, 149, 237)
6	0110	rgb(255, 165, 000)
7	0111	rgb(255, 255, 000)
8	1000	rgb(238, 130, 238)
9	1001	rgb(128, 128, 128)
10	1010	rgb(000, 000, 128)
11	1011	rgb(000, 000, 000)
12	1100	rgb(173, 255, 047)
13	1101	rgb(138, 043, 226)
14	1110	rgb(128, 000, 128)
15	1111	rgb(095, 158, 160)

Wertigkeit der Nibbles wird keine Vorschrift gemacht – die abgebildete Rot/Grün-Kombination könnte also sowohl als `0000 0001` = 1 als auch als `0001 0000` = 16 interpretiert werden.

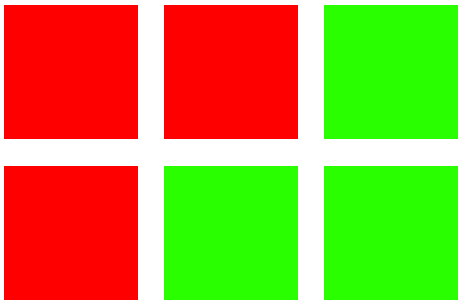
Weil je nach codierter Information die Länge der Codierung variiert, schlägt [2] vor, die ersten Nibbles als Header zu reservieren[1], welche als Information nur die Länge der nachfolgenden Informationen enthalten.

Die maximale Kapazität des Barcodes ist darum

$$C = \frac{102\,\mathrm{px} \cdot 100\,\mathrm{px}}{6\,\mathrm{px}^2\,/\,\mathrm{byte}} - 2\,\mathrm{bytes} = 1698\,\mathrm{bytes}\,.$$

[1] Bezüglich der Länge des Headers werden in [2] widersprüchliche Aussagen gemacht. Der Text reserviert die ersten 12 Pixel, was einer Länge von 2 Bytes entsprechen würde, wogegen die dazugehörige Abbildung eine Länge von 9 Pixeln kolportiert, was einer Länge von 1.5 Bytes entsprechen würde. Im Nachfolgenden wird darum eine Headerlänge von 2 Bytes angenommen.

Abbildung 2.4: Ein Byte in der vorgeschlagenen Methodik



3 Randbedingungen

Das beschriebene, untersuchte und implementierte Verfahren [2] impliziert in sich schon mehrere Randbedingungen beziehungsweise Einschränkungen bezüglich allgemeiner Brauchbarkeit, deren Diskussion in Kapitel 5.4 aufgenommen wird.

Vorauseilend kann schon einmal gesagt werden, dass in Punkto Kapazitätserweiterung kein Fortschritt erreicht wird, weil der *QR*-Code intrinsisch schon eine höhere Kapazität aufweist, ohne allerdings den Vorteil der Farbdimension für sich reklamieren zu können.

Diese Arbeit hat darum explizit nicht das Ziel, eine wirkliche Alternative oder eine sinnvolle Entwicklungsrichtung in Sachen Barcodes aufzuzeigen. Sie ist eher als Schulbeispiel zur Umsetzung einer einfachen Algorithmik mittels Parallelisierung zu verstehen.

4 Auswahl der Testdaten für die Arbeit

Die Aufgabe des Algorithmus ist, Text in ein Bild umzuformen und wieder zurück. Dementsprechend werden drei Texte ausgewählt, welche zusammen mit ihren entsprechenden Barcodes die Testdaten bilden:

1. Um den minimalen Fall zu testen, wird der leere Text ausgewählt. Der korrespondierende Barcode ist in Abbildung 4.1 zu sehen.

2. Um einen durchschnittlichen Fall zu testen, wird die Definition der Aufklärung von Immanuel Kant aus dessen Essay *Beantwortung der Frage: Was ist Aufklärung?* von 1784 [1] verwendet:

   > Aufklärung ist der Ausgang des Menschen aus seiner selbstverschuldeten Unmündigkeit. Unmündigkeit ist das Unvermögen, sich seines Verstandes ohne Leitung eines anderen zu bedienen. Selbstverschuldet ist diese Unmündigkeit, wenn die Ursache derselben nicht am Mangel des Verstandes, sondern der Entschliessung und des Muthes liegt, sich seiner ohne Leitung eines anderen zu bedienen. Sapere aude! Habe Muth, dich deines eigenen Verstandes zu bedienen! ist also der Wahlspruch der Aufklärung. [5]

   Die Länge dieses Textes beträgt (nach Ersetzung von `ß` durch `ss` sowie dem Einfügen einiger Zeilenumbrüche wegen der besseren Lesbarkeit) exakt 500 Bytes, was einer Ausnutzung der Kapazität von 29.4 Prozent entspricht. Der korrespondierende Barcode ist in Abbildung 4.2 zu sehen.

3. Um den maximalen Fall zu testen, wird ebendieser Text solange wiederholt, bis die maximale Kapazität erreicht ist. Der korrespondierende Barcode ist in Abbildung 4.3 zu sehen.

Abbildung 4.1: Der leere Barcode

Abbildung 4.2: Die Antwort auf die Frage: Was ist Aufklärung? als Barcode

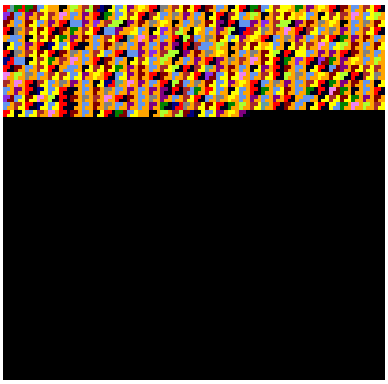
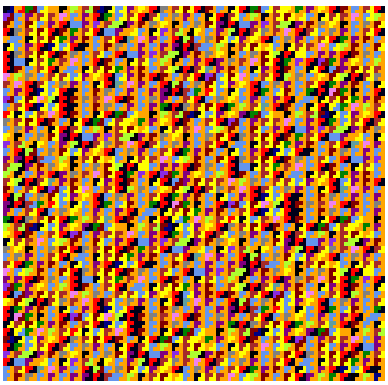


Abbildung 4.3: Möglichst viel Kant in einem Barcode



5 Algorithmische Beschreibung des Verfahrens

5.1 Codieren des Barcodes

Der folgende Algorithmus beschreibt den Codierungsvorgang eines Barcodes (seien `text` der zu codierende Text, `width` und `height` die Breite und Höhe des Barcodes, `color()` die Nibble-Farbe-Abbildung und `bc` der resultierende Barcode):

```
algo encode:
    for i in [0..|text|-1]:
        var x = (i*3) mod width
        var y = (i*3) / width * 2
        bc[x, y] = bc[x+1, y] = bc[x, y+1] = color(text[i].low)
        bc[x+1, y+1] = bc[x+2, y+1] = bc[x+2, y] = color(text[i].high)
```

5.2 Decodieren des Barcodes

Der folgende Algorithmus beschreibt den Decodierungsvorgang eines Barcodes (seien `bc` der zu dcodierende Barcode, `width` und `height` die Breite und Höhe des Barcodes, `nibble()` die Farbe-Nibble-Abbildung, `detect_color()` der weiter unten beschriebene Error-Correction-Algorithmus und `text` der resultierende Text):

```
algo decode:
    for (y, x) in [0..height-1] x [0..width-1] step (2, 3):
        var c1 = detect_color(bc[x, y], bc[x+1, y], bc[x, y+1])
        var c2 = detect_color(bc[x+1, y+1], bc[x+2, y+1], bc[x+2, y])
        text.append(nibble(c2) nibble(c1))
```

5.3 Error Correction

Die in [2] vorgeschlagene probabilistische Error Detection und Correction-Methode funktioniert folgendermassen (seien `m`, `s1`, `s2` die Farben an den entsprechenden Position innerhalb des `Ls`):

```
algo detect_color:
    if m = s1:
        return m
```

```
if s1 = s2:
    return s1
return s2
```

5.4 Bewertung des Verfahrens

An der von [2] proklamierten Einfachheit und `computational efficiency` des Verfahrens gibt es gar nichts auszusetzen: Unter der gültigen Annahme, dass sowohl `color()` als auch `nibble` als reine Hashtable-Lookup-Funktionen eine asymptotische Laufzeitkomplexität von (sei x der Text oder der Barcode):

$$f(x) \in \mathcal{O}(1)$$

haben, beträgt die asymptotische Laufzeitkomplexität sowohl von `encode` wie auch von `decode`

$$f(x) \in \mathcal{O}(|x|).$$

Durch diese verfahrenstechnische Effizienz werden allerdings mehrere Mängel in Kauf genommen, die sich schnurstracks auf die Effektivität des Verfahrens auswirken:

1. Der Platz, den ein Byte an Information einnehmen kann, ist steht immer in einem festen Verhältnis zur gesamten Grösse des Barcodes (nämlich 6 Pixel). Dadurch hat ein resultierender Barcode immer automatisch die maximal mögliche Informationsdichte (was in den Abbildungen 4.1 und 4.2 sehr schön ersichtlich ist: da die zu codierenden Informationen weniger sind als die maximal mögliche Kapazität, bleibt ein geographisch zusammenhängender Teil des Barcodes ungenutzt, sprich schwarz).

   Reifere Verfahren nutzen hingegen immer den ganzen zur Verfügung stehenden Platz, indem sie dynamisch die Informationsdichte ausdünnen. Dadurch vergrössern sie die Toleranz gegenüber der Ungenauigkeit während des Lesens (welche in der Realität bei Scans oder Fotografien natürlich immer da ist).

   Das Verfahren könnte in dieser Hinsicht natürlich verbessert werden, weil die Länge der Information aus dem Header ersichtlich ist. Daraus liesse sich die physische Grösse eines Pixels berechnen. Allerdings ist der Header selber als mehrere Pixel codiert. Deshalb müsste trotzdem ein Verfahren implementiert werden, welches quasi die Grösse eines Pixels rekonstruiert – was sich natürlich saftig in einer höheren Gesamtkomplexität niederschlagen würde.

2. Das Verfahren beinhaltet keinerlei Paritätsprüfung der Daten. Abgesehen davon, dass die Farbwerte der gescannten Pixel exakt denjenigen in Tabelle 2.1 entsprechen müssen (was bei einem Scan in der Realität nur schon wegen den variierenden Lichtverhältnissen wegen sowieso utopisch ist), würde der als `Error Correction und Detection`-Algorithmus [2] kolportierte Algorithmus `detect_color()` bei drei

verschiedenen (und gemäss Tabelle 2.1 gültigen!) Farbwerten unreflektierterweise einfach den letzten der drei Werte als richtig erachten.

Das Verfahren könnte verbessert werden, indem es um eine typische Paritätsprüfungsmethode ergänzt werden würde – beispielsweise `CRC`. Die aymptotische Komplexität würde nicht darunter leiden; der Seiteneffekt wäre allerdings, dass nicht mehr zwei Nibbles ein ganzes Byte ergeben würden und darum das Mapping zwischen Bytes und Zeichen massiv komplizierter würde.

6 Analyse, Design und Implementierung des Verfahrens

6.1 Parallelisierung

Die Codierungs- und Decodierungs-Algorithmen selber sind einigermassen simpel. Der parallelisierbare Teil der beiden Algorithmen besteht allerdings nur aus dem Mapping-Teil, da der zu codierende Text oder das zu decodierende Bild natürlich nur auf jeweils einem Node des Clusters vorliegen.

Da der Codierungs-Algorithmus mehr oder weniger die Umkehrfunktion des Decodierungs-Algorithmus ist, kann die Infrastruktur der beiden geteilt werden. Dadurch ist es möglich, die Parallelisierung vollständig aus den Algorithmen zu extrahieren: Sei `algo` einer der beiden besprochenen Algorithmen mit diesen Teilalgorithmen:

- `init`
- `get_sliced_input`
- `process_slice`
- `collect_slice`
- `finalize`,

dann sieht die parallelisierte Ausführung von `algo` auf dem Master-Node folgendermassen aus:

```
algo.init()
for each slave node:
    var in = get_sliced_input(slave)
    invoke_slave(in)
var my_in = get_my_input()
algo.process_slice(my_input)
result += algo.collect_slice(my_input)
for each slave node:
    result += algo.collect_slice(slave)
algo.finalize()
```

Die parallelisierte Ausführung auf einem Slave-Node besteht im Wesentlichen aus der Ausführung von `algo.process_slice()`.

Weil die Ausführung der Parallelisierung nun nichts mehr mit dem Ausführen der Algorithmen zu tun hat, kann nun natürlich das Strategy-Pattern angewandt werden, um die Ausführungen austauschbar zu machen. Im folgenden ist der Code der Methode `task_runner_factory::create_task_runner` abgebildet, welche anhand des zweiten Kommandozeilenparameters entweder einen `MPI`- oder einen simplen nichtparallelisierten `task_runner` herstellt:

```
task_runner* task_runner_factory::create_task_runner(int argc, char* argv
    []) {

  if (argc > 2 && strcmp(argv[2], "mpi") == 0) {
    return new mpi_task_runner();
  }

  return new simple_task_runner();

}
```

6.2 Kommunikation der Nodes

Damit die Algorithmen über einen Strang geschlagen werden können, müssen sie von einer gemeinsamen Basisklasse ableiten – `task`:

```
class task {

public:
  virtual void init(int argc, char* argv[]) = 0;
  virtual void* get_sliced_input(int slice_index, int slice_count) = 0;
  virtual void* process_slice(void* input) = 0;
  virtual void collect_slice(void* slice, int slice_index, int
      slice_count) = 0;
  virtual void finalize() = 0;

  virtual void pack_input(pack&, void*) = 0;
  virtual void pack_output(pack&, void*) = 0;
  virtual void* unpack_input(pack&) = 0;
  virtual void* unpack_output(pack&) = 0;

  virtual ~task() { }
};
```

Damit der operierende `task_runner` nicht wissen muss, welche Datentypen er in der `MPI`-World herumschickt (und die Basisklasse dies natürlich noch weniger weiss), funktioniert die In-Process-Kommunikation über `void`-Pointer, wovon die konkreten `task`s die korrekten (und nur für sie selber relevanten) Typen durch Casting zurückgewinnen können.

Ein Problem bleibt aber: Die verwendete `boost_serialization`-Library kann nur bekannte Typen serialisieren. Aus diesem Grund gibt es die `pack`-Klasse, welche ein

neutrales Paket machen kann, je nachdem ob `string`s oder Pixel verschickt werden sollen:

```
class pack {

public:
  string Str;
  vector<pair<rgb, rgb>> Pixels;

};

namespace boost {
  namespace serialization {

    template<typename Archive>
    void serialize (Archive& ar, pack& r, const unsigned int version) {
      ar & r.Str;
      ar & r.Pixels;
    }

  }
}
```

Die verschiedenen `(un)pack_*`-Methoden der einzelnen `task`s werden dann vom `task_runner` entsprechend an der richtigen Stelle aufgerufen, um die Pakete entsprechend vor- oder nachzubereiten.

6.3 Mapping

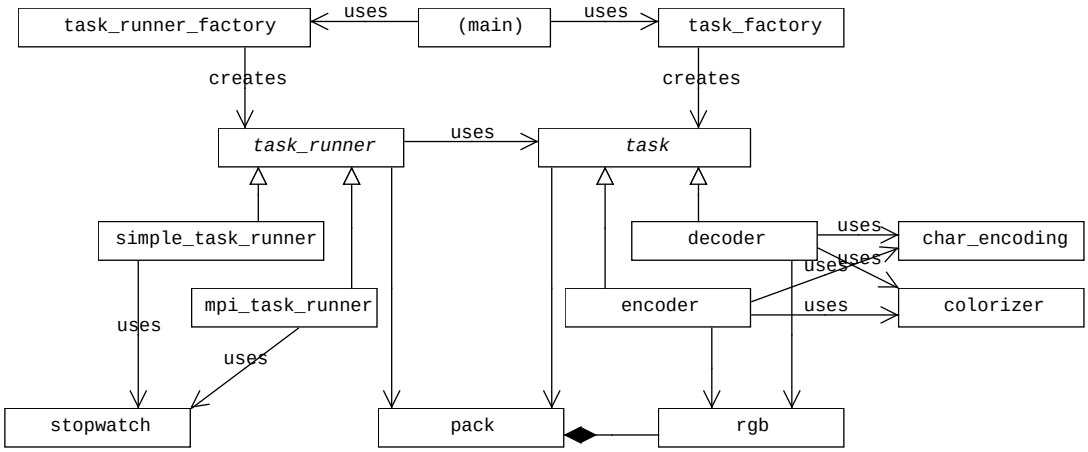
Das Mapping von Farben zu Nibbles ist bijektiv, ergo braucht es für die Implementation eine bidirektionale `map<>`[1]. In `boost` wird man mit der `bimap<>` natürlich fündig. Da diese Map von `unsigned short` zur benutzerdefinierten Klasse `rgb` abbildet und wieder zurück, muss eine natürliche Ordnung auf `rgb` implementiert werden, damit `bimap<>` die Eindeutigkeit garantieren kann. Dies wird über die Spezialisierung des Templates `std::less<>` gelöst:

```
namespace std {

  template<> struct less<rgb> : binary_function<rgb, rgb, bool> {
    bool operator() (const rgb& a, const rgb& b) const {
      if (a.r() < b.r()) {
        return true;
      }
      if (a.r() > b.r()) {
        return false;
      }
      if (a.g() < b.g()) {
        return true;
```

[1] Auf ein explizites Mapping von Bytes zu Zeichen wie in [2] vorgeschlagen wurde verzichtet – es scheint keinen sinnvollen Grund zu geben, in dieser Beziehung nicht die normale `ASCII`-Codierung zu verwenden.

Abbildung 6.1: Architektur




```
      }
      if (a.g() > b.g()) {
        return false;
      }
      return a.b() < b.b();
    }
  };

}
```

6.4 Design-technische Übersicht

Der Rest der Implementation ist eigentlich Schema F – die besprochenen Algorithmen werden mehr oder weniger 1:1 implementiert, das Übrige das Bemühen von `libpng` für IO mit dem Bild und das Zershreddern und Zusammenführen von `string`s und `vector<>`s. Stellvertretend illustriert Abbildung 6.1 die Klassenarchitektur der Implementation.

7 Vergleich der Performancezeiten für ein, zwei oder mehrere CPU-Cores

In Tabelle 7.1 sind die durchschnittlichen Ausführungszeiten pro Testfall (minimal, durchschnittlich und maximal), Algorithmus und Anzahl Cores angegeben. Die Zeiten sind in Millisekunden, und es wurden jeweils fünf Ausführungen durchgeführt.

Das Tests wurden auf einem virtuellen Debian Wheezy mit 1 GB RAM und 4 vom Hostbetriebssystem (Windows 8) zugewiesenen Prozessorkernen durchgeführt.

Es sticht ins Auge, dass die Ausführungszeit besser ist, je *weniger* parallelisiert wird. Dies kann dadurch erklärt werden, dass der parallelisierbare Teil des Verfahrens trivial ist und gegenüber dem Kommunikations-Overhead sich praktisch in Nichts auflöst.

Der notorisch bessere Performance von `decode` gegenüber `encode` lässt sich dadurch erklären, dass das Schreiben eines `png`-Files mehr Zeit benötigt als das Lesen desselben.

Tabelle 7.1: Durchschnittliche Ausführungszeiten [ms]

Testfall	1 Core	2 Cores	4 Cores
minimal – `encode`	1.306	2.367	4.513
minimal – `decode`	0.721	1.780	4.879
durchschnittlich – `encode`	2.801	4.979	9.262
durchschnittlich – `decode`	1.206	4.303	6.830
maximal – `encode`	5.582	15.965	17.742
maximal – `decode`	3.023	14.462	15.023

8 Zusammenfassung

Zusammenfassend kann man sagen, dass wir einiges über Barcode-Methodiken, -Verfahren und -Formate als solche gelernt haben. Insbesondere haben wir erfahren, dass sich das besprochene Verfahren gut dafür eignet, das grundsätzliche Prinzip zu illustrieren – im praktischen Einsatz ist es aber wegen seiner gravierenden Mängel und Unfertigkeiten nicht zu gebrauchen. Es gibt natürlich Möglichkeiten, dieses Verfahren zu verbessern – es würde dann aber einiges an seiner Einfachheit einbüssen.

Des weiteren haben wir gesehen, dass sich dieses Verfahren nur bedingt sinnvoll parallelisieren lässt – dies vor allem darum, weil das Auseinandernehmen und Zusammenführen des Problemfelds den Geschwindigkeitszuwachs des parallelisierbaren Teils wieder zunichte macht. Anders würde es wohl aussehen, wenn wir versuchen würden, Milliarden von Bytes in einen einzigen Barcode zu quetschen.

Zu guter Letzt haben wir gesehen, wie sich ein Algorithmus von seiner parallelen Ausführung in der Implementation trennen lässt – mit den besprochenen Mühsamitäten durch das generische Transportieren der Payload-Daten.

Durch diese Trennung wäre es möglich, die `MPI`-basierende Parallelisierung sehr einfach durch eine Thread-basierte zu ersetzen. Dies dürfte dann performancemässig einiges bringen, weil die Serialisierung und Deserialisierung der Daten damit entfällt. Die Synchronisation der Ressourcen müsste natürlich trotzdem sichergestellt werden – aber dafür existieren minimalinvasive Pattern und Hilfsmittel wie beispielsweise Reader/Writer-Locks.

Literaturverzeichnis

[1] Immanuel Kant. Beantwortung der Frage: Was ist Aufklärung? *Berlinische Monatsschrift*, 4:481–494, 1784.

[2] Md. Mashud Rana, M. E. Kawsar, M. E. Rabbani, S. M. M. Rashid und K. E. U. Ahmed. An Enhanced Two-Dimensional Color Barcode System. *Journal of Emerging Trends in Engineering and Applied Sciences (JETEAS)*, 2(1):126–131, 2011.

[3] Microsoft Research. About High Capacity Color Barcode Technology. `http://research.microsoft.com/en-us/projects/hccb/about.aspx`, 2013. [Online; accessed 14-June-2013].

[4] Wikipedia. Barcode — Wikipedia, The Free Encyclopedia. `http://en.wikipedia.org/w/index.php?title=Barcode&oldid=558925085`, 2013. [Online; accessed 14-June-2013].

[5] Wikipedia. Beantwortung der Frage: Was ist Aufklärung? — Wikipedia, Die freie Enzyklopädie. `http://de.wikipedia.org/w/index.php?title=Beantwortung_der_Frage:_Was_ist_Aufkl%C3%A4rung%3F&oldid=118761194`, 2013. [Online; Stand 14. Juni 2013].

[6] Wikipedia. High Capacity Color Barcode — Wikipedia, The Free Encyclopedia. `http://en.wikipedia.org/w/index.php?title=High_Capacity_Color_Barcode&oldid=559150607`, 2013. [Online; accessed 14-June-2013].

[7] Wikipedia. QR code — Wikipedia, The Free Encyclopedia. `http://en.wikipedia.org/w/index.php?title=QR_code&oldid=559415969`, 2013. [Online; accessed 14-June-2013].

[8] Wikipedia. Universal Product Code — Wikipedia, The Free Encyclopedia. `http://en.wikipedia.org/w/index.php?title=Universal_Product_Code&oldid=558763746`, 2013. [Online; accessed 14-June-2013].

Abbildungsverzeichnis

Tabellenverzeichnis