Praktikum 2 zu Parallele Programmierung

Fabian Czappa

TECHNISCHE UNIVERSITÄT DARMSTADT

Wintersemester 2023/2024
05. Dezember 2023

Allgemeines

- Die Standardsprache im Quellcode ist Englisch, weswegen auch die Typen/Methoden so benannt sind.
- Es gibt Fußnoten, wenn Konzepte das erste Mal verwendet werden; dort können Sie sich weiter informieren.
- *Es gibt Hinweise (zu gutem Programmierstil oder anderem) in kursiver Schrift.*

Bepunktung der Aufgaben

- Bei jeder Teilaufgabe stehen die erreichbaren Punkte dabei, welche die Anzahl an funktionalen Tests/Laufzeittests für die jeweige Teilaufgabe angibt.
- Für die funktionalen Tests gilt: Sollte der Test fehlschlagen, nicht kompilieren oder länger als **eine Minute** brauchen, erhalten Sie keinen Punkt dafür. Sollte der Test innerhalb von **einer Minute** korrekt durchlaufen, erhalten Sie dafür einen Punkt.
- Für die Laufzeittests gilt: Jeder solche enthält auch eine funktionale Überprüfung. Sollte der Test nicht kompilieren, länger als **eine Minute** brauchen oder die funktionale Komponente fehlschlagen, erhalten Sie dafür keinen Punkt. Sollte der Test innerhalb von **einer Minute** ein korrektes Skalierungsverhalten anzeigen, erhalten Sie dafür einen Punkt.

Zusätzliche Hinweise

- Nennen sie **keine** Ihrer Dinge (z.B. Funktionen, Dateien, Klassen) mit `fabian` als Bestandteil vom Namen.
- Ihnen steht frei weitere Funktionalität zu implementieren, solange nicht explizit anders geregelt.
- Dieses Praktikum baut auf dem ersten Praktikum auf. Wir haben einen Lösungsvorschlag in Moodle hochgeladen – passen Sie dahingehend Ihre Lösung an oder übernehmen Sie diesen (insbesondere neue Funktionalitäten).
- Achen Sie darauf, dass Sie nur standardkonformes C++/OpenMP benutzen. Das Benutzen von compiler-/laufzeit-/betriebssystemspezifischen Erweiterungen und Funktionalitäten abseits der von uns vorgegebenen Dinge (Frameworks für die funktionalen Tests und die Laufzeittests, Benutzung von Typen wie `std::uint32_t` etc.) können zu Punktabzug führen.
- Bitte beachten Sie, dass die Tutor:innen keine bindenden Aussagen treffen.

ParProg
Nachname, Vorname: _____ Matrikelnummer: ␣␣␣␣␣␣␣␣

Aufgabe 1: Bereitstellungen der notwendigen (Daten-)Klassen (Gesamt: 2 Punkte)

Die hier geforderten Funktionen werden in genau dieser Form getestet; eine solche ist durch die `Teletype` Schrift gekennzeichnet.

1a) Berechnung des großen Hashs (1 Punkt)

Die Berechnung der statischen Funktion `Hash::hash(std::uint64_t)` ist definiert als:

$$\texttt{result} = ((((\texttt{input} >> 14) + A) + ((\texttt{input} << 54) \oplus B)) << 4) \oplus ((\texttt{input mod } C) * 137)$$

wobei $\oplus$ bitweises XOR ist und es gilt:

$$A = 5\,647\,095\,006\,226\,412\,969$$

$$B = 41\,413\,938\,183\,913\,153$$

$$C = 6\,225\,658\,194\,131\,981\,369$$

Implementieren Sie die Funktionalität.
Führen Sie Zwischenschritte für die Berechnung ein.
Für eine bessere Lesbarkeit können Sie Trennzeichen in den Zahlen benutzen.[1]

1b) Ändern der BitmapImage-Klasse (1 Punkt)

Ändern Sie Ihre `BitmapImage`-Klasse dahingehend, dass Sie für alles, was mit Größen und Indizes zu tun hat, `std::uint32_t` benutzen. Ändern Sie außerdem Ihre Laufzeitchecks ab, sodass Bilder bis zur Größe von `1 024 000` auf `1 024 000` (beide inklusiv) erstellt werden können.
Zu ladende und speichernde Bilder unterliegen weiterhin den Einschränkungen des Bitmap-Formats.
Für Ihre Testfälle können Sie davon ausgehen, dass niemals mehr als 1 GB an reinen Daten pro Bild gebraucht wird.

Das betrifft vermutlich die Angabe der Größe im Konstruktor, das Indizieren der Pixel in `get_pixel` und `set_pixel`, die Rückgabe der Dimensionen in `get_height` und `get_width` und den dependent name `index_type`.
Da die Frage bei der Vorstellung aufkam: Sie müssen auch keine dünn-besetzten Bilder o.ä. handhaben können.

[1] `https://en.cppreference.com/w/cpp/language/integer_literal`

Aufgabe 2: Parallelisieren der Algorithmen (Gesamt: 10 Punkte)

Die hier geforderten Funktionen werden in genau dieser Form getestet; eine solche ist durch die `Teletype` Schrift gekennzeichnet.

Achtung: Hier benutzen wir das erste Mal Laufzeittests (benchmarks) zum Überprüfen der korrekten Implementierung. Wir postulieren dafür, dass die Parallelisierung des Codes eine signifikante Laufzeitsenkung mit sich bringt. Parallelisieren Sie jeweils nicht mehr als verlangt, weil das die Laufzeittests kaputt machen kann.

2a) Parallelisierung mehrere Reihen (2 Punkte)

In FES gibt es keine Abhängigkeiten bei der Verschlüsselung mehrerer Reihen.
Stellen Sie die statische Funktion `FES::encrypt_parallel_coarse` bereit, welche funktional `FES::encrypt` gleicht, aber einen dritten Parameter vom Typ `int` akzeptiert. Dieser soll den Standardwert 1 haben und gibt an, wie viele Threads gleichzeitig verschiedene Reihen verschlüsseln sollen. Sie dürfen annehmen, dass dieser Wert immer zwischen 1 und 96 ist. Parallelisieren Sie die Verschlüsselung von mehreren Reihen mit OpenMP und benutzen in dem Konstrukt die übergebene Anzahl Threads.

Achten Sie darauf, dass die Funktion auch mit zwei Argumenten aufgerufen werden kann.[2] Das gilt auch für spätere Aufgaben.

2b) Parallelisierung innerhalb einer Reihe (2 Punkte)

In FES gibt es keine Abhängigkeiten innerhalb einer Reihe zwischen der Ausgabe der verschlüsselten Blöcke und der Eingabe der unverschlüsselten Blöcke (für den jeweils nächsten Block).
Stellen Sie die statische Funktion `FES::encrypt_parallel_fine` bereit, welche funktional `FES::encrypt` gleicht, aber einen dritten Parameter vom Typ `int` akzeptiert. Dieser soll den Standardwert 1 haben und gibt an, wie viele Threads gleichzeitig Blöcke innerhalb einer Reihe verschlüsseln sollen. Sie dürfen annehmen, dass dieser Wert immer zwischen 1 und 96 ist. Parallelisieren Sie die Verschlüsselung einer Reihe mit OpenMP und benutzen in dem Konstrukt die übergebene Anzahl Threads.

2c) Kollabiertes Parallelisieren der Verschlüsselung (2 Punkte)

Stellen Sie die statische Funktion `FES::encrypt_parallel` bereit, welche funktional `FES::encrypt` gleicht, aber einen dritten Parameter vom Typ `int` akzeptiert. Dieser soll den Standardwert 1 haben und gibt an, wie viele Threads gleichzeitig verschlüsseln sollen. Sie dürfen annehmen, dass dieser Wert immer zwischen 1 und 96 ist. Parallelisieren Sie die Verschlüsselung (sowohl mehrerer Reihen als auch innerhalb einer Reihe) mit OpenMP und benutzen in dem Konstrukt die übergebene Anzahl Threads.
Hinweis: Da wir in der Vorlesung keinen verschachtelten Parallelismus behandeln, brauchen Sie diesen hier nicht zu verwenden.

2d) Parallelisiertes Transponieren (1 Punkt)

Stellen Sie die Funktion `BitmapImage::transpose_parallel` bereit, welche funktional `BitmapImage::transpose` gleicht, aber einen Parameter vom Typ `int` akzeptiert. Dieser soll den Standardwert 1 haben und gibt an, wie viele Threads gleichzeitig transponieren sollen. Sie dürfen annehmen, dass dieser Wert immer zwischen 1 und 96 ist. Parallelisieren Sie das Transponieren mit OpenMP und benutzen in dem Konstrukt die übergebene Anzahl Threads.
Hinweis: Da wir in der Vorlesung keinen verschachtelten Parallelismus behandeln, brauchen Sie diesen hier nicht zu verwenden.

[2] `https://en.cppreference.com/w/cpp/language/default_arguments`

2e) Parallelisiertes Entfärben (1 Punkt)

Stellen Sie die Funktion `BitmapImage::get_grayscale_parallel` bereit, welche funktional `BitmapImage::get_grayscale` gleicht, aber einen Parameter vom Typ `int` akzeptiert. Dieser soll den Standardwert 1 haben und gibt an, wie viele Threads gleichzeitig entfärben sollen. Sie dürfen annehmen, dass dieser Wert immer zwischen 1 und 96 ist. Parallelisieren Sie das Entfärben mit OpenMP und benutzen in dem Konstrukt die übergebene Anzahl Threads.
Hinweis: Da wir in der Vorlesung keinen verschachtelten Parallelismus behandeln, brauchen Sie diesen hier nicht zu verwenden.

2f) Parallelisiertes *Irgendwas* (2 Punkte)

Stellen Sie die Funktion `BitmapImage::mystery_parallel` bereit, welche funktional `BitmapImage::mystery` gleicht, aber einen Parameter vom Typ `int` akzeptiert. Dieser soll den Standardwert 1 haben und gibt an, wie viele Threads gleichzeitig *irgendwas* sollen. Sie dürfen annehmen, dass dieser Wert immer zwischen 1 und 96 ist. Parallelisieren Sie den gegebenen Algorithmus mit OpenMP und benutzen in dem Konstrukt die übergebene Anzahl Threads. Stellen Sie sicher, dass sowohl sehr "hohe", als auch sehr "breite"Bilder sinnvoll verarbeitet werden können.
Hinweis: Da wir in der Vorlesung keinen verschachtelten Parallelismus behandeln, brauchen Sie diesen hier nicht zu verwenden.

Aufgabe 3: Schlüssel (Gesamt: 8 Punkte)

Die hier geforderten Funktionen werden in genau dieser Form getestet; eine solche ist durch die `Teletype` Schrift gekennzeichnet.

Achtung: Hier benutzen wir das erste Mal Laufzeittests (benchmarks) zum Überprüfen der korrekten Implementierung. Wir postulieren dafür, dass die Parallelisierung des Codes eine signifikante Laufzeitsenkung mit sich bringt. Parallelisieren Sie jeweils nicht mehr als verlangt, weil das die Laufzeittests kaputt machen kann.

3a) Berechnung neuer Schlüssel (1 Punkt)

Stellen Sie die statische Funktion `produce_new_key` in der Klasse `Key` bereit, welche einen `key_type` als Argument nimmt und einen `key_type` zurück gibt. Gehen Sie dabei wie folgt vor:
Fassen Sie jeweils acht aufeinander folgende `std::uint8_t` zu einem `std::uint64_t` zusammen (dies gibt Ihnen also sechs Pakete). Dies machen Sie so, dass niederwertige Byte im Key auch niederwertige Byte im `std::uint64_t` sind. Ausgehend von jedem dieser Werte berechnen Sie mit `Hash::hash` einen neuen Wert und schreiben den an die entsprechende Stelle im Rückgabewert.

Wenn Ihr System Little Endian benutzt[3], können Sie die Byte einfach kopieren[4].
*Achten Sie darauf, dass Sie **nicht** mit `reinterpret_cast` arbeiten (den kennen Sie auch gar nicht), da das undefiniertes Verhalten produziert. Im Endeffekt möchten wir diesen nachbauen.*

3b) Hash eines Schlüssels (1 Punkt)

Stellen Sie die statische Funktion `hash` in der Klasse `Key` bereit, welche einen `key_type` akzeptiert und ein `std::uint64_t` zurück gibt. Berechnen Sie den Wert wie folgt:
Teilen Sie den Schlüssel in sechs Pakete und berechnen Sie den jeweiligen Hash (genau wie in Aufgabe 3a)). Kombinieren Sie dann mit `Hash::combine_hashes` die Werte miteinander und geben das Ergebnis zurück.

Sie können die Variable, welche Sie zum Kombinieren benutzen, mit 0 initialisieren, da das neutral bgzl. XOR ist.

3c) Finden des minimalen Hashs (2 Punkte)

Stellen Sie die statische Funktion `get_smallest_hash` in der Klasse `Key` bereit. Diese akzeptiert als Parameter ein `std::span`[5] mit konstanten `key_type` und gibt ein `std::uint64_t` zurück. Die Funktion ermittelt den minimalen Hash aller übergebener Schlüssel und gibt diesen zurück. Falls keine Werte übergeben werden, soll das maximal mögliche Wert[6] zurückgegeben werden.

3d) Parallelisierung des minimalen Hashs (1 Punkt)

Stellen Sie die statische Funktion `get_smallest_hash_parallel` in der Klasse `Key` bereit, welche funktional `Key::get_smallest_hash` gleicht, aber einen zweiten Parameter vom Typ `int` akzeptiert. Dieser soll den Standardwert 1 haben und gibt an wie viele Threads gleichzeitig den minimalen Wert suchen sollen. Sie dürfen annehmen, dass dieser Wert immer zwischen 1 und 96 ist. Parallelisieren Sie die Suche mit OpenMP und benutzen Sie in dem Konstrukt die übergebene Anzahl Threads.

[3] `https://en.cppreference.com/w/cpp/types/endian`
[4] `https://en.cppreference.com/w/cpp/string/byte/memcpy`
[5] `https://en.cppreference.com/w/cpp/container/span`
[6] `https://en.cppreference.com/w/cpp/types/numeric_limits/max`

3e) Finden eines Schlüssels (1 Punkt)

Stellen Sie die statische Funktion `find_key` in der Klasse `Key` bereit. Diese akzeptiert zwei Parameter: Den ersten vom Typ `std::span` mit konstanten `key_type` und den zweiten vom Typ `std::uint64_t`. Geben Sie einen Schlüssel zurück, dessen berechneter Hashwert gleich dem zweiten Argument ist (sollte es mehrere davon geben, können Sie einen beliebigen davon zurück geben).
Sie dürfen annehmen, dass ein solcher immer akzeptiert.
Ihr Compiler benötigt ggf. die Rückgabe eines Dummywertes, wenn kein passender Schlüssel gefunden wird (weil ihr Compiler nicht obige Zusicherung hat). Sie können in diesem Fall einen beliebigen Wert zurück geben (da dies nie passieren sollte).

3f) Parallelisierung der Suche (2 Punkte)

Stellen Sie die statische Funktion `find_key_parallel` in der Klasse `Key` bereit, welche funktional `find_key` gleicht, aber einen dritten Parameter vom Typ `int` akzeptiert. Dieser soll den Standardwert 1 haben und gibt an wie viele Threads gleichzeitig den minimalen Wert suchen sollen. Sie dürfen annehmen, dass dieser Wert immer zwischen 1 und 96 ist. Parallelisieren Sie die Suche mit OpenMP und benutzen Sie in dem Konstrukt die übergebene Anzahl Threads.

Nachname, Vorname: _____ Matrikelnummer: ␣␣␣␣␣␣␣␣

Hinweise zur Abgabe

Hier ein paar zusätzliche Formalitäten zu Ihrer Abgabe. Falls Sie diese nicht beachten, ist es möglich, dass Sie keine Punkte für eine oder alle Aufgaben erhalten.

Angabe der Autorschaft

Geben Sie in der Datei /source/authors.h an, wer von Ihnen an welcher Teilaufgabe mitgearbeitet hat. Mehrere Namen pro Teilaufgabe sind ok, trennen Sie diese z.B. mit Komma. Wir fordern eine Teilnahme von allen! Sollten Sie nicht an dem Praktikum mitgearbeitet haben (basierend auf den Angaben), erhalten Sie 0 Punkte auf das ganze Praktikum.

Hochladen des Quellcodes

Zippen Sie folgende Dateien und laden Sie das zip-Archiv in Moodle hoch:

- Den Ordner /cmake/
- Den Ordner /source/
- Die Datei CMakeLists.txt

Sie sind alle für die korrekte Abgabe verantwortlich. Zu spät eingereichte Dateien, Dateien mit fehlenden Lösungen, etc., liegen in allein Ihrer Verantwortung.

Hinweise zum Lichtenberg

Der Lichtenberg-Hochleistungsrechner ist aufgeteilt in sogenannte Loginknoten und Rechenknoten, wobei sich alle das gleiche Dateisystem teilen. Erstere sind mit **ssh** und **scp** zu erreichen und werden benutzt um Rechenaufgaben für letztere zu kreieren. Wenn Sie eine sinnvolle Linux-Distribution verwenden, kann /bin/bash nativ **ssh** und Sie können das Dateisystem des Lichtenberg direkt in Ihrem Explorer (z.B. Nemo) einbinden. Wenn Sie Windows benutzen, empfiehlt sich das Terminal bzw. PuTTy für **ssh** und WinSCP für **scp**.

Sobald Sie eingeloggt sind, sollten Sie zusätzliche Softwarepakete laden. Das erreichen Sie beispielsweise mit **module load git/2.40.0 cmake/3.26.1 gcc/11.2.0 cuda/11.8 boost/1.81.0**. Kopieren Sie die Dateien auf den Hochleistungsrechner und kompilieren Sie dort das Programm.

Wenn Sie eine Rechenaufgabe lösen möchten, müssen Sie dafür SLURM benutzen. Effektiv führen Sie den Befehl **sbatch <filename>** aus, wobei <filename> hier eine Datei ist, die, sobald Ihr Programm Zeit und Resourcen zugeteilt bekommen hat, auf den Rechenknoten ausgeführt wird. Ein beispielhaftes Skript sieht so aus:

```
#!/bin/bash

#SBATCH -J parprog
#SBATCH -e /home/kurse/kurs00072/<TUID>/stderr/stderr.parprog.%j.txt
#SBATCH -o /home/kurse/kurs00072/<TUID>/stdout/stdout.parprog.%j.txt
#SBATCH -C avx512
#SBATCH -n 1
#SBATCH --mem-per-cpu=1024
#SBATCH --time=5
#SBATCH --cpus-per-task=1
#SBATCH -A kurs00072
#SBATCH -p kurs00072
#SBATCH --reservation=kurs00072

module load git/2.40.0 cmake/3.26.1 gcc/11.2.0 cuda/11.8 boost/1.81.0

echo "This is job $SLURM_JOB_ID"
```

Der Kurs ist ab dem 01. November 2023 freigeschaltet. Alle Informationen erhalten Sie auch hier: `https://www.hrz.tu-darmstadt.de/hlr/nutzung_hlr/zugang_hlr/loginknoten_hlr/index.de.jsp`