Praktikum 0 zu Parallele Programmierung – Die Probe

Lukas Rothenberger

TECHNISCHE UNIVERSITÄT DARMSTADT

Wintersemester 2024/2025
15.10.2024

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. Diese werden nicht bewertet.*
- Stellen Sie sicher, dass Ihre Abgabe auf dem Hochleistungsrechner der TU Darmstadt lauffähig ist, andernfalls ist mit Punktabzug zu rechnen.

Bepunktung der Aufgaben

- Bei jeder Teilaufgabe stehen die erreichbaren Punkte dabei, welche die Anzahl an funktionalen Tests für die jeweige Teilaufgabe angibt.
- Für jeden funktionalen Test 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.

Zusätzliche Hinweise

- Nennen sie **keine** Ihrer Dinge (z.B. Funktionen, Dateien, Klassen) mit `ppws24` als Bestandteil vom Namen.
- Ihnen steht frei weitere Funktionalität zu implementieren, solange nicht explizit anders geregelt.
- Achen Sie darauf, dass Sie nur standardkonformes C++ 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.
- Beachten Sie, dass die Tutor:innen deren Sprechstunden Sie besuchen nicht notwendigerweise Ihre Abgaben bewerten.

Achtung: Dies ist nur eine Probe, die Punkte erhalten Sie nicht wirklich.

Nachname, Vorname: _____ Matrikelnummer: ⌴⌴⌴⌴⌴⌴⌴

Aufgabe 1: Einrichten der Umgebung

1a) Installieren der Software

Dieses und die folgenden Praktika benötigen Softwarepakete, mit denen Sie gegebenenfalls noch nicht gearbeitet haben: CMake, Make, git, einen modernen C/C++-Kompiler, später CUDA.
Auf dem Lichtenberg-Hochleistungsrechner erhalten Sie diese mit: `module load git cmake gcc/11`.
Falls Sie eine sinnvolle Linuxdistribution verwenden, können Sie diese installieren mit: `sudo apt install git cmake binutils g++-11 gcc-11`.
Falls Sie Windows verwenden, empfehlen wir Visual Studio 2022, ggf. brauchen Sie CMake und git extra.

1b) Einrichten der Entwicklungsumgebung

Laden Sie das zip-Archiv mit dem Quellcode runter und entpacken Sie dieses an einem geeigneten Ort.
Wenn Sie eine sinnvolle Linuxdistribution verwenden (wie auf dem Lichtenberg), navigieren Sie im Terminal in den entpackten Ordner und führen aus: `mkdir build && cd build && cmake .. && make -j`. Damit erstellen Sie den Ordner, in dem das Programm gebaut wird, konfigurieren das Projekt (ggf. wiederholen, s.u.) und kompilieren Ihren Quellcode (ggf. wiederholen, s.u.).
Wenn Sie Windows benutzen, erstellen Sie ein neues VS-Projekt (z.B. über die CMake-GUI) und wählen als Ordner, in dem die Binärdateien kompiliert werden, auch `build` o.ä. aus. In der CMake-GUI klicken Sie nacheinander auf Konfigurieren, Generieren, Projekt öffnen.

1c) Struktur von CMake

Die CMake-Dateien sind für die Struktur Ihres Projektes zuständig. Darin sind mehrere Projekte definiert:

- `lab_lib`: Darin entwickeln Sie das ganze eigentliche Projekt. Die entsprechende CMake-Datei ist /source/CMakeLists.txt. Falls Sie weitere .cpp-Dateien zum Projekt hinzufügen möchten, müssen Sie diese dort in die Liste eintragen.
- `lab`: Das Projekt beinhaltet nur den Haupteinsprungspunkt des Programms (die main-Funktion). Darin verarbeiten Sie die Kommandozeilenargumente und rufen die Funktionen Ihres Programms auf.
- `lab_benchmarks`: Das Projekt verwendet das Framework von Google für micro-benchmarking. Die entsprechende CMake-Datei ist /benchmark/CMakeLists.txt.
- `lab_test`: Das Projekt verwendet das Framework von Google für Unittests. Die entsprechende CMake-Datei ist /test/CMakeLists.txt
- `/cmake/...`: Der Ordner beinhaltet mehrere Skriptdateien, die Sie eigentlich nicht ändern müssen. Falls Sie Visual Studio benutzen und dort Ihre .h-Dateien nicht angezeigt werden, fügen Sie diese in /cmake/VisualStudio.cmake hinzu.

Aufgabe 2: Programmieraufgabe (Gesamt: 3 Punkte)

Ziel dieser Aufgabe wird es sein, einen eindimensionalen Zellulären Automaten[1] ohne topologische Verknüpfung der Randpunkte mit der Evolutionsregel 82[2] zu implementieren.

2a) Autoren

Tragen Sie in der Datei /source/authors.h zu jeder Teilaufgabe ein, wer sie bearbeitet hat!

2b) Programmierung des internen Zustandes (1 Punkt)

Implementieren Sie die Funktionen `AutomatonState::set_state`, `AutomatonState::get_state` und `AutomatonState::print` in `source/util/AutomatonState.cpp`. Dabei soll gelten:

- `AutomatonState::set_state` überschreibt den aktuellen Wert von `internal_state`,
- `AutomatonState::get_state` gibt aktuellen Wert von `internal_state` zurück,
- `AutomatonState::print` gibt den aktuellen Wert von `std::vector<bool> internal_state` auf der Komandozeile aus. Die Ausgabe soll in genau eine Zeile erfolgen. Die in `internal_state` abgelegten Werte sollen von vorne nach hinten ausgegeben werden. Für die Darstellung der Werte soll folgendes gelten: `true` wird als #, `false` als einfache Leerstelle dargestellt.

2c) Vorbereitung des zellulären Automaten (1 Punkt)

Implementieren Sie zunächst den Konstruktor der Klasse `CellularAutomaton` in der Datei `source/util/CellularAutomaton.cpp`. Dieser soll:

- den Wert des internen `automaton_state` mit den `initial_state_values` überschreiben,
- den Wert von `simulation_iterations` mit `iterations` überschreiben,
- den Wert von `state_size` setzen, wobei diese der Länge des `std::vector<bool> initial_state` entsprechen soll.

Sie können gegebenenfalls `https://en.cppreference.com/w/cpp/container/vector` zu Hilfe nehmen.

Weiterhin ist die Funktion `get_new_value_from_pattern` in der Datei `source/util/Mapping.cpp` zu implementieren. Diese Bestimmt aus dem aktuellen Zustand einer Zelle und den Zuständen der Nachbarzellen den neuen Zustand der Zelle. Die Funktion soll das folgende Mapping abbilden:

left	1	1	1	1	0	0	0	0
center	1	1	0	0	1	1	0	0
right	1	0	1	0	1	0	1	0
Rückgabe	0	1	0	1	0	0	1	0

[1] `https://en.wikipedia.org/wiki/Cellular_automaton`
[2] `https://plato.stanford.edu/entries/cellular-automata/supplement.html`

2d) Ausführung des zellulären Automaten (1 Punkt)

Implementieren Sie die Funktion `simulate` der Klasse `CellularAutomaton`. Diese soll:

- den ursprünglichen Zustand des Systems auf der Kommandozeile ausgeben, falls `enable_printing` aktiviert ist,
- `simulation_iterations` Simulationsschritte ausführen,
- in jedem Simulationsschritt:
  - den neuen Wert einer Zelle mittels `get_new_value_from_pattern` bestimmen, wobei die Zustände der Randzellen unverändert übernommen werden
  - den internen `automaton_state` des zellulären Automaten updaten
  - den neuen Zustand auf der Kommandozeile anzeigen, falls `enable_printing` aktiviert ist
- den finalen Zustand des Systems zurückgeben.

Aufgabe 3: Weitere Tests und Benchmarks

Schreiben Sie sich weitere Tests und Benchmarks.

Aufgabe 4: Hochladen des Quellcodes

Zippen Sie:

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

In einem echten Praktikum würden Sie diese in Moodle hochladen müssen.

Aufgabe 5: Überprüfung der Aufgaben

Laden Sie das zip-Archiv mit der Bewertung runter. Kopieren (und ersetzen) Sie die Ordner /benchmark/ und /test/. Kompilieren Sie neu (auch CMake neu ausführen). Führen Sie /build/bin/.../lab_benchmarks und /build/bin/.../lab_test aus.