Praktikum 1 zu Parallele Programmierung

Lukas Rothenberger

TECHNISCHE UNIVERSITÄT DARMSTADT

Wintersemester 2025/2026
04.11.2025

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 funktionale Test selbst, oder der zugehörige öffentliche Test auf kompilierbarkeit des Programmes 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.
- Sie erhalten keine Punkte für Teilaufgaben mit fehlschlagenden öffentlichen Compiletests! Sie sind dafür Verantwortlich, einen kompilierenden, mit den Vorgaben kompatiblen Code einzureichen. Verändern Sie die vorgebenen Signaturen nicht!

Zusätzliche Hinweise

- Nennen sie **keine** Ihrer Dinge (z.B. Funktionen, Dateien, Klassen) mit `ppws25` 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.

Allgemeine Hinweise

Wenn eine Methode einen Parameter nicht ändert, sollten Sie den als `const` klassifizieren – dies vermeidet Programmierfehler.[1]
Wenn eine Methode das aktuelle Objekt nicht ändert, sollten Sie diese als `const` klassifizieren. Dadurch sind die Methoden aufrufbar, auch wenn das Objekt als solches selbst `const` ist (siehe vorherigen Hinweis).[2]
Wenn eine Methode keine Exception werfen kann, sollten Sie diese als `noexcept` markieren.[3]
Wenn eine Methode einen Wert zurück gibt und ein logischer Fehler wäre, diesen nicht zu speichern, können Sie diese mit `[[nodiscard]]` markieren. Wird der Wert nicht beim Aufruf gespeichert, erzeugt dies eine Warnung.[4]

[1] `https://en.cppreference.com/w/cpp/language/cv`
[2] `https://en.cppreference.com/w/cpp/language/member_functions`
[3] `https://en.cppreference.com/w/cpp/language/noexcept_spec`
[4] `https://en.cppreference.com/w/cpp/language/attributes/nodiscard`

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

Aufgabe 1: Einleitung

Das Ziel des ersten Praktikums wird es sein, ein Framework zur Darstellung und Verwaltung gerichteter Graphen zu implementieren, welches schlussendlich eine Erreichbarkeitsanalyse ermöglichen soll. Das Praktikum ist dabei wie folgt Strukturiert:

- Aufgabe 2: Implementierung der Knoten-Datenstruktur
- Aufgabe 3: Implementierung des gerichteten Graphen
- Aufgabe 4: Exportieren von Graphen
- Aufgabe 5 (Optional): Importieren von Graphen
- Aufgabe 6: Implementieren der Datenstruktur Matrix
- Aufgabe 7: Adjazenzmatrizen und Konvertierungen
- Aufgabe 8: Erreichbarkeitsanalyse

Da die einzelnen Aufgaben teilweise aufeinander aufbauend sind, empfehlen wir Ihnen die Reihenfolge bei der Bearbeitung einzuhalten. Verändern Sie die vorgegebenen bzw. gefragten Signaturen nicht, da diese automatisiert getestet werden. Sie können eigene Funktionen und Klassen zur Implementierung der geforderten Funktionalität definieren und nutzen.

Aufgabe 2: Node (Gesamt: 2 Punkte)

Sie finden ein vorbereitetes Code-Skelett zur Implementierung der Datenstruktur *Node* in `source/Node.h` bzw. `source/Node.cpp`.

2a) Constructor and get_id (1 Punkt)

Implementieren Sie den Konstruktor der Klasse `Node`. Stellen Sie sicher, dass die private `std::int32_t id` nach dem Aufruf den korrekten Wert beinhaltet. Implementieren Sie zudem die Methode `std::int32_t get_id()`, welche die `id` eines Knoten zurückgeben soll.

2b) Edge management (1 Punkt)

Implementieren Sie die Methoden `add_in_edge`, `add_out_edge`, `get_in_edges`, und `get_out_edges`, wobei folgendes gelten soll:

- `void add_in_edge(Node* source)` soll einen Eintrag für `source` im `std::vector<Node*> in_edges` anlegen, um eingehende Kanten sichern zu können.
- `void add_out_edge(Node* target)` soll einen Eintrag für `target` im `std::vector<Node*> out_edges` anlegen, um ausgehende Kanten darstellen zu können.
- `std::vector<Node*> get_in_edges()` soll die Ursprungsknoten eingehender Kanten als `std::vector<Node*>` zurückgeben.
- `std::vector<Node*> get_out_edges()` soll die Zielknoten ausgehender Kanten als `std::vector<Node*>` zurückgeben.

Aufgabe 3: Directed Graph (Gesamt: 4 Punkte)

Sie finden ein vorbereitetes Code-Skelett zur Implementierung der Datenstruktur *DiGraph* zur Darstellung eines gerichteten Graphen in `source/DiGraph.h` bzw. `source/DiGraph.cpp`.

3a) Registering and retrieving nodes (2 Punkte)

Implementieren Sie zunächst den Konstruktor der Klasse `DiGraph` so, dass die private `std::int32_t next_free_node_id` mit `0` initialisiert wird. Dieser Wert soll stets eine bislang nicht verwendete und somit ohne Konflikte verwendbare Knoten-ID beinhalten. Sie müssen Knoten-IDs nicht wiederverwenden, inkrementieren ist also ausreichend.

Implementieren Sie die Methode `Node* add_node()`, welche einen Knoten vom Typ `Node` unter Verwendung der nächsten, unbenutzten Knoten-ID erstellen soll. Weiterhin soll ein Eintrag in der `std::unordered_map<int32_t, Node*> node_map` angelegt werden, welche eine Zuordnung von Knoten-IDs zu `Node*` beinhalten soll. Der `pointer` zum angelegten Knoten soll zurückgegeben werden. Stellen Sie sicher, dass `next_free_node_id` wie vorgegeben angepasst wird.

Implementieren Sie die Methode `Node* add_node_carelessly(std::int32_t new_node_id)`, welche einen neuen Knoten unter Verwendung von `new_node_id` anlegt, einen Eintrag in `node_map` erstellt, und den `pointer` zum neuen Knoten zurückgibt. `next_free_node_id` muss nicht verändert werden, Sie können annehmen, dass der Aufrufer für das Sicherstellen der Konsistenz verantwortlich ist.

Implementieren Sie die Methode `std::int32_t get_node_count()`, welche die Anzahl der aktuell im Graphen registrierten Knoten zurückgibt.

Implementieren Sie die Methode `Node* get_node_by_id(std::int32_t node_id)`, welche einen `pointer` zum Knoten mit der Knoten-ID `node_id` zurückgibt. Werfen Sie einen Fehler vom Typ `std::invalid_argument`, falls kein entsprechender Knoten im Graphen enthalten ist.

Implementieren Sie die Methode `void set_next_free_node_id(std::int32_t value)`, welche den aktuellen Wert von `next_free_node_id` mit `value` überschreibt.

Implementieren Sie den Destruktor der Klasse `DiGraph`. Stellen Sie insbesondere sicher, dass der gesamte dynamisch allokierte Speicher korrekt freigegeben wird.

3b) Registering edges (2 Punkte)

Implementieren Sie die Methode `void add_edge(std::int32_t source_id, std::int32_t target_id)`. Die Methode soll eine Kante im Graphen anlegen, welche vom Knoten mit der Knoten-ID `source_id` ausgeht und im Knoten mit der Knoten-ID `target_id` endet. Stellen Sie sicher, dass ein Fehler vom Typ `std::invalid_argument` geworfen wird, falls mindestens eine der beiden Knoten-IDs nicht im Graphen enthalten ist.

Aufgabe 4: Exporting the graph (Gesamt: 1 Punkt)

Um Visualisierungen der mittels `DiGraph` definierten Graphen erzeugen zu können, soll in dieser Aufgabe ein Export der Graphen in das von Graphviz unterstützte DOT-Format [5], bzw. ein im folgenden definiertes Subset davon ermöglicht werden.

4a) Export the graph into a .dot-file (1 Punkt)

Implementieren Sie die Methode `void export_to_dot_file(std::string file_path)` der Klasse `DiGraph` in `source/DiGraph.cpp`. Die Methode soll den `DiGraph` im folgenden Format in einer Datei ablegen, welche durch `file_path` mittels Dateipfad angegeben ist. Sie können dabei davon ausgehen, dass die Datei zuvor nicht existiert, bzw. eine existierenden ohne Bestätigung überschreiben. Verwenden Sie das folgende Format:

```
digraph {
<tab><source_id_0> -> <target_id_0>; // edge 0
...
<tab><source_id_n> -> <target_id_n>; // edge n
}
```

wobei `<tab>` den tab-Charakter repräsentiert und z.B. `<source_id_0>` durch die Knoten-ID des Ursprungsknoten der Kante ersetzt werden muss. Gleiches gilt für z.B. `<target_id_0>` als Beispiel für die Zielknoten. Sie finden ein Beispiel für das zu verwendende Format in `input/simple.dot`. Sie können annehmen, dass nur solche Knoten relevant sind, welche mindestens eine eingehende bzw. ausgehende Kante besitzen, und somit durch das beschriebende Format beschreibbar sind. Knoten, welche mit keiner Kante verbunden sind, müssen nicht explizit aufgeführt werden.

4b) Create visualization from .dot file (Optional)

Um Visualisierungen aus den erstellten Dateien im DOT-Format zu erstellen können Sie `dot`, bereitgestellt von Graphviz, verwenden. Graphviz ist auf den Loginknoten des Lichtenberg-Clusters für Sie verfügbar. Um Beispielsweise aus einer Datei im DOT-Format (`input.dot`) eine SVG-Visualisierung (`output.svg`) zu erstellen, können Sie das folgende Kommando verwenden:

```
dot -Tsvg -o output.svg input.dot
```

Ein Beispiel für eine solche Visualisierung finden Sie vorbereitet im Ordner `input`.

Aufgabe 5: Importing a graph

Wir haben Ihnen in `source/import.cpp` die Funktion `DiGraph get_graph_from_file(const std::string path)` bereitgestellt, welche einen Graphen im beschriebenen DOT-Format aus der Datei im Dateipfad `path` einliest und einen entsprechenden `DiGraph` erstellt. Sie können diese Funktion z.B. zu Testzwecken, aber auch zur Implementierung weiterer Funktionalität in `source/main.cpp` verwenden.

[5] `https://graphviz.org/doc/info/lang.html`

Aufgabe 6: Matrix (Gesamt: 8 Punkte)

Ziel dieser Aufgabe wird es sein, eine Datenstruktur zur Repräsentation von zweidimensionalen Matrizen beliebiger Daten zu implementieren. Sie finden ein vorbereitetes Code-Skelett zur Implementierung der Datenstruktur *Matrix* in `source/Matrix.h`. Da es sich um einen Template-Typ handelt, wird sämtliche Funktionalität der Klasse in der Headerdatei implementiert.

6a) Constructor and getters for dimensions (1 Punkt)

Implementieren Sie den Konstruktor der Klasse `Matrix`, welcher die beiden Argumente `std::int32_t rows` und `std::int32_t columns` entgegennehmen soll. `rows` soll dabei die Anzahl der Zeilen der erstellten Matrix angeben, wobei `columns` die Anzahl der Spalten der Matrix angeben soll. Elemente der Matrix sollen vom Template-Typ `T` sein. Der Konstruktor soll einen Speicherbereich allokieren, welcher ausreichend groß für die geforderte Anzahl an Elementen der Matrix ist, und die Elemente mit `0` initialisieren. Der Anfang des Speicherbereichs soll mittels der privaten Variablen `contents` Zugreifbar sein. Weiterhin sollen die Dimensionen der Matrix in den entsprechenden privaten Variablen `rows` bzw. `columns` festgehalten werden.

Implementieren Sie zudem die Methoden `std::int32_t get_rows()` bzw. `std::int32_t get_columns()`, welche die zuvor gesicherten Dimensionen der Matrix zurückgeben sollen.

6b) Destructor (1 Punkt)

Implementieren Sie den Destruktor der Klasse `Matrix`. Stellen Sie insbesondere sicher, dass der gesamte, dynamisch allokierte Speicher freigegeben wird.

6c) get_content and set_content (2 Punkte)

Implementieren Sie die Methoden `T get_content(std::int32_t row, std::int32_t column)` und `void set_content(std::int32_t row, std::int32_t column, T value)`, wobei `get_content` zum Auslesen, und `set_content` zum Schreiben von Werten der Matrix verwendet werden sollen. In beiden Fällen soll `row` die Zeile des zugegriffenen Wertes angeben, während `column` die Spalte des Wertes benennt. Die Indexierung der Zeilen und Spalten soll jeweils bei `0` beginnen. Ein Beispiel zur Indexierung der Zellen finden Sie in der folgenden Abbildung, wobei $\langle R_0, C_n \rangle$ beispielsweise die Reihe 0 und Spalte n beschreibt.

$$\begin{pmatrix} \langle R_0, C_0 \rangle & \langle R_0, C1 \rangle & \langle R_0, C2 \rangle \\ \langle R_1, C_0 \rangle & \langle R_1, C1 \rangle & \langle R_1, C2 \rangle \\ \langle R_2, C_0 \rangle & \langle R_2, C1 \rangle & \langle R_2, C2 \rangle \end{pmatrix}$$

Stellen Sie bei Ihrer Implementierung sicher, dass das folgende Speicherlayout (Row-major ordering) verwendet wird:

Memory	0x0	$\cdots$						$\cdots$	0xFFFFFFFF
Contents			$\langle R_0, C_0 \rangle$	$\langle R_0, C_1 \rangle$	$\langle R_0, C_2 \rangle$	$\langle R_1, C_0 \rangle$	$\langle R_1, C_1 \rangle$	$\cdots$	
Pointers			contents						

Stellen Sie zudem sicher, dass Exceptions vom Typ `std::invalid_argument` geworfen werden, falls ungültige Werte für `row` bzw. `column` übergeben wurden.

6d) get_row and get_column (1 Punkt)

Implementieren Sie die Methode `std::vector<T> get_row(std::int32_t row)`, welche die Elemente der Matrix in Zeile `row` als `std::vector` zurückgeben soll. Die im Vector enthaltenen Elemente der ausgewählten Zeile sollen in aufsteigender Reihenfolge ihrer Spaltenindizes abgelegt werden.

Implementieren Sie die Methode `std::vector<T> get_column(std::int32_t column)`, welche die Elemente der Matrix in Spalte `column` als `std::vector` zurückgeben soll. Die im Vector enthaltenen Elemente der ausgewählten Spalte sollen in aufsteigender Reihenfolge ihrer Zeilenindizes abgelegt werden.

Stellen Sie zudem sicher, dass Exceptions vom Typ `std::invalid_argument` geworfen werden, falls ungültige Werte für `row` bzw. `column` übergeben wurden.

6e) operator + (1 Punkt)

Implementieren Sie den **+**-Operator der Klasse `Matrix`. Dieser soll sich wie die aus der Mathematik bekannte Matrizenaddition verhalten. Stellen Sie sicher, dass eine Exception vom Type `std::invalid_argument` geworfen wird, falls die Dimensionen der an der Operation beteiligten Matrizen nicht kompatibel sind. Der Operator darf die Operanden nicht verändern. Nehmen Sie an, dass es sich bei der als Argument übergebenen Matrix im Fall einer Infixdarstellung um den rechten Operand handelt. Sie können weiterhin annehmen, dass sowohl der **+**, als auch der*****-Operator für T definiert ist.

6f) operator * (1 Punkt)

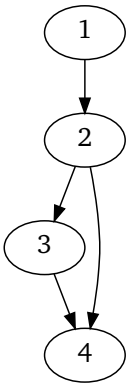
Implementieren Sie den *****-Operator der Klasse `Matrix`. Dieser soll sich wie die aus der Mathematik bekannte Matrizenmultiplikation verhalten. Stellen Sie sicher, dass eine Exception vom Type `std::invalid_exception` geworfen wird, falls die Dimensionen der an der Operation beteiligten Matrizen nicht kompatibel sind. Der Operator darf die Operanden nicht verändern. Nehmen Sie an, dass es sich bei der als Argument übergebenen Matrix im Fall einer Infixdarstellung um den rechten Operand handelt. Sie können weiterhin annehmen, dass sowohl der **+**, als auch der *****-Operator für T definiert ist.

6g) Supporting template type (1 Punkt)

Stellen Sie sicher, dass Ihre Implementierung den Template-Typ T unterstützt. Sie können annehmen, dass es sich bei T insbesondere um numerische und boolesche Werte handeln wird.

Aufgabe 7: Conversions (Gesamt: 4 Punkte)

Um Knoten und Kanten eines Graphen darzustellen können Adjazenzmatrizen verwendet werden. Adjazenzmatrizen sind quadratische Matrizen, bei denen jeder Knoten des Graphen durch eine Zeile bzw. Spalte repräsentiert wird. Einträge in der Matrix stellen dabei das Verhältnis zwischen dem betrachteten Ursprungs- und Zielknoten der Kanten dar, wobei typischerweise der Wert `0` repräsentiert, dass keine Verbindung vorliegt, und sonstige Werte als Kantengewichte betrachtet werden können. Da im Fall dieses Praktikums keine Kantengewichte, sondern lediglich die Eigenschaft einer existierenden Kante zwischen zwei Knoten betrachtet wird, sind in unserem Fall die Werte `0` und `1` ausreichend. Aus diesem Grund werden wir im folgenden Adjazenzmatrizen als `Matrix<bool>` modellieren. In unserem Fall wird die Zeile der Matrix den Ursprung der Kante, und die Spalte der Matrix das Ziel der Kante repräsentieren. Vereinfachend werden wir weiterhin annehmen, dass der jeweilige Index der Zeile bzw. Spalte der jeweiligen Knoten-ID entspricht. Ein Beispiel für einen gerichteten Graphen und dessen Repräsentation als Adjazenzmatrix finden Sie in den folgenden Abbildungen. Beachten Sie, dass kein Knoten mit der Knoten-ID `0` existiert, Zeile und Spalte `0` aufgrund unserer Vereinfachung aber dennoch für diesen reserviert sind.



	0	1	2	3	4
0	0	0	0	0	0
1	0	0	1	0	0
2	0	0	0	1	1
3	0	0	0	0	1
4	0	0	0	0	0

Adjazenzmatrix

7a) Convert graph into adjacency matrix (2 Punkte)

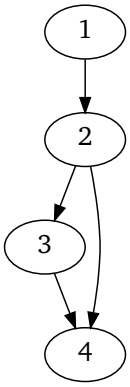
Implementieren Sie die Methode `Matrix<bool> get_adjacency_matrix_from_graph()` in `source/DiGraph.cpp`, welche einen `DiGraph` in eine äquivalente Adjazenzmatrix gemäß der vorherigen Spezifikation konvertiert.

7b) Convert adjacency matrix into graph (2 Punkte)

Implementieren Sie die Funktion
`DiGraph get_graph_from_adjacency_matrix(const Matrix<bool>& adjacency_matrix)` in `source/utils.cpp`, welche die als Referenz übergebene Adjazenzmatrix `adjacency_matrix` in einen äquivalenten `DiGraph` konvertiert.

Aufgabe 8: Reachability (Gesamt: 1 Punkt)

Erreichbarkeitsmatrizen stellen die Erreichbarkeit zwischen Knoten in einem Graphen dar. Ihr Format ist dabei äquivalent zu den Adjazenzmatrizen, jeder Knoten im Graph wird also einer Zeile bzw. Spalte zugeordnet, wobei ein Eintrag ungleich `0` anzeigt, dass ein Pfad vom Ursprungsknoten zum Zielknoten existiert. Im Rahmen dieses Praktikums wählen wir das gleiche Format wie zuvor bereits für die Adjazenzmatrizen beschrieben. Zeilen repräsentieren also den Ursprungsknoten, wohingegen Spalten den Zielknoten repräsentieren. Auch im Fall der Erreichbarkeitsmatrizen soll angenommen werden, dass der Zeilen- bzw. Spaltenindex der jeweiligen Knoten-ID entspricht. Per Konvention ist jeder Knoten trivial von sich selbst aus erreichbar. Ein Beispiel für einen gerichteten Graphen samt zugehöriger Adjazenz- und Erreichbarkeitsmatrix finden Sie im Folgenden.



	0	1	2	3	4
0	0	0	0	0	0
1	0	0	1	0	0
2	0	0	0	1	1
3	0	0	0	0	1
4	0	0	0	0	0

Adjazenzmatrix

	0	1	2	3	4
0	1	0	0	0	0
1	0	1	1	1	1
2	0	0	1	1	1
3	0	0	0	1	1
4	0	0	0	0	1

Erreichbarkeitsmatrix

8a) Calculating reachability matrices (1 Punkt)

Implementieren Sie die Funktion `Matrix<bool> get_reachability_matrix(const DiGraph& graph)` in `source/utils.cpp`, welche die Erreichbarkeitsmatrix des als Referenz übergebenen `DiGraph graph` berechnet. Sie können eine $N \times N$-Erreichbarkeitsmatrix (R) wie folgt aus einer $N \times N$-Adjazenzmatrix (A) berechnen:

$$R = A + A + \cdots + A^N$$

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.

Nameskonvention

Benennen Sie Ihre Abgabe bitte nach dem folgenden Schema: `Px_Gy.zip`, wobei `x` der Nummer des Praktikums und `y` Ihrer Gruppennummer entspricht.

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 allein in Ihrer Verantwortung.

Die Ausführbarkeit der Abgabe auf dem Lichtenberg-Hochleistungsrechner ist entscheidend, da dieses System zur Bewertung verwendet wird. Sollte Ihre Abgabe auf dem Lichtenberg-Hochleistungsrechner nicht kompilierbar und ausführbar sein, ist mit Punktabzug zu rechnen.

Compiletests

In jedem Praktikum werden Ihnen öffentliche Compiletest für Ihre Abgabe bereitgestellt. Diese werden verwendet, um die Kompilierbarkeit Ihrer Abgabe unabhängig vom Bearbeitungsstand zu überprüfen. Zudem wird in diesen Tests sichergestellt, dass beispielsweise Funktionssignaturen unverändert geblieben, und somit kompatibel mit unserem Testframework sind. Sie finden diese öffentlichen Tests im Verzeichnis `test_compilation`. Sie können die Compiletests via `mkdir build && cd build && cmake .. && make lab_test_compilation -j` ausführen. Sofern keine Fehler vorliegen, wird die ausführbare Datei /build/bin/lab_test_compilation erstellt, welche ebenfalls keine Fehler aufzeigen sollte.

Wichtig: Sie erhalten keine Punkte für Teilaufgaben mit fehlschlagenden öffentlichen Compiletests! Sie sind dafür Verantwortlich, einen kompilierenden, mit den Vorgaben kompatiblen Code einzureichen. Verändern Sie die vorgebenen Funktionssignaturen nicht!

Hinweis: Nutzen Sie die öffentlichen Tests zur Selbstkontrolle vor Einreichen Ihrer Abgabe.

Wichtig: Sie erhalten **keine** Punkte für das Bestehen der öffentlichen Compiletests.

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/2.47 cmake/3.30 gcc/13.3 cuda/11.8 boost/1.86**. 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 /work/scratch/kurse/kurs00093/<TUID>/stderr/stderr.parprog.%j.txt
#SBATCH -o /work/scratch/kurse/kurs00093/<TUID>/stdout/stdout.parprog.%j.txt
#SBATCH -C avx512
#SBATCH -n 1
#SBATCH --mem-per-cpu=1024
#SBATCH --time=5
#SBATCH --cpus-per-task=96
#SBATCH -c 96
#SBATCH -A kurs00093
#SBATCH -p kurs00093
#SBATCH --gres=gpu:v100
#SBATCH --reservation=kurs00093

module load git/2.47 cmake/3.30 gcc/13.3 cuda/11.8 boost/1.86

echo "This is job $SLURM_JOB_ID"
cd lab_1/build
bin/lab_test
```

Der Kurs ist ab dem 04. November 2025 freigeschaltet. Alle Informationen erhalten Sie auch hier: `https://www.hrz.tu-darmstadt.de/hlr/nutzung_hlr/zugang_hlr/loginknoten_hlr/index.de.jsp`

Sollten Sie große Ausgabedaten erzeugen, legen Sie diese bitte in `/work/scratch/kurse/kurs00093/TU-ID` ab, um Quotaüberschreitungen in `/work/kurse/kurs00093` zu vermeiden.
Wichtig: Dateien auf `/work/scratch/kurse/kurs00093/TU-ID` werden 8 Wochen nach der Erstellung gelöscht. Zudem wird von dem Verzeichnis kein Backup angelegt.

Für den Fall, dass Probleme oder Fragen Ihrerseits bezüglich der Nutzung des Lichtenberg-Hochleistungsrechners auftreten, möchten wir Sie auf das FAQ sowie die Website des HRZ (`https://www.hrz.tu-darmstadt.de/hlr/support_hlr/faq_hlr/index.en.jsp`) hinweisen. Selbstverständlich können Sie sich ebenfalls im Rahmen der angebotenen Sprechstunden an die Tutor:innen wenden.