Scientific Computing
Prof. Dr. Christian Bischof

TECHNISCHE UNIVERSITÄT DARMSTADT

spp-support@lists.sc.informatik.tu-darmstadt.de

Systemnahe und parallele Programmierung (WS 22/23)

Praktikum: C/C++ und OpenMP

Die Lösungen müssen bis zum 13. Dezember 2022, 16:00 Uhr, in Moodle submittiert werden. Anschließend müssen Sie ihre Lösung einem Tutor vorführen. Das Praktikum wird benotet. Im Folgenden einige allgemeine Bemerkungen und Anforderungen, die für alle Aufgaben auf diesem Blatt gelten:

- Es gibt jeweils ein vorgegebene CMake- und Makefiles welches das Programm kompiliert, das Sie entwickeln sollen.
- Dynamisch allokierter Speicher muss freigegeben werden, bevor das Programm endet.
- Aus der C++-Standardbibliothek wird der Containtertyp `std::vector` benutzt[1].
- Nutzen Sie einen geeigneten modernen Compiler auf dem Lichtenberg Cluster, zB., `module load gcc/10`.
- Die Programme müssen auf dem Lichtenberg Cluster kompilierbar und ausführbar sein. Alle Zeitmessungen müssen auf dem Lichtenberg Cluster compute nodes ausgeführt werden (mittels sbatch). Siehe Anleitung zur Nutzung des Clusters.
- Die jeweilig erstellten `sbatch`-Dateien fuer die Performanzmessungen müssen zur Abgabe beigelegt werden.
- Vorgegebener C/C++ Code in den Dateivorlagen darf prinzipiell nicht geändert werden und muss wie gegeben verwendet werden.
- Zeitmessungen sind bereits integriert.
- Alle Lösungen, die keinen Programmcode erfordern, bitte zusammen in einer PDF Datei einreichen.
- Laden Sie alle Dateien in einem zip-Archiv in Moodle hoch. Die Ordnernamen sollten Ihre Gruppennummer beinhalten.

[1] `https://en.cppreference.com/w/cpp/container/vector`

Aufgabe 1

Parallel daxpy (20 Punkte) Bei der `daxpy`-Operation berechnet man aus den **`double`**-Vektoren $\vec{y}$ und $\vec{x}$ der Länge n sowie dem Skalar a:

$$\vec{y} = \vec{y} + \vec{x} \cdot a \tag{1}$$

In dieser Aufgabe entwickeln Sie eine sequenzielle und eine parallele Version zur Berechnung von `daxpy`. Nutzen Sie zur Lösung die Dateivorlage `daxpy.c`. Das Programm wird mit einem Aufrufparameter $t \in \mathbb{N}$ gestartet, der angibt, wie viele Threads in parallelen Programmteilen genutzt werden.

a) (4 Punkte) Die Funktion `daxpy_ser` soll `daxpy` sequenziell berechnen.

b) (2 Punkte) Die Funktion `main` soll zu Beginn die Anzahl der genutzten OpenMP-Threads auf den Wert des durch die Eingabeparameter definierten Wert `nt` setzen.

c) (4 Punkte) Um den Erfolg der vorigen Aufgabe zu verifizieren, soll `main` anschließend auslesen, mit wie vielen Threads die OpenMP-Umgebung arbeitet und diesen Wert in einer einzigen Konsolenzeilenausgabe an den Nutzer zurückgeben.

d) (3 Punkte) `daxpy_par` `daxpy` parallel berechnen. Benutzen Sie beim Spawnen der parallelen Region die **`default`**`(none)`-und geben Sie die Speicherklauseln aller Variablen explizit an.

e) (4 Punkte) Messen Sie die sequenzielle Ausführungszeit `daxpy_ser` und die Zeit der parallelen Berechnung `daxpy_par` mit $1, 2, 4, 8$ und 16 Threads für die vorgegebene Vektorlänge 400.000.000. Stellen Sie die Ergebnisse grafisch oder tabellarisch dar.

f) (3 Punkte) Setzen Sie das Scheduling der parallelisierten **`for`**-Schleife auf `(`**`static`**`,3)`. Wie verhalten sich nun die Ausführungszeiten? Begründen Sie die Veränderung.

Aufgabe 2

Parallel Quicksort (16 Punkte) Quicksort[2] ist ein rekursiver Sortieralgorithmus, der im Durchschnitt $O(n \cdot \log n)$ Vergleiche durchführt um ein Array der Länge n zu sortieren.
Aus dem Array wird ein Eintrag als Pivotelement gewählt und das Array anschließend so aufgeteilt, dass, der linke Teil alle Elemente kleiner dem Pivotelement enthält, der rechte Teil die größeren. Nun ruft man für jedes Teil-Array Quicksort rekursiv auf. Die Rekursion endet, wenn die Teil Arrays Länge 0 erreichen. Aufgrund der Anwendung dieses "Teile und herrsche" Prinzips, lässt Quicksort sich mit dem OpenMP Task Konstrukt parallelisieren.
Eine Beispiel Implementierung ist in der Datei `quicksort.cpp` gegeben.

a) (8 Punkte) Parallelisieren Sie die Routine **`void`** `quickSort_par( ... )` mittels Tasking pragmas. Messen Sie die Laufzeit der Version ohne Pragmas **`void`** `quickSort_ser` und der parallelisierten Version mit 1, 2, 4 und 8 Threads. Wie entwickelt sich die Laufzeit?

Hinweis: Sie müssen für diese Aufgabe nur Code in den Funktionen `main` (unter dem Kommentar `Call the parallel quickSort implementation`) und `quickSort_par` hinzufügen.

Hinweis 2: Die `main`-Routine lässt die parallele Implementierung automatisch mit verschiedener Threadanzahl laufen und gibt die benötigte Laufzeit dafür an.

b) (8 Punkte) Wie erklären Sie das Laufzeitverhalten? Fügen Sie weiteren Code hinzu, der die Geschwindigkeit deutlich verbessert. Tragen Sie die Laufzeiten Ihrer verbesserten Version in die Tabelle aus Teil a) ein.

[2] https://de.wikipedia.org/wiki/Quicksort

Aufgabe 3

Parallel N-body Problem (39 Punkte)

Einführung: N-body Problem Das astronomische N-body Problem beschäftigt sich mit der Bestimmung von Positionen und Bewegungen von Körpern im Raum, auf die Gravitationskräfte von anderen Körpern wirken. Da jeder Körper jeden anderen beeinflusst, ergibt sich eine Komplexität von $O(n^2)$. Die Bestimmung der Position und Geschwindigkeit basiert auf den Newtonschen Gesetzen der Physik. Die Körper werden im folgenden als nicht verformbare Partikel mit Masse `m` angesehen. Ferner wird die Berechnung auf den 2D Fall beschränkt. Das bedeutet, jeder Partikel hat eine Position **`p`** `= (x, y)`, sowie eine Geschwindigkeit **`v`** `= (vx, vy)`.

Simulation Pseudocode Zunächst der generelle Ablauf dieser Simulation sieht folgend aus:

```
1  for(step = 0;step < num_steps;++steps) { /* for each timestep */
2      for (i = 0; i < num_bodies; ++i) { /* for each body i */
3          for (j = 0; j < num_bodies; ++j) { /* influence of body j on body i */
4              if(i != j)
5                  accel_i = accel_i + accel_routine(i, j); /* acceleration of i based on j */
6          }
7      }
8      /* Integration step (semi-implicit euler) */
9      for (i = 0; i < num_bodies; ++i) { /* for each body i update ... */
10         v[i]_new = v[i] + dt*accel_i; /* new velocity based on acceleration */
11         p[i]_new = p[i] + dt*v[i]_new; /* new position based on new velocity */
12     }
13 }
```

Hierbei ist `dt` ein Zeitschritt der Simulation und wird für den Integrationsschritt zum Berechnen der neuen Positionen und Geschwindigkeiten der Körper verwendet.

Berechnen der Beschleunigung. Das Berechnen der Beschleunigung eines Körpers, `accel_i`, wird im Folgenden erläutert. Grundsätzlich basiert die Berechnung auf dem Newtonschen Gravitationsgesetz, das die Kräfte zwischen zwei Körpern `i` und `j` beschreibt: $F = G\frac{m_i m_j}{r^2}$. Hierbei ist `m` die Masse eines Körpers und `r` der Abstand zwischen beiden. Da wir ein 2D Problem mit Vektorkomponenten `x, y` betrachten, ergibt sich folgende Formel: $F = G\frac{m_i m_j (p_j - p_i)}{r^3}$. Hierbei ist `p` die aktuelle Position des entsprechenden Körpers. Da über die Kraft die Beschleunigung mit der Beziehung $F = ma$ berechnet werden kann, ergibt sich folgende Formel: $\frac{F}{m_i} = a_i = G\frac{m_j (p_j - p_i)}{r^3}$:

```
1  accel_routine(i, j) { /* influence of j on acceleration of i */
2      G = 6.673e-11f; /* gravitational constant */
3      delta = p_j - p_i;
4      r = sqrt(delta_x^2 + delta_y^2 + eps); /* distance r of the bodies,
5                                                eps very small number
6                                                in case bodies completely overlap */
7      fac = 1.0 / r^3;
8      a = G * m_j * fac * delta; /* calculate acceleration based on formula */
9      return a;
10 }
```

Es wird zunächst die Beschleunigung des Körpers `i` basierend auf den Einflüssen aller anderen Körper berechnet. Hier gibt es den Fall, dass Körper eventuell überlappen und die Distanz `r = 0` ist. Durch die Einführung des Faktors `eps` kann man eine Division durch 0 verhindern. Im Anschluss wird diese Beschleunigung verwendet um die Geschwindigkeit und Position zu aktualisieren.

Behandlung von Kollisionen Als Erweiterung der Simulation wird der Fall behandelt, wenn sich zwei Körper sehr nahe kommen und kollidieren. Die Kollisionsbehandlung beinhaltet die Überprüfung des Abstands zweier Körper. Ist dieser unterschritten, werden beide Körper zu einem neuen vereint. Der neue Körper hat die Masse bestehend aus der Summer der beiden kollidierten Körper. Die Geschwindigkeit des neuen Körpers wird mit Hilfe des Impulserhaltungssatzes berechnet.

```
1 /* new Body when collision is detected */
2 m = m_i + m_j;
3 v = (m_i*v_i + m_j*v_j) / m;
4 p = p_i;
```

Das Vereinen der Körper macht es nun unnötig, `eps` für die Berechnung des Abstands zweier Körper zu verwenden. Der allgemeine Ablauf des Algorithmus wird nun erweitert. Vor der Berechnung der Beschleunigung eines Körpers wird nun eine Prüfung auf Kollisionen durchgeführt. Sollten zwei Körper `i` und `j` einen bestimmten Abstand unterschreiten, so werden sie fusioniert und ein neuer Körper mit dem oben beschriebenen Verfahren erstellt. *Beachten Sie*, eine Kollision von Körper `i` mit `j` ist das Gleiche wie eine Kollision von `j` mit `i`.

Kollisionen Vorgehen Eine Möglichkeit der Realisierung ist es, die Masse des Körpers `j` einfach auf 0 zu setzen und mit Hilfe der Formeln Körper `i` zu aktualisieren und als Kollisionsresultat anzusehen. Dieses Vorgehen macht es nötig allgemein zu prüfen, ob die Masse eines Körpers nicht 0 ist. Ansonsten könnten numerische Probleme auftreten bzw. nicht existente Körper die Berechnungen beeinflussen.
Die Parallelisierung von diesem Vorgehen erfordert es, bei dem Verändern der Körper auf mögliche *data races* zu achten. Hier kann man beispielsweise alle Kollisionen (parallel) erkennen und sammeln (Zwischenspeichern), und dann in einem Thread seriell nach dem vorgestellten Schema abarbeiten:

```
1 /* new body when collision is detected */
2 m = m_i + m_j;
3 v_i = (m_i*v_i + m_j*v_j) / m;
4 m_i = m;
5 m_j = 0;
```

Software Framework

Das bestehende Software Framework dient zur Hilfe der Umsetzung des oben beschriebenen Simulationssystems. Die Klasse `Simulation` ist die Hauptklasse, in der die prinzipielle N-body Simulation durchgeführt werden soll.
`Simulation` implementiert ferner das Interface `SimGUIAdapter`, welches die Schnittstelle zur bereitgestellten graphischen Umgebung darstellt. Die GUI soll ihnen, mit einer einfachen Visualisierung der Körper in ihrem System, helfen die Plausibilität ihrer Berechnungen zu überprüfen. Die GUI wurde mit dem Framework `Qt` umgesetzt. Da Sie davon nichts wissen müssen, wurde die Abstraktion `SimGUIAdapter.h` eingeführt. Ist die GUI aktiv, übernimmt diese die Kontrolle der Aufrufe für die Simulationsschritte.
Die Simulationsdaten sind in einfachen Textdateien abgelegt. Diese haben ein bestimmtes Format. Eine Datei wird mit Hilfe der Klasse `Config.h` eingelesen oder abgespeichert. Ferner bietet die Klasse das Erstellen von zufälligen Simulationsszenarien an.

Code Gesamtübersicht Folgend eine Übersicht des Codes mit kurzer Beschreibung.

OmpPractical Einstiegspunkt des Praktikums. An diesem Punkt werden die Objekte initialisiert und gegebenenfalls die GUI gestartet. Die GUI bekommt ein Simulations-Objekt.

Simulation Setzt den allgemeinen Ablauf der Simulation um, liest Dateien mit Hilfe von `Config.cpp` ein oder schreibt sie auf die Festplatte. Bietet ein Interface für die GUI. Hier müssen verschiedene Methoden vervollständigt/ergänzt werden.

Body Repräsentiert einen Körper mit Masse im 2D Raum. Hier müssen verschiedene Methoden vervollständigt/ergänzt werden.

Config Liest die Konfigurationsdateien für die Simulation ein. Kann Ergebnisse abspeichern für spätere Fortsetzung. Generiert Zufällige Szenarien.

PTime Dient zur präzisen Zeitnahme für Performanzmessungen.

SimGUIAdapter Interface der Simulation-Klasse um den momentanen Zustand des Systems zu visualiseren. Aufrufe durch die GUI, Visualisierung mit entsprechenden Callbacks.

gui/SimulationGUI Übernimmt die Visualisierung der Körper.

Am bestehenden Framework müssen nur `Simulation` und `Body` angepasst werden. Zusätzliche Klassen können Sie für das Erreichen der Lösung hinzufügen. Die Steuerung des Ablaufs geschieht jedoch über `Simulation`.

Datenformat für Simulationsdateien Jede Datei Beginnt mit 6 float Werten, gefolgt von den definierenden Werten eines jeden Körpers im System. Folgend ist der prinzipielle Aufbau dargestellt:

```
*Anzahl der Simulationsschritte*
*Anzahl der Körper im System*
*Zeitschrittweite dt*
*Wert für Koordinaten-Skalierung für die GUI*
*Distanz zwischen Körpern für Kollisionsbestimmung*
*x y v_x v_y Masse*
*x y v_x v_y Masse*
....
*x y v_x v_y Masse*
```

Bereitgestellte Simulationsdateien

Für das Praktikum sind drei Dateien mit bereitgestellten Werten zur Simulation mitgeliefert. Jede dieser Dateien wird gesondert betrachtet.

nbody.txt Berechnet die Bahnen der *inneren* Planeten unseres Sonnensystems. Alle Werte der Datei entsprechen den Dimensionen in der Realität. Die Positionen sind in `m`, Geschwindigkeiten in `m/s`, Masse in `kg` angegeben. Aufgrund der großen Dimensionen müssen die Zeitschritte entsprechend groß gewählt werden. Je kleiner der Wert, desto genauer ist die Berechnung. Hier ist er auf einen relativ großen Wert von `30e3` gesetzt worden. Abbildung 1 zeigt die Visualisierung des Systems bei Zeitschritt 206.

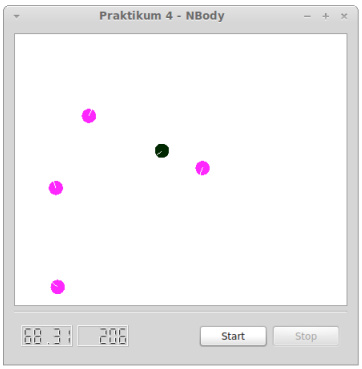

Figure 1: Innere Planeten. Simuliert mit dem beschriebenen N-body-Algorithmus.

collision.txt Dieses Szenario behandelt eine Kollsion von 2-4 Körpern (je nachdem, wie viele aus der Datei ausgelesen werden) und stellt einen guten Test für die Kollisionsbehandlung dar. Abbildung 2 zeigt vier Körper, die aufeinander zufliegen.

random.txt Das letzte Szenario ist zufällig generiert. Abbildung 3 zeigt den Zustand nach wenigen Zeitschritten.

Build Prozess mit CMake

CMake Generell Im root-Order des Projekts aufrufen, danach ist die executable im `bin/` Ordner.

```
module load gcc/10 llvm/12 // Optionally load Clang/LLVM compiler
cd *root-folder*
// ohne Qt oder OpenMP als debug build:
```

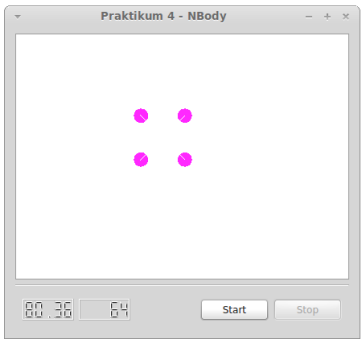

Figure 2: 4 Körper stoßen zeitgleich aufeinander.

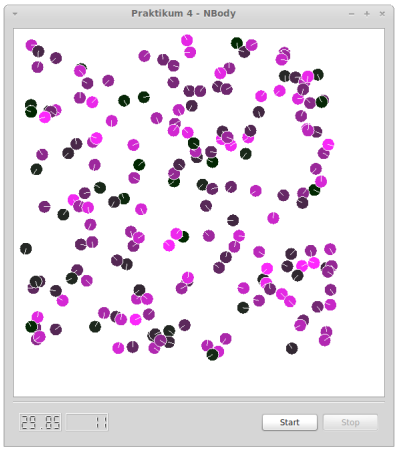

Figure 3: Szenario, das zufällig mit `Config` generiert wurde.

```
cmake -B build
cmake --build build --parallel 6
// ohne Qt oder OpenMP als release build:
cmake -B build -DCMAKE_BUILD_TYPE=Release
cmake --build build --parallel 6 --config Release
```

Qt & OpenMP Folgende flags können kombiniert werden:

```
// Um die GUI zu aktivieren (debug):
cmake -B build -DSPP_WITH_QT=ON
// Um OpenMP zu aktivieren (debug):
cmake -B build -DSPP_WITH_OPENMP=ON
```

Beispielaufrufe Der Aufruf der entstehenden executable muss mit Übergabe eines Pfades zu einer Eingabedatei (Simulationsdaten) geschehen. Optional kann man noch einen Pfad zu einer Ausgabedatei für die Speicherung der Simulationsergebnisse übergeben.

- Beispielaufruf: `./bin/spp-nbody-omp scenarios/nbody.txt nbody-out.txt`
- Zum Generieren von randomisierten Szenarien: `./bin/spp-nbody-omp -g *out file* *num timesteps* *num bodies*`

Hinweis: Die Zeitmessungen sollen *ohne* aktivierte GUI und mit Optimierungen stattfinden.

Aufgabenstellung

a) **5 Punkte** `Body`: Hier sind 2 Funktionen zu vervollständigen.

- `Body::applyForces`, bei der die Wirkung eines anderen Körpers auf die aktuelle Beschleinigung berechnet wird.
- `Body::update`, berechnet Geschwindigkeit und Position basierend auf der Zeitschrittweite `dt`.

b) **8 Punkte** `Simulation::nextTimestep`: Implementieren Sie eine serielle Variante des Codes, der das N-body Problem simuliert (siehe *Simulation Pseudocode*). Sie sollten hierbei auch die GUI verwenden, um bspw. das Nbody-Szenario zu testen.

c) **8 Punkte** `Simulation::handleCollision`: Implementieren Sie eine serielle Kollisionsbehandlung basierend auf der Impulserhaltung, siehe *Behandlung von Kollisionen*. Rufen Sie `handleCollision` während `nextTimestep` an geeigneter Stelle *vor* Berechnung der Kräfte auf. Sie sollten hierbei auch die GUI verwenden, um bspw. das Collision-Szenario (mit 2-4 Körpern) zu testen. Verändern Sie in der Datei die Zahl der Körper (zweiter Wert, siehe *Datenformat*), damit 2,3,4 Körper ausgelesen werden und kollidieren.

d) **12 Punkte** Nachdem der serielle Code funktioniert, soll nun OpenMP verwendet werden, welches den Code innerhalb eines Zeitschritts beschleunigt. Beschleunigen Sie die Simulation der Körper *und* der Kollisionsbehandlung. Achtung, eine Synchronisation zwischen den Threads ist hier eventuell nötig.

e) **6 Punkte** Der Einsatz von OpenMP lohnt sich erst ab einer bestimmten Menge an Körpern in der Simulation. Testen sie verschiedene Größen. Wie ist der Zeitgewinn bei variabler Threadzahl (2,4,8,16) für eine hinreichend große Anzahl an Körpern (`random.txt` ist zu klein). Wie sieht hier der Vergleich zwischen seriellem (ohne OpenMP flags) und parallelem Code aus? *Hinweis*: Zeitmessungen nur mit aktivierten Optimierungen, ohne aktive GUI (siehe *Build Prozess*). Mit der environment variable `OMP_NUM_THREADS` koennen Sie die Anzahl der OpenMP-Threads festlegen, siehe Vorlesungsfolien zu OpenMP.