Übungen zur Vorlesung

Paralleles Höchstleistungsrechnen

Dr. S. Lang, D. Popović

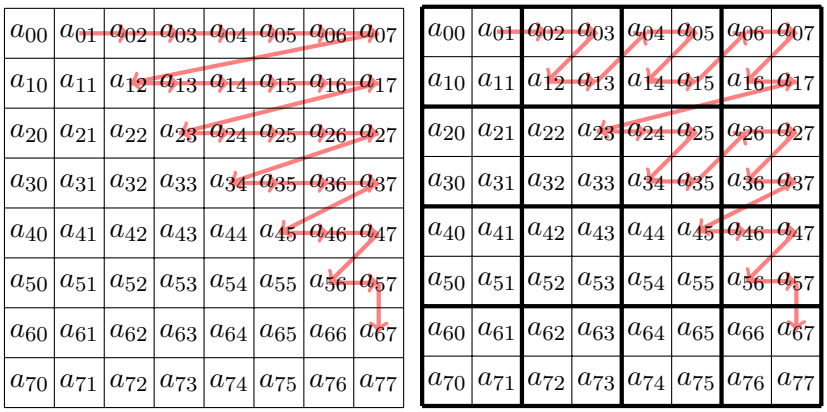
Abgabe: 12. Januar 2010 in der Übung

Beginnend mit dieser Übung wollen wir ein zum in der Vorlesung besprochenen N-Körper-Problem ähnliches Problem untersuchen und mit den bisher verwendeten Techniken parallelisieren. Lesen Sie zunächst den Abschnitt am Ende des Übungsblattes, in dem die von uns verwendete Variante des N-Körper-Problems erklärt wird. In den bereitgestellten Dateien (Hauptdatei `nbody_vanilla.c`) finden Sie eine serielle Implementierung des Problems. Machen Sie sich mit dem Code vertraut, der die Bewegung mehrerer Körper im leeren Raum, die sich gegenseitig durch Gravitation anziehen, berechnet. Bemerkungen finden Sie in den Kommentaren und im letzten Abschnitt dieses Übungsblattes.

Wir werden die verwendeten Funktionen durch Kachelung, mit *OpenMP* und *MPI* parallelisieren und auch auf der Grafikkarte implementieren. Es wird für jede Variante eine Kopie des seriellen Codes zur Verfügung gestellt, die Sie verwenden können. Natürlich dürfen Sie alle Änderungen auch in eine Datei einbauen, achten Sie dann darauf, dass Sie immer eine serielle lauffähige Variante haben (d.h. etwa, für die Kachelung eine neue Funktion schreiben und nicht nur die serielle verändern).

Übung 25 N-Körper-Problem mit Kachelung (10 Punkte)

In der seriellen Variante des N-Körper-Problems berechnet die Funktion `acceleration()` die Beschleunigung aller Körper. Für einen Körper i muss sie über alle anderen Körper $j \in [i+1, \ldots, N-1]$ iterieren und ihre Positionen `x[j]` und Massen `m[j]` laden, die Beschleunigungen $a_{ij} = -a_{ji}$ berechnen und in `a[i]` und `a[j]` akkumulieren. Wir wollen nun versuchen, das Programm durch Kachelung zu Beschleunigen. In der folgenden Abbildung wird die serielle Durchlaufreihenfolge zur Berechnung der Beschleunigungen links gezeigt, rechts ist die optimierte Reihenfolge für das Kacheln zu sehen, mit einer Blockgröße von $B = 2$.



Verwenden Sie das Gerüst `nbody_tiled.c`, das eine Kopie des seriellen Codes aus `nbody_vanilla.c` enthält, und verändern Sie die Funktion `acceleration()`, so dass sie eine Kachelung verwendet. Die Loop über `i` und `j` können Sie in Loops über die Blöcke und innere Loops aufteilen, zum Beispiel:

```
for(I = 0; I < N; I += B)
  for(J = I; J < N; J += B)
    for(i = I; i < MIN(N, I+B); ++i)
      for(j = MAX(i+1,J); j < MIN(N, J+B); ++j) {
        /* code goes here */
      }
```

(a) Führen Sie nun eine Perfomance-Analyse mit Kachelung durch, in dem Sie die *MFLOP*-Rate messen. Verwenden Sie dazu verschiedene Werte von N und B. Wählen Sie die Problemgröße auch so, dass sie auf jeden Fall nicht mehr in den L2-Cache passt. Bei den Testmessungen ergab sich im Pool mit Kachelung keine Verbesserung der *MFLOP*-Rate, wenn es Ihnen also auch nicht gelingt, wundern Sie sich nicht. Auf anderen Rechner konnte mit Kachelung hingegen ein Performance-Gewinn wie erwartet festgestellt werden.

(b) Um das Problem des nicht vorhandenen Perfomance-Gewinns genauer zu untersuchen, wollen wir das Datenlayout ändern, um die Daten in anderer Reihenfolge in den Cache zu laden. Definieren Sie dazu eine Datenstruktur `Body`, die `m, x, v` und `a` für jeden Körper speichert, und legen Sie dann ein Feld `Body[n]` für die `n` Körper an. Die temporären Beschleunigungen können Sie in einem Feld wie gehabt lassen. Ändern Sie nun die Funktionen `accelerate` und `leapfrog` so, dass sie jeweils volle Körper laden. Im Gegensatz zur Speicherung einzelner Felder für die Komponenten liegen die Körper nun konsekutiv im Speicher. Ist eine Verbesserung der in (a) durchgeführten Messungen erkennbar? Alternativ dürfen Sie gerne eigene Ideen zur Ausnutzung des Caches einbauen.

Übung 26 N-Körper-Problem mit *OpenMP* (10 Punkte)

Parallelisieren Sie das N-Körper-Problem mit *OpenMP*. Verwenden Sie dazu die Datei `nbody_openmp.c`, die eine Kopie von `nbody_vanilla.c` ist. Parallelisieren Sie nur die Funktion `acceleration()`, die die Hauptlast der Berechnungen trägt (sie hat die Komplexität $O(N^2)$, alles andere hingegen $O(N)$). Vergleichen Sie die Zeiten des parallelen Programms mit der sequentiellen Variante und messen Sie den Speed-Up.

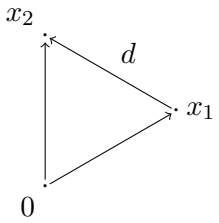
Hinweis: Um die parallele Variante auf Richtigkeit zu prüfen, können Sie die generierten *VTK*-Dateien vergleichen. Die Simulations-Ergebnisse können Sich aber in einigen Nachkommastellen unterscheiden. Das bereitgestellte Skript `fuzzy_diff` kann zwei *VTK*-Dateien vergleichen bezüglich einer vorgegebenen Toleranz:

```
./fuzzy_diff sequential.vtk parallel.vtk 1e-10
```

Das Gravitations-N-Körper-Problem

Dieser Abschnitt erklärt in Kürze die verwendete Modellierung und Numerik des N-Körper-Problems sowie die Implementierung und verwendeten Tools.

Es sei folgende Situation mit zwei beteiligten Körpern im leeren Raum gegeben:



$$x_1 + d = x_2$$
$$\Rightarrow d = x_2 - x_1$$

Mit dem Gesetz von Newton kann die Kraft, die Körper 1 mit Masse m_1 auf Körper 2 mit der Masse m_2 ausübt, ausgedrückt werden:

$$F_{12}(t) = \gamma \frac{m_1 m_2}{\|x_2 - x_1\|^3}(x_2 - x_1),$$

wobei γ die Gravitationskonstante ist. Für den Fall mit N Körpern ist die Gesamt-Kraft auf Körper i durch die Superposition der Teilkräfte gegeben:

$$F_i(t) = \sum_{\substack{j=0 \\ j\neq i}}^{N-1} \gamma \frac{m_i m_j}{\|x_j - x_i\|^3}(x_j - x_i) \qquad \forall i = 0, \ldots, N-1$$

Das zweite Newtonsche Gesetz verknüpft die Beschleunigung eines Körpers mit der Kraft, die auf ihn ausgeübt wird: $F_i(t) = m_i a_i(t) \quad \forall i = 0, \ldots, N-1$. Jeder Körper i hat eine Geschwindigkeit $v_i(t) = dx_i(t)/dt$ und $a_i(t) = dv_i(t)/dt$, woraus sich folgendes lineare System gewöhnlicher Differentialgleichungen ergibt:

$$\left.\begin{aligned} \frac{dx_i(t)}{dt} &= v_i(t) \\ \frac{dv_i(t)}{dt} &= \sum_{\substack{j=0 \\ j\neq i}}^{N-1} \gamma \frac{m_j}{\|x_j - x_i\|^3}(x_j - x_i) \end{aligned}\right\} \qquad \forall i = 0, \ldots, N-1.$$

Im Folgenden wird folgende Bezeichnung für die Unbekannten $x_i(t)$ verwendet:

$$\vec{x}(t) = \begin{pmatrix} \vec{x}_0(t) \\ \vdots \\ \vec{x}_{N-1}(t) \end{pmatrix} = \begin{pmatrix} (x_0(t))_0 \\ (x_0(t))_1 \\ (x_0(t))_2 \\ (x_1(t))_0 \\ \vdots \end{pmatrix}$$

In der gleichen Weise werden $\vec{a}(t)$, $\vec{v}(t)$ and $\vec{m}$ verwendet.

Massenschwerpunkts-Korrektur

Der Massenschwerpunkt und seine Geschwindigkeit sind

$$R(t) = \frac{1}{M}\sum_{i=0}^{N-1} m_i x_i \qquad V(t) = \frac{1}{M}\sum_{i=0}^{N-1} m_i v_i \qquad \text{mit} \quad M = \sum_{i=0}^{N-1} m_i.$$

Da keine externen Kräfte vorhanden sind, ist $V(t)$ konstant und der Schwerpunkt bewegt sich geradlinig. Daher kann man Massenschwerpunkt am Usrprung eines Koordinatensystems festlegen:

$$\left.\begin{aligned} \tilde{x}_i(0) &= x_i(0) - R(0) \\ \tilde{v}_i(0) &= v_i(0) - V(0) \end{aligned}\right\} \qquad \forall i = 0, \ldots, N-1.$$

Diese Korrektur wird einmal am Beginn vorgenommen. Die Tilde $\tilde{}$ wird von jetzt an nicht mehr mitgeschrieben und es wird angenommen dass $R(t) = V(t) = 0$.

Stabilisierung des Modells

Wenn zwei Körper kollidieren, $x_j \to x_i$, wird die Kraft, die ein Körper auf den Anderen ausübt, unendlich gross, $F_{ij} \to \infty$. Um diesen Fall numerisch handzuhaben wird der Nenner in der Beschleunigung so geändert, dass er niemals 0 wird:

$$a_{ij} = \frac{\gamma m_i}{(\|x_j - x_i\|^2 + \epsilon^2)^{3/2}}(r_j - r_i)$$

Die korrespondierende Kraft kann vom sogenannten *Plummer*-Potential

$$\Phi(x_i, x_j, \epsilon) = -\frac{\gamma m_i m_j}{(\|x_j - x_i\|^2 + \epsilon^2)^{1/2}}$$

abgeleitet werden. Die physikalische Interpretation ist, dass die ursprünglich hergeleitete Kraft für Punktmassen gilt, während die korrigierte für Massen mit ausgedehnter, variabler Dichteverteilung

$$\rho(r) \propto \frac{1}{(r^2 + \epsilon^2)^{5/2}}$$

gilt. Das *Plummer*-Potential hat ausserdem den Vorteil, an der Singularitätsstelle des zugehörigen Gravitationspotentials stetig und differenzierbar zu sein.

Numerik

Für die Zeitdiskretisierung wird ein Einschritt-Verfahren mit folgender Iterations-Vorschrift verwendet:

$$\begin{aligned}\vec{x}^{k+1} &= \vec{x}^k + \vec{v}^k \cdot \Delta t + \vec{a}^k \tfrac{1}{2}(\Delta t)^2 \\ \vec{v}^{k+1} &= \vec{v}^k + (\vec{a}^k + \vec{a}^{k+1}) \cdot \tfrac{1}{2}\Delta t\end{aligned}$$

Dieses Verfahren

- braucht nur eine Auswertung von $\vec{a}$ pro Schritt (falls $\vec{a}^k$ und $\vec{a}^{k+1}$ gespeichert werden),
- kann Δt lokal adaptiv verwenden, (allerdings unter Verlust der Energieerhaltung),
- ist 2. Ordnung akkurat in der Zeitdiskretisierung.

Die Konvergenz zweiter Ordnung sieht an bei Verwendung einer Taylor-Entwicklung für $\vec{x}(t)$ und $\vec{v}(t)$. Für $\vec{x}(t)$ lautet diese zum Beispiel:

$$\begin{aligned}\vec{x}(t + \Delta t) &= \vec{x}(t) + \Delta t \frac{d\vec{x}(t)}{dt} + \tfrac{1}{2}\Delta t^2 \frac{d^2\vec{v}(t)}{dt^2} + O(\Delta t^3) \\ &= \vec{x}(t) + \Delta t \vec{v}(t) + \tfrac{1}{2}\Delta t^2 \vec{a}(t) + O(\Delta t^3) \\ \rightsquigarrow \vec{x}^{k+1} &= \vec{x}^k + \vec{v}^k \cdot \Delta t + \vec{a}^k \tfrac{1}{2}(\Delta t)^2.\end{aligned}$$

In ähnlicher Weise gilt für $\vec{v}(t)$:

$$\begin{aligned}\vec{v}(t + \Delta t) &= \vec{v}(t) + \Delta t \frac{d\vec{v}(t)}{dt} + \tfrac{1}{2}\Delta t^2 \frac{d^2\vec{v}(t)}{dt^2} + O(\Delta t^3) \\ &= \vec{v}(t) + \Delta t \vec{a}(t) + \tfrac{1}{2}\Delta t^2 \frac{d\vec{a}(t)}{dt} + O(\Delta t^3).\end{aligned}$$

Die verbleibende Zeitableitung $d\vec{a}(t)/dt$ kann mit der Taylorentwicklung für $a(t)$ behandelt werden:

$$\frac{d\vec{a}(t)}{dt} = \frac{\vec{a}(t + \Delta t) - \vec{a}(t)}{\Delta t} + O(\Delta t),$$

was zu

$$\begin{aligned}\vec{v}(t + \Delta t) &= \vec{v}(t) + \tfrac{1}{2}\Delta t \vec{a}(t) + \tfrac{1}{2}\Delta t \vec{a}(t + \Delta t) + O(\Delta t^3) \\ \rightsquigarrow \vec{v}^{k+1} &= \vec{v}^k \tfrac{1}{2}\Delta t \vec{a}^k + \tfrac{1}{2}\Delta t \vec{a}^{k+1}\end{aligned}$$

führt. Daher hat das Verfahren die Ordnung 2.

Implementierung

Datenstrukturen

Die Daten $\vec{x}^k$, $\vec{v}^k$, $\vec{a}^k$, $\vec{a}^{k+1}$ und m werden in der Form *Name – Körper-Id – Komponente* gespeichert, also `double x[N][3], v[N][3], a[N][3], anew[N][3], m[N]`.

Zeitmessung

Die Funktion `double get_time()` aus `stopwatch.h` gibt die Wall-Time in s zurück. Zur Zeitmessung werden Differenzen der Rückgabewerte der Funktion betrachtet. Der Code kann die gemessenen *MFLOP*-Rate ausgeben.

Anfangsbedingungen

Es gibt zwei Funktionen, um die Anfangswerte für die Massen und Geschwindigkeiten zu erzeugen:

`plummer()` Alle Körper haben die gleiche Masse $\frac{1}{N}$:

$$\Phi(r) = -\gamma M \frac{1}{(r^2 + \vec{a}^2)^{1/2}}.$$

Das System ist im Gleichgewicht, und die Massenschwerpunkt-Korrektur wird angewendet.

`cube()` Alle Körper sind zufällig in einem Würfel verteilt. Die Körper haben keine Geschwindigkeit und ihre Masse ist zufällig in $m \pm \Delta m$ verteilt. Die Massenschwerpunkts-Korrektur wird verwendet.

Die Daten sind in Dateien im *VTK*-Format (Visualisation Toolkit, siehe den nächsten Abschnitt) gespeichert. Die Funktionen, um diese Dateien zu Lesen und zu Schreiben heissen `read_vtk_file_double()` und `write_vtk_file_double()` und liegen in `io_vanilla.h`.

Ein- und Ausgabe von Daten

Der Code erzeugt zum Visualisieren der Simulationsergebnisse *Paraview*-Dateien im *VTK*-Format (siehe dazu auch die Hinweise auf der Homepage). Diese Dateien haben je nach Art der dargestellten Daten verschiedene Endungen, polygonale Daten werden in `.vtp`, Daten auf unstrukturierten Gittern in `.vtu` Dateien gespeichert. Die Dateien können mit *Paraview* geöffnet und angesehen werden. Bei Zeitreihen werden mehrere Dateien, für jeden Zeitpunkt eine, herausgeschrieben. Öffnet man die erste Datei, lädt *Paraview* automatisch die konsekutiv folgenden und im Animations-Dialog kann man einen Film starten. Die Software startet im Pool mit `paraview`. Wenn Sie sich über `ssh` einloggen, vergessen Sie bitte nicht, die Grafikausgabe mit dem Flag `-X` auf Ihren lokalen Rechner umzuleiten: `ssh -X phlr025@pool.iwr.uni-heidelberg.de`.

Das Haupt-Programm

`main()` Die `main()`-Funktion liest Programm-Parameter, allokiert Speicher für die Variablen und initialisiert sie. Ausserdem startet sie die Zeitschritt-Intregration (Aufruf von `leapfrog()` und schreibt Output in Dateien.

`leapfrog()` Diese Funktion führt einen Zeitschritt aus.

Input-Parameter: Anzahl Körper n, Zeitschrittweite `dt`, Vektoren der alten Positionen `x` und alten Geschwindigkeiten `v`, der Massen `m`, und der alten Beschleunigungen `a`.

Output-Parameter: Vektoren der neuen Positionen `x`, neuen Geschwindigkeiten `v` und neuen Beschleunigungen `a`.

Der zusätzliche Parameter `aold` wird als temporärer Speicher für die alten Beschleunigungen verwendet. Seine Werte bei Input and Output sind aber nicht relevant.

`acceleration()` Diese Funktion wird von `main()` verwendet, um die initialen Werte der Beschleunigungen aus den initialen Werten der Positionen zu berechnen. Auch `leapfrog()` verwendet sie, um die neuen Beschleunigungs-Vektoren aus den neu berechneten Positionen zu berechnen.

Input-Parameter: Vektoren der Positionen `x` und Massen `m`.

Output-Parameter: Vektor der Beschleunigungen `a`. Die Funktion akkumuliert die Beschleunigungen in dem Vektor, so dass der Vektor mit bei Eintritt mit 0 belegt werden muss.

Die Funktion nutzt die Symmetrie in der Wirkung zweier Körper i und j auf die Beschleunigung des jeweils Anderen ($a_{ij} = -a_{ji}$).