Marcell Kehmstedt
m.kehmstedt@tu-bs.de

MPI-Datentypen

Basisdatentypen

Die Funktionen von MPI für das Versenden und Empfangen von Nachrichten sowie alle Funktionen, die mit lokalen Speicherbereichen in irgendeiner Form zu tun haben, verwenden abstrakte MPI-spezifische Datentypen, die das interne Layout sowie die Größe des benötigten Speichers für ein Element eines solchen Typs spezifizieren.

Nach dem Aufruf von `MPI_Init` stehen die sog. Basisdatentypen unmittelbar zur Verfügung. Diese sind wie folgt aufgeführt, samt ihrer Entsprechung in C:

MPI_CHAR	`signed char`
MPI_SHORT	`signed short int`
MPI_INT	`signed int`
MPI_LONG	`signed long int`
MPI_UNSIGNED_CHAR	`unsigned char`
MPI_UNSIGNED_SHORT	`unsigned short int`
MPI_UNSIGNED	`unsigned int`
MPI_UNSIGNED_LONG	`unsigned long int`
MPI_FLOAT	`float`
MPI_DOUBLE	`double`
MPI_LONG_DOUBLE	`long double`
MPI_BYTE	(keine Entsprechung; typloser 8bit Speicherbereich)
MPI_PACKED	(keine Entsprechung; kennzeichnet gepackte Daten)

Abbildung 1: MPI-Basisdatentypen

Abgeleitete Datentypen

Neben den Basisdatentypen bietet MPI die Möglichkeit, eigene Datentypen zu definieren. Diese können mithilfe spezieller Ableitungsoperatoren aus den Basisdatentypen konstruiert werden. Vorteil von abgeleiteten Datentypen ist die Möglichkeit, größere Speicherbereiche mit logischem Zusammenhang im Code geschlossen abzubilden. Darüberhinaus können nicht zusammenhängende Speicherbereiche einfach verarbeitet sowie Strides (Zugriff auf Elemente mit festem Abstand) auf einfache Weise behandelt werden, ohne die nötigen Sprünge im Speicherbereich von Hand ausformulieren zu müssen.

Vorgehensweise

Zunächst wird ein abgeleiteter Datentyp mithilfe eines der Konstruktoren für abgeleitete Datentypen erzeugt. Bevor dieser jedoch verwendet werden kann, muss er im MPI-Subsystem „angemeldet“ werden. Nach der Anmeldung kann der neue Datentyp ganz normal überall dort verwendet werden, wo MPI einen MPI-Datentyp erwartet. Ein nicht mehr benötigter Datentyp kann auch wieder „abgemeldet“ werden. Basisdatentypen müssen nicht explizit angemeldet werden.

MPI_Type_commit Meldet einen abgeleiteten Datentyp an.
Syntax: `MPI_Type_commit(&datatype)`

MPI_Type_free Meldet einen angemeldeten Datentyp ab.
Syntax: `MPI_Type_free(&datatype)`

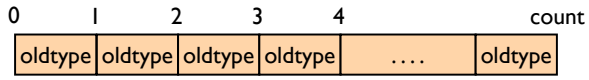
Marcell Kehmstedt
m.kehmstedt@tu-bs.de

Beispiel:

```
#include "mpi.h"
...
MPI_Datatype datatype;
...
// construct derived datatype here
...
MPI_Type_commit(&datatype);
...
MPI_Type_free(&datatype);
```

Konstruktoren

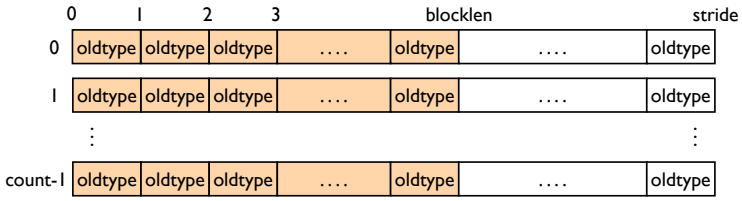
MPI_Type_contiguous Gleichartige Elemente zusammenhängend im Speicher.
Syntax: `MPI_Type_contiguous(count, oldtype, &newtype)`
count Anzahl der Elemente vom Typ `oldtype`
oldtype Typ der aneinandergereihten Elemente



Beispiel:

```
int count;
MPI_Datatype oldtype, newtype;
...
MPI_Type_contiguous(count, oldtype, &newtype);
```

MPI_Type_vector Blöcke gleichartiger Elemente mit Versatz.
Syntax: `MPI_Type_vector(count, blocklen, stride, oldtype, &newtype)`
count Anzahl der Blöcke
blocklen Anzahl der Elemente je Block
stride Anzahl der Elemente zwischen je zwei Blockanfängen (insbondere gilt: blocklen $\leq$ stride)
oldtype Typ der Blockelemente

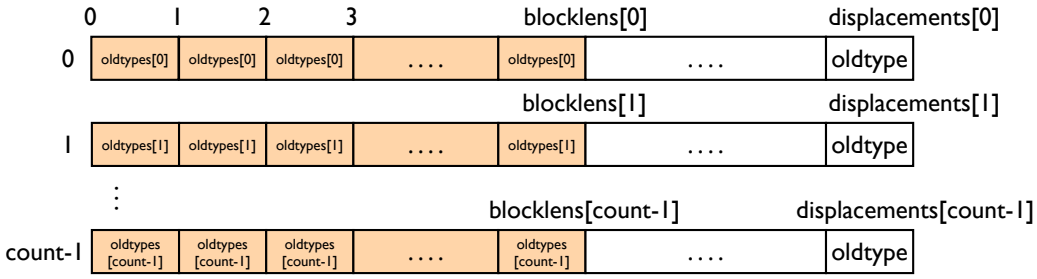


Beispiel:

```
MPI_Datatype FIRST2COLUMNS;
double *matrix; // initialised to be a matrix of size NxN
int count = N;
blocklen = 2;
stride = N;
MPI_Type_contiguous(count, blocklen, stride, MPI_DOUBLE, &
   FIRST2COLUMNS);
```

Marcell Kehmstedt
m.kehmstedt@tu-bs.de

MPI_Type_struct Blöcke unterschiedlicher Elemente, Größe und Versatz.
Syntax:
`MPI_Type_struct(count, blocklens, displacements, oldtypes, &newtype)`
count Anzahl der Blöcke
blocklens Anzahlen der Elemente je Block (Feld von Integern)
displacements Anzahl von Bytes zwischen je zwei Blockanfängen (Feld von Integern)
oldtypes Typ der Blockelemente je Block (Feld von Datentypen)



Transparenz

Dank der abgeleiteten Datentypen wird das Versenden und Empfangen von Strukturen durch MPI transparent. MPI kümmert sich selbständig darum, dass die Datentypen richtig in Nachrichten verpackt und beim Empfänger wieder ausgepackt werden. Ein „Zählen von Bytes“ sowie Sprünge im Speicher unter Beachtung plattformspezifischer Eigenheiten wird vor dem Entwickler verborgen und somit überflüssig. Dies spart nicht nur Zeit und Nerven, sondern macht den

```
#include "mpi.h"
...
unsigned char *rgbimage; // rgb-image with N pixels and 8bit per
    color
MPI_Datatype PIXELCOMPONENTS;
MPI_Type_vector(N, 1, 3, MPI_CHAR, &PIXELCOMPONENTS); // vector of
    specific component of all N pixels
...
MPI_Send(rgbimage+2, 1, PIXELCOMPONENTS, destination, messageTag,
    comm); // send blue component of every pixel
MPI_Send(rgbimage+0, 1, PIXELCOMPONENTS, destination, messageTag,
    comm); // send red component of every pixel
MPI_Send(rgbimage+1, 1, PIXELCOMPONENTS, destination, messageTag,
    comm); // send green component of every pixel
// Warning: This is only an example. Don't send an image component-
    wise like this!
```

Abbildung 2: Versand mit abgeleiteten Datentypen

Code auch deutlich lesbarer, was Fehlern vorbeugt, sowie häufig auch performanter, da benötigte Operationen durch die optimierten MPI-Bibliotheken übernommen werden. Besonders für den Versand von Datenstrukturen mit begrenzter Größe sind diese also ideal geeignet.

Marcell Kehmstedt
m.kehmstedt@tu-bs.de

Gepackte Daten

Neben dem Versand/Empfang von abgeleiteten Datentypen und den damit verbundenen autmatischen Verpackoperationen druch MPI wird auch das sog. „Packen“ von Daten unterstützt. Dabei werden Daten beliebiger MPI-Datentypen, die nicht-zusammenhängend im Speicher vorliegen müssen, in einen zusammenhängenden Speicherbereich kopiert. Dieser Speicherbereich kann anschließend mit einer einzelnen Versandoperation übertragen werden. Nach dem Empfang der kompletten Nachricht beim Empfänger kann diese mit einer umgekehrten Auspackoperation wieder in ihre ursprüngliche Bestandteile zerlegt werden.

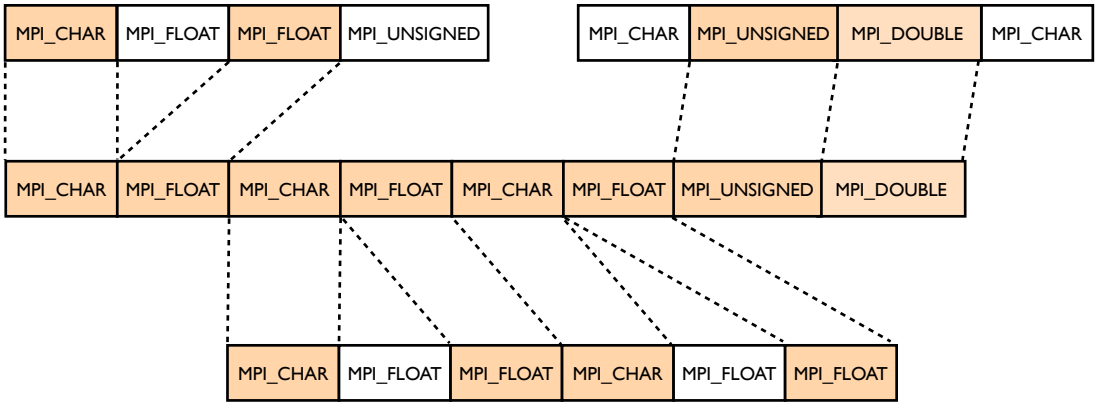


Abbildung 3: Daten in zusammenhängenden Speicher gepackt

Verwendung

Für das Übertragen von nicht-zusammenhängenden Datentypen können in MPI abgeleitete Datentypen verwendet werden. Damit ist es prinzipiell möglich, solche Daten mit nur einer Sende- bzw. Empfangsoperation zu übertragen. Allerdings müssen alle Daten, die als solche beliebig im Speicher zerstreut vorliegen können, zunächst durch einen abgeleiteten Datentypen gemeinsam beschrieben werden. Bei vielfacher Verwendung des Typmechanismus entstehen dann leicht abgeleitete Datentypen aus anderen abgeleiteten Datentypen und damit schließlich ganze Hierachien. Anstelle bei jeder Gelegenheit also einen neuen ageleiteten Datentyp zu erstellen, der womöglich nur für eine einzige Datenübertragung benötigt wird, können die Daten in einen zusammenhängenden Puffer kopiert und gemeinsam übertragen werden. Mit den beiden Funktionen `MPI_Pack` sowie `MPI_Unpack` leistet MPI hierbei gute Dienste.

Performance

Weiterhin gilt es zu bedenken, dass der Versand weniger Nachrichten mit größerem Inhalt dem Versand vieler kleiner Nachrichten vorzuziehen ist. Die Bandbreite des Übertragungskanals bleibt jeweils gleich und die Übertragungszeit der Nutzdaten verhält sich zu ihr proportional. Zu dieser reinen Übertragungszeit kommt noch die sog. Startzeit (Latenzzeit), die konstant ist und zu jeder Nachricht dazukommt; egal wie kurz die Nachricht sein mag. Große Nachrichten haben also ein günstigeres Verhältnis von Übertragungszeit zu Startzeit und sind daher wann immer es möglich ist, vorzuziehen. Das Packen von Daten sowie ihre Übertragung in einer einzelnen Operation sind also eine effiziente Art verschiedenartige bzw. verstreut vorliegende Daten zu übertragen und bieten i.A. die beste Performance.

Marcell Kehmstedt
m.kehmstedt@tu-bs.de

Funktionen

MPI_Pack Packen von Daten in zusammenhängenden Speicherbereich.
Syntax: `MPI_Pack(&in, count, type, &out, outmax, &pos, comm)`
in Puffer aus dem zu packende Daten geholt werden.
count Anzahl von Elementen vom Type *type*, die gepackt werden sollen.
type Datentyp der zu kopierenden Elemente.
out Puffer, in den die Elemente hintereinander kopiert werden sollen.
outmax Größe des Puffers in Bytes.
pos Position im Ausgabepuffer (in Bytes) ab der Daten eingefügt werden sollen. Wird nach der Operation um die Größe der eingefügten Daten erhöht.

MPI_Unpack Auspacken von Daten.
Syntax: `MPI_Unpack(&in, insize, &pos, &out, count, type, comm)`
in Puffer aus dem gepackte Daten ausgepackt werden sollen.
insize Größe des Eingabepuffers (in Bytes).
pos Position im Eingabepuffer (in Bytes) ab der Daten ausgepackt werden sollen. Wird nach der Operation um die Größe der ausgepackten Daten erhöht.
out Puffer, in den die Elemente hintereinander kopiert werden sollen.
count Anzahl der auszupackenden Elemente vom Typ *type*.
type Datentyp der zu auszupackenden Elemente.

Das Mehrkörperproblem

Ziel dieses letzten Übungsblattes soll die Umsetzung des Mehrkörperproblems unter Verwendung von abgeleiteten MPI-Datentypen sowie gepackten Daten sein. Das Mehrkörperproblem befasst sich mit der Simulation von mehreren Körpern verschiedener Größe und Masse, die dem Einfluss einer besonderen Kraft, der Graviationskraft, unterworfen sind und sich demzufolge gegenseitig anziehen.

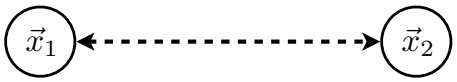


Abbildung 4: Gegenseitige Anziehung zweier Körper

Newtonsches Gravitationsgesetz

Nach dem Newtonschen Gravitationsgesetz wirkt ein Körper i am Ort $\boldsymbol{x}_i$ und der Masse m_i auf einen zweiten Körper j eine Kraft, die Gravitationskraft,

$$\boldsymbol{F}_{ij} = Gm_i m_j \frac{\boldsymbol{x}_j - \boldsymbol{x}_i}{|\boldsymbol{x}_j - \boldsymbol{x}_i|^3}$$

aus. Eine gleichgroße aber entgegengesetzt gerichtete Kraft

$$\boldsymbol{F}_{ji} = -\boldsymbol{F}_{ij}$$

wirkt seinerseits der Körper j auf den Körper i aus.
G ist die sog. Gravitationskonstante und es gilt:

$$G = 6,673 \cdot 10^{-11} \frac{m^3}{kg \cdot s^2}$$

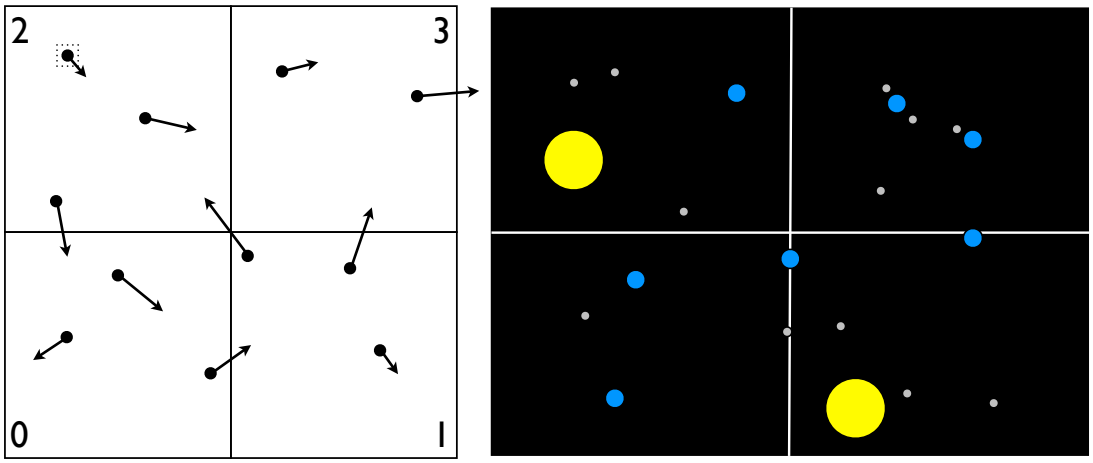
Marcell Kehmstedt
m.kehmstedt@tu-bs.de

Wirken mehrere Körper aufeinander, so addieren sich die einzelnen Gravitationskräfte zu einer Gesamtkraft. Für n Körper $i_1, i_2, \ldots, i_n$ wirkt auf den Körper i also die Gesamtkraft:

$$\boldsymbol{F}_i = \sum_{j \neq i} \boldsymbol{F}_{ij}$$

Modellproblem

Bei dem betrachteten Problem soll es sich um ein Modell von Himmelskörpern (Sonnen, Planeten, Kometen, etc.) handeln, die sich in einem Modellausschnitt des Universums durch gegenseitige Gravitationseinwirkung bewegen. Als Ausschnitt des Universums wollen wir ein endlich großes, rechteckiges Gebiet betrachten. Dieses Gebiet soll entsprechend der Anzahl paralleler Prozesse aufgeteilt und von je einem der Prozesse behandelt werden. In jedem dieser Gebiete befinden sich zu einem Zeitpunkt eine bestimmte Menge von Himmelskörpern. Von Zeitpunkt zu Zeitpunkt bewegen sich diese unter dem Einfluss der Gravitation.



Es ist klar, dass durch die Bewegung die Objekte das Gebiet eines Prozesses verlassen können. In diesem Fall müssen sie einem neuen Prozess zugeordnet werden. Nämlich demjenigen, der für das Gebiet zuständig ist, in den das jeweilige Objekt eintritt. So kann es auch passieren, dass sich in manchen Gebieten sehr viele Objekte zu einem Zeitpunkt aufhalten; in anderen Gebieten dagegen wenige bis gar keine. Die Prozesse haben somit unterschiedlich viel zu tun, was der parallelen Effizienz nicht zuträglich ist. Eine effiziente Implementierung würde dies berücksichtigen und die Größe der Gebiete sowie Zuordnung zu den Prozessen dynamisch anpassen. Man spricht von **Load-Balancing**.

Simulation

Simulation ist ein zeitlicher Prozess, der in diskreten Zeitschritten voranschreitet. Die Himmelskörper bewegen sich von Zeitschritt zu Schritt jeweils unter dem Einfluss der Gravitationskraft, die zu einem bestimmten Zeitpunkt auf sie einwirkt. Am darauffolgenden Zeitpunkt wirkt eine andere Kraft, welche in jedem Zeitschritt für alle Körper neu berechnet werden muss. Die Bewegung selbst erfolgt durch Anpassung der Position $\boldsymbol{x}_i$ eines Körpers i. Für die Position gilt dabei, dass deren zweite zeitliche Ableitung gleich der Beschleunigung a_i des Körpers zum jeweiligen Zeitpunkt ist. In Zeichen:

$$\frac{\partial^2}{\partial t^2} \boldsymbol{x}_i = a_i$$

Marcell Kehmstedt
m.kehmstedt@tu-bs.de

Aus dem zweiten Newtonschen Gesetz folgt der Zusammenhang $\boldsymbol{F}_i = m_i \cdot \boldsymbol{a}_i$, wodurch man nach Umstellen die Beschleunigung erhält:

$$\boldsymbol{a}_i = \frac{\boldsymbol{F}_i}{m_i}$$

Setzt man nun alle diese Formeln zusammen, so erhält man den Ort eines Körpers zu einem Zeitpunkt in Abhängigkeit von seiner Masse und der auf ihn wirkenden Kräfte:

$$\boldsymbol{x}_i = \int\int \boldsymbol{a}_i = \int\int \frac{\boldsymbol{F}_i}{m_i} = \int\int \frac{\sum\limits_{j \neq i} \boldsymbol{F}_{ij}}{m_i} = \int\int \sum_{j \neq i} G m_j \frac{\boldsymbol{x}_j - \boldsymbol{x}_i}{|\boldsymbol{x}_j - \boldsymbol{x}_i|^3}$$

Beim Lösen jedes dieser Integrale erhält man je eine Integrationskonstante. Im Zuge dieses Modells kommt beiden eine Bedeutung zu als *Startort* $\boldsymbol{x}_i(0)$ und *Initialgeschwindigkeit* $\boldsymbol{v}_i(0)$.

Numerische Integration

Bleibt noch die Frage zu klären wie die beiden Integrale zu lösen sind. Hierfür kennt die Numerische Mathematik die Klasse der Runge-Kutta-Verfahren. Das einfachste dieser Verfahren ist das **eulersche Polygonzugverfahren**. Hierbei handelt es sich um ein explizites Runge-Kutta-Verfahren 1. Ordnung. Wendet man dieses Verfahren zweimal auf die obige Gleichung an, so erhält man schließlich folgende Berechnungsvorschrift:

$$\begin{pmatrix} \boldsymbol{x}_i^{(t)} \\ \boldsymbol{v}_i^{(t)} \end{pmatrix} := \begin{pmatrix} \boldsymbol{x}_i^{(t-1)} + \delta t \cdot \boldsymbol{v}_i^{(t-1)} \\ \boldsymbol{v}_i^{(t-1)} + \delta t \cdot \boldsymbol{a}_i^{(t-1)} \end{pmatrix}$$

$\boldsymbol{v}_i$ ist die Geschwindigkeit des Körpers i.

Kollisionen

Wo sich Körper frei bewegen kann es zu Kollisionen kommen. Kollision liegt genau dann vor, wenn zwei Körper mit ihrer Ausdehnung Teile des Raumes gleichzeitig überdecken. Für die hier behandelten Himmelskörper kann deren äußere Form einfach als Kreisscheibe angenommen werden. Zwei Kreisscheiben auf Schnitt zu testen ist zwar an sich keine schwierige Aufgabe, jedoch wird häufig auf eine Vereinfachung zurückgegriffen. Dazu wird um die Kreisscheibe mit Radius r ein Quadrat mit den Kantenlängen r gelegt, das die Kreisscheibe vollständig einschließt. Statt nun die jeweiligen Kreisscheiben zweier Körper auf Schnitt zu testen, werden deren umgebenden Quadrate auf Schnitt getestet. Schneiden sich diese, so soll von Kollision der Körper gesprochen werden. Eine Möglichkeit Kollisionen aufzulösen ist Reflexion, wo nach der Körper sich im gleichen Winkel von dem gegnerischen Körper entfernt, wie der Winkel in dem er mit ihm zusammengetroffen ist. Die Bestimmung der Winkel ist jedoch manchmal schwierig, so dass auch eine andere deutlich einfachere Variante zur Anwendung kommen kann. Dabei werden die Körper einfach um die Länge, die sie sich durchdringen würden, entgegen ihrer ursprünglichen Bewegungsrichtung zurückgesetzt (**Rückstoß-Prinzip**).

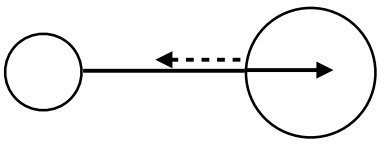


Abbildung 5: Kollision und Auflösung

Marcell Kehmstedt
m.kehmstedt@tu-bs.de

Gebietszerlegung

Nachdem nun alle Schritte für die Simulation der Körper geklärt sind, soll nocheinmal auf die Zerlegung des Simulationsgebiets sowie die parallele Berechnung eingegangen werden. Die Simulation erfolgt zu diskreten Zeitpunkten mit einer Zeitschrittweite δt. Jeder Körper bewegt sich dabei von Zeitpunkt zu Zeitpunkt. Sollte ein Körper dabei die Grenze eines Teilgebietes überschreiten, so muss dieses Ereignis festgestellt und gesondert behandelt werden. Tritt der Körper in ein benachbartes Gebiet ein, für das ein anderer Prozess zuständig ist, so muss der Körper dem anderen Prozess übergeben werden, damit dieser von nun an die Simulation des Körpers übernimmt. Was aber passiert wenn ein Körper eine Grenze überschreitet, an der kein anderes Gebiet angrenzt? So z.B. am ganz oberen oder unteren Rand. Eine Möglichkeit ist die Anwendung von Reflexion, so dass der Körper in das Gebiet zurückgeschleudert wird. Allerdings ist dies recht praxisfern, da das bekannte Universum keine Grenzen hat. Eine andere gute Möglichkeit ist es jeweils die Gebiete am oberen und unteren bzw. am linken und rechten Rand als benachbart aufzufassen. So dass ein Körper, der am unteren Rand sein Gebiet verlässt, am oberen Rand mit gleicher Geschwindigkeit in ein anderes Gebiet eintritt (**Turnaround**).

Algorithmus

Initialisierung $(t = 0)$**:**

a) Erzeuge n Himmelskörper (Sonnen, Planeten, Kometen, etc.).

b) Initialisiere jeweils Ort (2D), Geschwindigkeit (2D), Masse (1D), Radius (1D) und Typ zufällig.

c) Löse Kollisionen auf, so dass sich zu Beginn keine Körper gegenseitig durchdringen.

Simulationsschleife $(t = 1, 2, \dots)$**:**

a) Berechne die wirkende Gravitationskraft $\boldsymbol{F}_i^{(t-1)}$ für jeden Körper i basierend auf den Orten $\boldsymbol{x}^{(t-1)}$ zum Zeitpunkt $t-1$.

b) Aktualisiere Orte: $\boldsymbol{x}_i^{(t)} := \boldsymbol{x}_i^{(t-1)} + \delta t \cdot \boldsymbol{v}_i^{(t-1)}$

c) Aktualisiere Geschwindigkeiten: $\boldsymbol{v}_i^{(t)} := \boldsymbol{v}_i^{(t-1)} + \frac{\delta t}{m_i} \cdot \boldsymbol{F}_i^{(t-1)}$

d) Löse Kollisionen auf.

e) Korrigiere Gebietszuordnungen.

Aufgabe 1

- Implementieren Sie das Mehrkörperproblem und führen Sie Simulationen durch.
- Visualisieren Sie die Simulationsschritte und erstellen Sie ein 30 bis 40 sekündiges Video.
- Verwenden Sie ein Gebiet der Größe 512*512 sowie eine geeignete Menge von Körpern verschiedenen Typs. Die Typen sollen sich in Größe (Radius), Masse und Farbe der Darstellung unterscheiden.
- Behandeln Sie Kollisionen nach dem Rückstoß-Prinzip.
- Verwenden Sie eine Gebietsunterteilung mit mindestens 4 Teilgebiet.
- Wenden Sie Turnaround auf die äußersten Gebietsgrenzen an.
- Verwenden Sie abgeleitete Datentypen und gepackte Daten zum Übertragen der Körper.
- (Bonus) Implementieren Sie ein Load-Balancing mit dynamischer Gebietsanpassung.

(30+5 Punkte)