Informatik und Angewandte Kognitionswissenschaft
Lehrstuhl für Hochleistungsrechnen

UNIVERSITÄT DUISBURG ESSEN
Open-Minded

**Rainer Schlönvoigt**
**Thomas Fogal**
**Prof. Dr. Jens Krüger**

# High-Performance Computing

http://hpc.uni-duisburg-essen.de/teaching/wt2013/pp-nbody.html

## Exercise 2 - MPI N-Body (40 Points)

All assignments are to be uploaded to Moodle. Assignments are due at midnight on the due date. No late assignments will be accepted.

All assignments must include a `Makefile` for compiling your assignments. The assignment specification should include what the default target of your makefile should be. Assignments which do not compile will receive 0 points. Sometimes, we provide sample inputs and outputs; assignments which do not satisfy these test inputs will receive very few points.

Please do not include additional output other than what was requested by the assignment details. *Hint: if you want more debugging output, use a 'debug' flag in your program's arguments and have it only print when that flag is active.*

Your assignment will be graded on the `duecray.uni-due.de` supercomputer. It does not matter if your program runs correctly on another machine; it must run correctly on `duecray` to receive credit.

**Note**: components in blue are (mathematical) changes from the previous assignment!

# 1 Einführung

Eure Aufgabe in dieser Übung ist es zunächst, euren Code auf die in den folgenden Kapitel erwähnten änderungen anzupassen. Anschließend sollt ihr euren N-Body Code auf die Verwendung des Message Passing Interface umzuschreiben.

# 2 Anpassungen

Da es Probleme gab mit der letzten Übung, haben wir unseren Algorithmus etwas angepasst und vereinfacht. Dafür werden andere Konfigurationswerte und damit auch ein anderes Dateiformat benötigt. (Siehe 2.1)

Die anderen Anpassungen betreffen eine feste Größe des Zeitschrittes, anstelle einer adaptiven Berechnung (Siehe 2.3), sowie einer Anpassung der Beschleunigungsberechnung (Siehe 2.2).

## 2.1 Input File Format

```
5, 30000000, 30, 20,
 1,      1,      1,     20000000
 1000,   1000,   1000,  200000000
-1000,  -1000,  -1000,  200000000
-15000,  2000,   0,     900000000
 1500,  -1500,   0,     70000000
```

**Listing 1: Simple input file.**

Die Zeile der Konfigurationswerte besitzt nun nur noch 4 Werte und folgendes Format: (Punkt 3 und 4 sind neu)

1. Die Anzahl der Objekte.
2. Ein "end time criterion", an dem die Simulation stoppen soll.
3. Eine Konstante $\Delta t$ als feste Größe für den Zeitschrittl.
4. Eine Konstante $\epsilon$, die Instabilitäts-Erscheinungen unterdrücken soll.

Entsprechende Beispiel-Dateien werden euch zur Verfügung gestellt.

## 2.2 Calculate new acceleration

$$\vec{p_i} = G \sum_{j=0|j\neq i}^{N} \frac{m_j(p_j - p_i)}{(\|p_j - p_i\|^2 + \epsilon^2)^{\frac{3}{2}}} \tag{1}$$

Das neu hinzugekommende $\epsilon$ ersetzt den Mindestabstand der letzten Übung durch einen Dämpfungsfaktor. Dieser stellt sicher, dass wir nicht durch 0 dividieren und stammt aus der Eingabe-Datei. (Siehe 2.1)

### 2.3 Calculate the timestep

In der letzten Übung haben wir es euch überlassen, ob ihr einen festes $\Delta t$ verwendet oder $\Delta t$ mit jeder Iteration neu berechnet.

Die Resultate in beiden Ansätzen sind stark unterschiedlich, daher wir in dieser Übung die Verwendung eines festen $\Delta t$ verlangt, das in der Eingabe-Datei definiert ist. (Siehe 2.1)

$$t_{i+1} = t_i + \Delta t \tag{2}$$

## 3 Output Format

Innerhalb der letzten Übung wurde das Ausgabe-Dateiformat geändert, da die Version mit # am Anfang der ersten Zeile für Abstürze bei ParaView gesorgt hat. Da dies nicht jeder bemerkt hat, haben wir es hier nochmals aufgeführt. (Siehe Listing 2)

```
x, y, z, name1, name2, name3, ...
X0, Y0, Z0, scalar1, scalar2, scalar3, ...
X1, Y1, Z1, scalar1, scalar2, scalar3, ...
X2, Y2, Z2, scalar1, scalar2, scalar3, ...
X3, Y3, Z3, scalar1, scalar2, scalar3, ...
```

**Listing 2: CSV output file format**

$X0, Y0, Z0$ stehen hierbei für die Position der Partikel, $scalar1$, $scalar2$, usw. sind zusätzliche Skalarwerte. Diese könnt ihr bspw. für die Geschwindigkeit oder die Masse verwenden.

In den meisten Fällen solltet ihr dafür Gleitkommazahlen verwenden. Für ein konkretes Beispiel eines Ausgabeformates, siehe Listing 3. Die Datei beschreibt 4 Partikel, wobei jedes Partikel über 3 weitere Skalarwerte verfügt (Vermutlich die 3 Komponenten der Geschwindigkeit).

Um die Visualisierung zu vereinfach, macht es Sinn, die Datei nach dem zugehörigen Zeitschritt zu bennen, z.B.: "`164.csv`".

```
"x", "y", "z", "vx", "vy", "vz", "mass"
07.3713,   0.3218,   3.1043,  1.1833, 2.0134,   2.2291, 1000.0
10.2746,  26.4729,  70.5028,  8.0630, 8.4932,   6.0171, 1500.0
02.2911,  09.1276,  10.8019,  6.3234, 1.1237,  15.5061,  330.0
24.1717,  10.4758,  15.2072,  2.0334, 4.7838, 101.1451, 1394.0
```

**Listing 3: Sample CSV output file**

# 4 MPI (Wiederholung)

Wie ihr vermutlich in der ersten Übung bemerkt habt, nimmt der Berechnungsaufwand mit steigender Anzahl an Objekten stark zu. Dies lässt sich nicht verhindern, mit geeigneten Mitteln lässt sich die anfallende Rechenlast aber auf mehrere Kerne verteilen.

Diese Kerne können sich auf der selben Maschine oder auf im Netzwerk verteilten Maschinen befinden. MPI abstrahiert dies und nimmt dem Programmierer dadurch die Arbeit ab, zwischen den beiden Fällen im Code zu unterscheiden. Für den Code macht es daher keinen Unterschied auf wie vielen Maschinen er ausgeführt wird, sondern lediglich auf wie vielen Prozessen.

Nicht jeder Code bietet sich an für Parallelisierung. Code muss sich in Subtasks unterteilen lassen und der gewonnene Leistungszuwachs muss den neu hinzugekommenen zusätzlichen Kommunikationsaufwand wert sein. Falls der Code auf mehrere Computer verteilt wird, so ist jegliche Kommunikation sehr teuer.

### 4.0.1 Computer Cluster

Ein Cluster besteht aus mehreren verbundenen Computern und ermöglich auf diese Weise eine Parallelisierung in Ausmaßen, die für Heimcomputer unerreichbar erscheinen. Obwohl das Programmieren gegen die MPI Schnittstelle keine Kenntnisse der unterliegenden Hardware benötigt, so muss ein Cluster dennoch vorkonfiguriert werden. Dies wird euch aber von den verantwortlichen Admins abgenommen.

### 4.0.2 PBS

Im Gegensatz zu eurem Privat-Rechner wird ein Cluster von einer Vielzahl an Personen gleichzeitig verwendet. Daher kann nicht jeder Job sofort ausgeführt werden, sondern erst sobald die benötigte Anzahl an Ressourcen verfügbar ist und kein Job mit höherer Priorität vorliegt. Zur Ausführung eurer Anwendung wird daher ein PBS-Skript benötigt, das beinhaltet welche Ressourcen euer Programm benötigt und mit welchen Parametern euer Programm ausgeführt werden soll. Die Verwendung eines PBS Skriptes solltet ihr bereits gelernt haben.

Im Zweifelsfall wendet euch nochmals an Assignment 0.

# 5 Arbeitsaufteilung

For-Schleifen, in denen Berechnungen für jedes Objekt einzeln durchgeführt werden, ohne die anderen Objekte zu verändern, sind ein idealer Fall für Parallelisierung.

In unserem Fall lässt sich bspw. die aktuelle Beschleunigung jedes Objektes für mehrere Objekte gleichzeitig berechnen, ohne dass diese sich dabei gegenseitig zu stören.

## 5.1 Gerechte Verteilung

Ein typischer Anfängerfehler ist es, verschiedenen Prozessen Rollen wie Manager- und Arbeiter-Prozess zuzuschreiben. Obwohl dies durchaus möglich ist, benötigt es ständige Kommunikation zwischen den verschiedenen Prozessen, während der diese nicht Arbeiten können, da sie noch auf Antwort warten müssen.

Da sich die Anzahl der Objekte während der Iteration nie ändert, können wir stattdessen jedem Prozess einmalig zuschreiben, welche Elemente in seinem Aufgabengebiet liegen. Der selbe Prozess berechnet also die Beschleunigung der selben Objekte während jeder Iteration. Jeder Prozess kann seinen eigenen Arbeitsaufwand (anhand des MPI-Ranks und der Gesamtzahl an Objekten) sogar für sich selbst berechnen.

Der erste Teil der Umstellung auf MPI wird es also sein, Code zu verfassen, anhand dessen jeder Prozess berechnet für welche Objekte er verantwortlich ist. Ein Beispiel für eine solche Verteilung befindet sich in Listing 4.

```
1  There is a total of 34 objects!
2  Hi there, i am worker 0 and i have to work on 9 items starting with index 0 and ending with 8
3  Hi there, i am worker 1 and i have to work on 9 items starting with index 9 and ending with 17
4  Hi there, i am worker 2 and i have to work on 8 items starting with index 18 and ending with 25
5  Hi there, i am worker 3 and i have to work on 8 items starting with index 26 and ending with 33
```

**Listing 4: Verteilungs-Beispiel.**

Beachtet dabei, dass die Anzahl der zu bearbeitenden Elemente nicht unbedingt für jeden Prozess gleich hoch ist.

## 5.2 Synchronisieren der Reichweiten

Aus Gründen die in Kapitel 7.1 behandelt werden, ist es hilfreich, wenn jeder Prozess auch die Index-Reichweiten aller anderen Prozesse kennt. Be-

nutzt dafür die Funktion *MPI_Bcast()* und überprüft, ob dies vernünftig Funktioniert.

Hinweis: Die *MPI_Bcast()*-Funktion dient gleichzeitig zum Senden und Empfangen, wobei durch den Parameter *root* angegeben wird, wer die Quelle des Broadcasts sein soll. Daher muss der Broadcast auch so oft ausgeführt werden, wie es Prozesse gibt.

```
1  void MPI_Iterator::syncRanges() {
2    for (int i = 0; i < actuallyNeededProcesses; i++) {
3      MPI_Bcast(&startingIndexes[i], 1, MPI_INT, i, MPI_COMM_WORLD);
4      MPI_Bcast(&actualItemCounts[i], 1, MPI_INT, i, MPI_COMM_WORLD);
5    }
6  }
```

**Listing 5: sfasf**

# 6 Iteration

Die Berechnungen der Beschleunigungen, Geschwindigkeiten und der neuen Position, die ihr bisher auf jedem Objekt durchgeführt habt, muss jeder Prozess nun lediglich auf seiner eigenen Objekt-"Range" durchführen.

Dies entspricht einer Umstellung von Code wie in Listing 6 zu Code wie in Listing 7.

Zudem müssen die Prozesse nach jedem vollendeten Iterationsschritt ihre Resultate synchronisieren. (Erklärung siehe Kapitel 7)

```
1  void Serial_Iterator::calculateAccelerations() {
2
3    //calculate accelerations for workload
4    for (int i = 0; i <= objectCount; i++) {
5      _accelerations[i] = calculateAccelerationForElementWithIndex(i);
6    }
7  }
```

**Listing 6: Beschleunigungs-Schleife vorher**

```
1  void MPI_Iterator::calculateAccelerations() {
2
3    //calculate accelerations for workload
4    for (int i = startingIndex; i <= endingIndex; i++) {
5      _accelerations[i] = calculateAccelerationForElementWithIndex(i);
6    }
7  }
```

**Listing 7: Arbeitsbeschränkte Beschleunigungs-Schleife**

# 7 Abhängigkeiten behandeln

Am Ende jeder Iteration hat jeder Prozess lediglich die Position derjenigen Objekte aktualisiert, für die er verantwortlich ist. Im nächsten Iterationsschritt wollen wir jedoch wieder die Beschleunigung für jedes Objekt berechnen, wofür jeder Prozess wieder über die Position jedes anderen Objekt bescheid wissen muss.

Daher müssen wir nach jeder Iteration alle neuen Position über alle Prozesse hinweg synchronisieren.

## 7.1 Synchronisieren der Positionen

Die Synchronisation lässt sich über mehrere Wege realisieren;

1. Jeder Prozess schickt zunächst via *MPI_Send()* seine geänderten Positionen an einen zentralen Prozess, der alle neuen Positionen sammelt und anschlieend einen Broadcast aller Positionen an alle Prozesse durchführt,

2. oder jeder Prozess macht einen Broadcast seiner geänderten Positionen an alle anderen Prozesse.

Die Figur 1 zeigt beide Verhalten. Macht euch Gedanken über Vor- und Nachteile der beiden Ansätze.

# 8 *sine qua non*

Pack your file into a single compressed tarball for submission. The tarball should extract all files into a single subdirectory. Upload your tarball to Moodle.

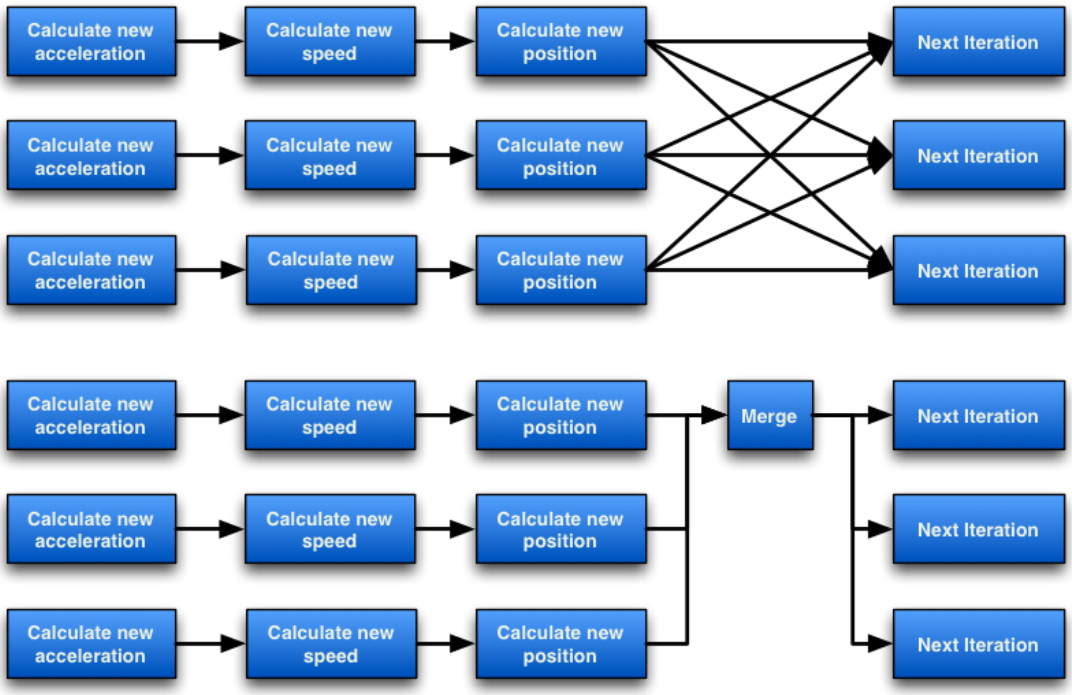


Figure 1: Broadcast-X *vs* Merge & Single-Broadcast