Institut
für Informatik

Scientific Computing

Ein Vortrag von

M. Jensen & T. Dziuk

Im Rahmen des Seminars

Nebenläufige und

Verteilte Programmierung

Überblick

- Motivation
- Grid Computations
- Particle Computations
- Fazit

Motivation

- Scientific Computing- was ist das?
  - Grid computing
    - „Grid" = Gitter ↯ Rechnerverbund
  - Particle Computing
  - Matrix Computing
  - ...
- Parallele Ausführung
  - wozu?
  - shared memory – im Folgenden: parallel
  - message passing – im Folgenden: distributed

Grid Computing

- Problemstellung:
  - Modell auf einem Kontinuum
  - Gesucht ist eine Funktion zur Berechnung des Modells
  - Laplace dient als Modell für z.B. Diffusion
- Laplace ist eine partielle Differentialgleichung
- Grid Computing für Approximation der Lösung von z.B. partiellen Differentialgleichungen
  - Jacobi Iteration
  - Gauß-Seidel

Laplace Gleichung

Geg.: Rechteck R

Ges.: Funktion $u : R \to \Re,$
$(x, y) \mapsto r$

Eine gutes Modell ist $\frac{\partial^2}{\partial^2 x} u + \frac{\partial^2}{\partial^2 y} u = 0$, die Laplace-Gleichung

Nicht lösbar $\longrightarrow$ Approximation

Teile das Rechteck in einzelne Punkte des Abstandes h ein:

$$R_h = \{(x, y) \mid x = i \cdot h, y = j \cdot h; i, j \in \mathbb{Z}\} \cap R$$

Geg.: R,h,R_h

Ges.: Gitterfunktion u_h

Laplace Gleichung

Approximation von Laplace: $\frac{\partial^2}{\partial^2 x} u(\bar{x}) = \frac{1}{h^2}[u(\bar{x}-h) - 2u(\bar{x}) + u(\bar{x}+h)]$

Mit $(\bar{x}, \bar{y}) \in R_h \setminus \partial R$ ergibt sich für Laplace als Approximation

$$0 = \frac{\partial^2}{\partial^2 x} u(\bar{x}, \bar{y}) + \frac{\partial^2}{\partial^2 y} u(\bar{x}, \bar{y}) = \frac{1}{h^2} \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} u(h)$$

und damit $u_h : R_h \to \Re$

mit

$$\frac{1}{h^2} \begin{bmatrix} -4 & 1 & 0 & \cdots & 1 \\ 1 & -4 & 1 & \cdots & 0 \\ \vdots & & \ddots & & \vdots \\ \vdots & & & -4 & \vdots \\ & & \cdots & & -4 \end{bmatrix} \begin{bmatrix} u \\ \\ \vdots \\ \\ u \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix}$$

Ist lösbar!

Iteration von Laplace

Aus der approximierten Laplace-Gleichung ergibt sich

$$0 = \frac{\partial^2}{\partial^2 x} u(\bar{x}, \bar{y}) + \frac{\partial^2}{\partial^2 y} u(\bar{x}, \bar{y}) = \frac{1}{h^2} \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} u(h)$$

$$\Leftrightarrow 4u(\bar{x}, \bar{y}) = u(\bar{x} - h, \bar{y}) + u(\overline{x + h}, \bar{y}) + u(\bar{x}, \overline{y - h}) + u(\bar{x}, \overline{y + h})$$

$$\Leftrightarrow u(\bar{x}, \bar{y}) = \frac{1}{4}\left(u(\bar{x} - h, \bar{y}) + u(\overline{x + h}, \bar{y}) + u(\bar{x}, \overline{y - h}) + u(\bar{x}, \overline{y + h})\right)$$

Definiere nun $grid[i, j] \cong u(ih, jh)$ mit $\bar{x} = i \cdot h, \bar{y} = j \cdot h$

$$\Rightarrow grid[i, j] = \frac{1}{4}\left(grid[i-1, j] + grid[i+1, j] + grid[i, j-1] + grid[i, j+1]\right)$$

Die sequentielle Jacobi Iteration

Institut für Informatik

```
While (true){
    for [i=1 to n, j=1 to n]
        new[i,j]=(grid[i-1,j]+grid[i+1,j]+grid[i,j-1]+grid[i,j+1])/4;
    iters++;

    maxdiff=0.0;
    for [i=1 to n, j=1 to n]
        maxdiff=max(maxdiff,abs(new[i,j]-grid[i,j]));
        if (maxdiff < EPSILON)
            break;
    for [i=1 to n, j=1 to n]
        grid[i,j] = new[i,j];
}
```

Optimierung (1)


```
While (true){
    for [i=1 to n, j=1 to n]
        new[i,j]=(grid[i-1,j]+grid[i+1,j]+grid[i,j-1]+grid[i,j+1])*0.25;
    iters++;

    maxdiff=0.0;
    for [i=1 to n, j=1 to n]
        maxdiff=max(maxdiff,abs(new[i,j]-grid[i,j]));
        if (maxdiff < EPSILON)
            break;
    for [i=1 to n, j=1 to n]
        grid[i,j] = new[i,j];
}
```

Optimierung (2)


```
for (iters=1 to MAXITERS){
    for [i=1 to n, j=1 to n]
        new[i,j]=(grid[i-1,j]+grid[i+1,j]+grid[i,j-1]+grid[i,j+1])*0.25;

    for [i=1 to n, j=1 to n]
        grid[i,j] = new[i,j];

    maxdiff=0.0;
    for [i=1 to n, j=1 to n]
        maxdiff=max(maxdiff,abs(new[i,j]-grid[i,j]));
}
```

Optimierung (3)


```
real grid[0:1][0:n+1][0:n+1]
int old=0, int new=1;

for (iters=1 to MAXITERS){
    for [i=1 to n, j=1 to n]
        grid[new][i,j]=(grid[old][i-1,j]+grid [old][ i+1,j]+grid [old][ i,j-1]
                                        +grid [old][ i,j+1])*0.25;
old=1-old; new=1-new;
}
```

Optimierung (4)


```
for (iters=1 to MAXITERS by 2){
    for [i=1 to n, j=1 to n]
        grid[new][i,j]=(grid[old][i-1,j]+grid [old][ i+1,j]+grid [old][ i,j-1]
                                        +grid [old][ i,j+1])*0.25;
    old=1-old; new=1-new;

    for [i=1 to n, j=1 to n]
        grid[new][i,j]=(grid[old][i-1,j]+grid [old][ i+1,j]+grid [old][ i,j-1]
                                        +grid [old][ i,j+1])*0.25;
    old=1-old; new=1-new;
}
```

Optimierung (5)


```
for (iters=1 to MAXITERS by 2){
    for [i=1 to n, j=1 to n]
        new][i,j]=(grid[i-1,j]+grid[ i+1,j]+grid[ i,j-1]+grid[ i,j+1])*0.25;

    for [i=1 to n, j=1 to n]
        grid[i,j]=(new[i-1,j]+new[ i+1,j]+new[ i,j-1]+new[ i,j+1])*0.25;

}
```

Jacobi parallel

Zu lösen: $Ax=b$

da $A=D-U-L$, ist $(D-U-L)x=b$ zu lösen.

$\Leftrightarrow Dx_{neu} = (L+U)x_{alt}+b$

Dx_{neu} beeinflusst L und U nicht, d.h. die Berechung von D kann in beliebiger Reihenfolge vorgenommen werden.

$\Rightarrow$ Gut parallelisierbar

Die Jacobi-Iteration ist ein Gesamtschritt-verfahren

Jacobi parallel: Shared Variables


```
Int HEIGTH= n/PR;

Process worker[w=1 to PR]
    int firstrow = (w-1)*HEIGHT +1;
    int lastrow = firstrow + HEIGHT –1;
    <barrier>
    for(iters=1 to MAXITERS by 2){
            for(i=firstrow to lastrow, j=1 to n)
                    new[i,j]=grid ...
            <barrier>
            for(i=firstrow to lastrow, j=1 to n)
                    grid[i,j]=new ...
            <barrier>
    }
    for(i=firstrow to lastrow, j=1 to n){
                    mydiff...
            maxdiff(w) = mydiff;
            <barrier>
    }
```

Jacobi parallel: Message Passing


```
chan up[1:PR] (real edge[0:n+1]);
chan down[1:PR] (real edge[0:n+1);
chan diff (real);

Process worker[w=1 to PR]
int HEIGHT = n/PR;
*/initialisiere grid und new
for(iters=1 to MAXITERS by 2){
        for(i=1 to HEIGHT, j=1 to n)
                new[i,j]=grid ...
        */exchange edges
        for(i=2 to HEIGHT-1, j=1 to n)
                grid[i,j]=new ...
        */exchange edges
}
*/worker 1 berechnet maxdiff
```

```
if(w>1)
   send up[w-1](new[1,*]);
if(w<PR)
   send down[w+1](new[HEIGHT,*]);
if(w<PR)
   receive up[w] (new[HEIGHT+1,*]);
If(w>1)
   receive down[w] (new[0,*]);
```

Gauß-Seidel

Zu lösen: $Ax=b$

da $A=D-U-L$, ist $(D-U-L)x=b$ zu lösen.

$\Leftrightarrow (D-L)x_{neu} = Ux_{alt}+b$

$(D-L)x_{neu}$ beeinflusst L und ist deshalb nicht so einfach zu parallelisieren.

Gauß-Seidel ist ein Einzelschrittverfahren.

Ausblick :

- Adaptive Grid
  - Einteilungsfeinheit variiert
- Aufteilung des Problemes in mehrere Grids
  - Mehrere Grids verschiedener Feinheit

Institut
für Informatik

Scientific Computing
(Teil 2: particle computing)

Ein Vortrag von

M. Jensen und T. Dziuk

Im Rahmen des Seminars

Nebenläufige und

Verteilte Programmierung

Überblick

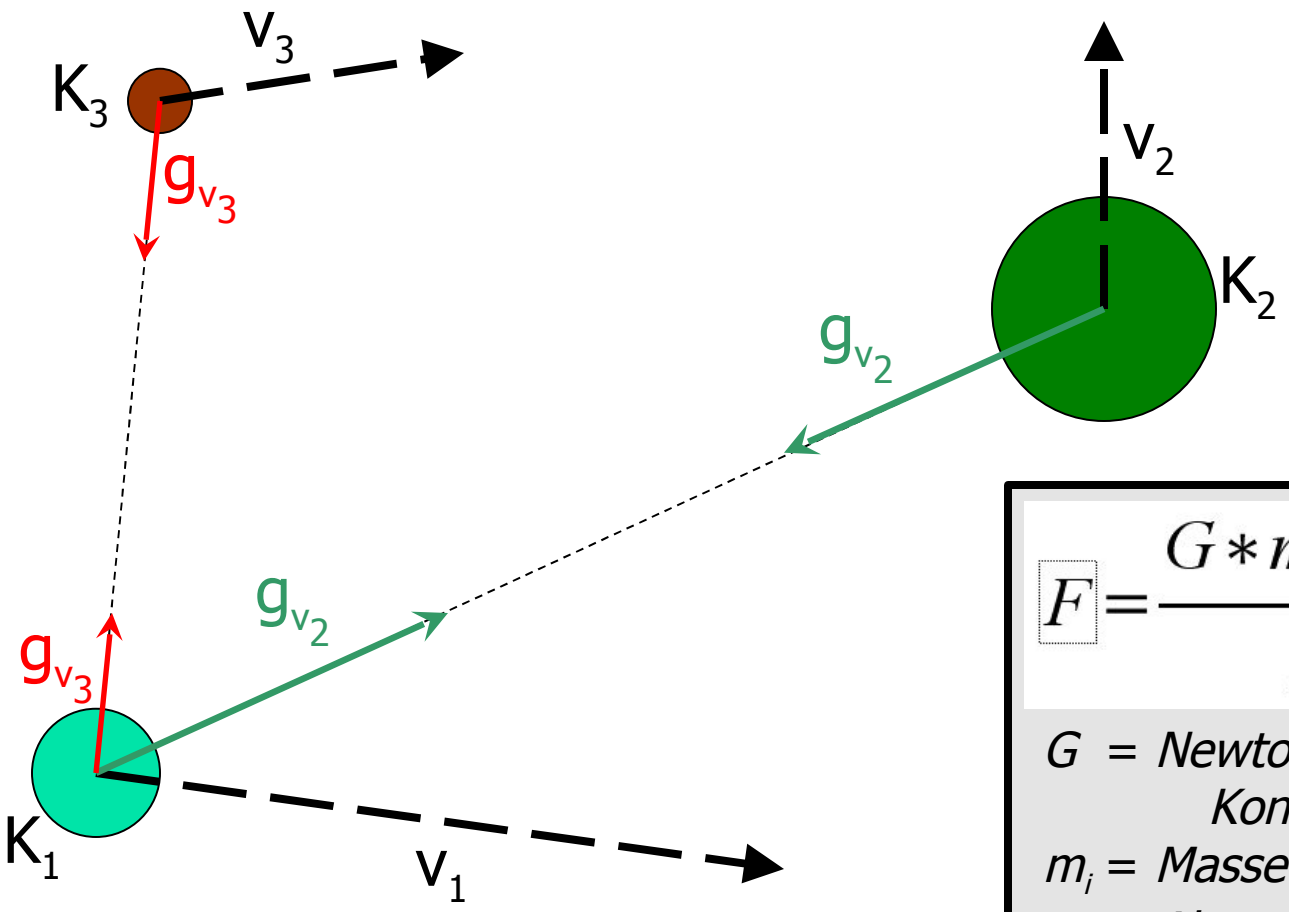
- Das N-Körper-Problem
  - Lösungsalgorithmus
  - Sequentielle Implementierung
  - Parallele Implementierung
  - Verteilte Implementierung
  - Barnes-Hut-Approximation
- Fazit

Das N-Körper-Problem

- Gegeben: N Körper (Partikel),
- diese wirken aufeinander ein...
- ...und bewegen/verändern sich...
- ...in Abhängigkeit von den Einwirkungen aller anderen Körper.

➔ Die Veränderung eines Körpers hängt vom aktuellen Zustand aller N-1 anderen Körper ab!

Das N-Körper-Problem



$$F = \frac{G * m_1 * m_2}{r^2}$$

G = *Newton-Konstante*
m_i = *Masse*
r = *Abstand*

Lösungsalgorithmus

Zeit und Position diskretisieren!

- Unterteile Betrachtungszeitraum in einzelne Zeitschritte
- Für jeden Zeitschritt:
  - Berechne den Endzustand aus dem Anfangszustand
  - Für jeden Körper einzeln:
    1. Berechne die Einflüsse aller N-1 anderen Körper
    2. Berechne Endzustands-Position aus
       - Anfangsposition
       - Anfangsgeschwindigkeit
       - Summe der Einflüsse anderer Körper

Sequentielle Implementierung


```
foreach delta_t ( delta_t ∈ <<Betrachtungszeitraum>>) {
    // calculate new movement vector
    foreach K[i] (i ∈ 1..N) {
        foreach K[ j] ( j ∈ 1..N, j !=i ) {
            f[i,j] = calculate_forces(K[i], K[ j]);
        }
    }
    // move particles
    foreach K[i] (i ∈ 1..N) {
        v[i] = <<Summe der f[i,j], j ∈ 1..N>>
        move_particle(K[i], v[i]);
    }
}
```

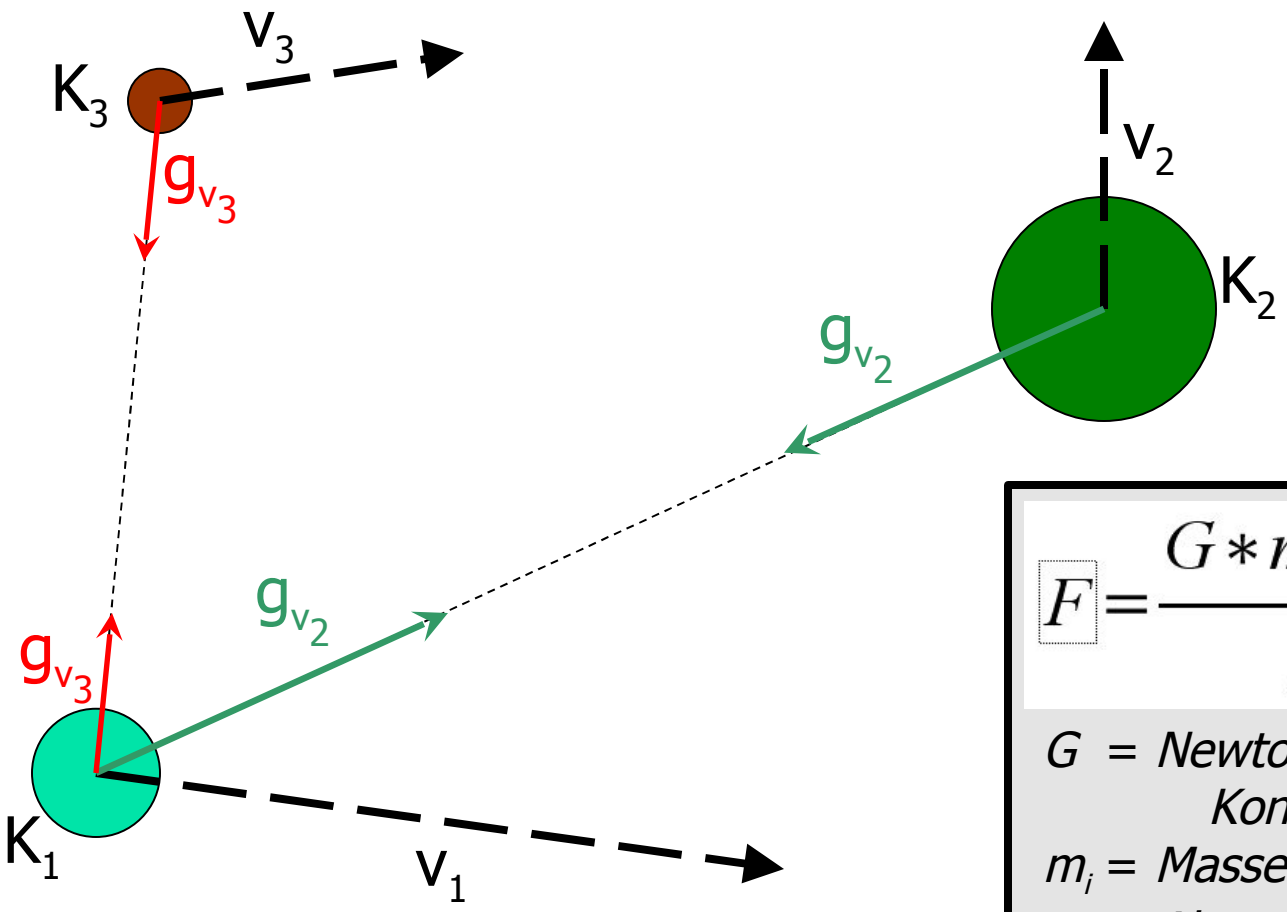
K[i] = Körper
v[i] = Geschwindigkeiten
f[i,j] = Kräftewirkung zwischen den Körpern

Sequentielle Implementierung

Institut für Informatik

	1	2	3	4	5	6	7	8	9	...	N
1	0	f[1,2]	f[1,3]	f[1,4]	f[1,5]	f[1,6]	f[1,7]	f[1,8]	f[1,9]	...	f[1,N]
2	f[2,1]	0	f[2,3]	f[2,4]	f[2,5]	f[2,6]	f[2,7]	f[2,8]	f[2,9]	...	f[2,N]
3	f[3,1]	f[3,2]	0	f[3,4]	f[3,5]	f[3,6]	f[3,7]	f[3,8]	f[3,9]	...	f[3,N]
4	f[4,1]	f[4,2]	f[4,3]	0	f[4,5]	f[4,6]	f[4,7]	f[4,8]	f[4,9]	...	f[4,N]
5	f[5,1]	f[5,2]	f[5,3]	f[5,4]	0	f[5,6]	f[5,7]	f[5,8]	f[5,9]	...	f[5,N]
6	f[6,1]	f[6,2]	f[6,3]	f[6,4]	f[6,5]	0	f[6,7]	f[6,8]	f[6,9]	...	f[6,N]
7	f[7,1]	f[7,2]	f[7,3]	f[7,4]	f[7,5]	f[7,6]	0	f[7,8]	f[7,9]	...	f[7,N]
8	f[8,1]	f[8,2]	f[8,3]	f[8,4]	f[8,5]	f[8,6]	f[8,7]	0	f[8,9]	...	f[8,N]
9	f[9,1]	f[9,2]	f[9,3]	f[9,4]	f[9,5]	f[9,6]	f[9,7]	f[9,8]	0	...	f[9,N]
...	...	...	...	...	...	...	...	...	...	...	...
N	f[N,1]	f[N,2]	f[N,3]	f[N,4]	f[N,5]	f[N,6]	f[N,7]	f[N,8]	f[N,9]	...	0

Sequentielle Implementierung



$$F = \frac{G * m_1 * m_2}{r^2}$$

G = *Newton-Konstante*
m_i = *Masse*
r = *Abstand*

Sequentielle Implementierung

Institut für Informatik

	1	2	3	4	5	6	7	8	9	...	N
1	0	f[1,2]	f[1,3]	f[1,4]	f[1,5]	f[1,6]	f[1,7]	f[1,8]	f[1,9]	...	f[1,N]
2	f[2,1]	0	f[2,3]	f[2,4]	f[2,5]	f[2,6]	f[2,7]	f[2,8]	f[2,9]	...	f[2,N]
3	f[3,1]	f[3,2]	0	f[3,4]	f[3,5]	f[3,6]	f[3,7]	f[3,8]	f[3,9]	...	f[3,N]
4	f[4,1]	f[4,2]	f[4,3]	0	f[4,5]	f[4,6]	f[4,7]	f[4,8]	f[4,9]	...	f[4,N]
5	f[5,1]	f[5,2]	f[5,3]	f[5,4]	0	f[5,6]	f[5,7]	f[5,8]	f[5,9]	...	f[5,N]
6	f[6,1]	f[6,2]	f[6,3]	f[6,4]	f[6,5]	0	f[6,7]	f[6,8]	f[6,9]	...	f[6,N]
7	f[7,1]	f[7,2]	f[7,3]	f[7,4]	f[7,5]	f[7,6]	0	f[7,8]	f[7,9]	...	f[7,N]
8	f[8,1]	f[8,2]	f[8,3]	f[8,4]	f[8,5]	f[8,6]	f[8,7]	0	f[8,9]	...	f[8,N]
9	f[9,1]	f[9,2]	f[9,3]	f[9,4]	f[9,5]	f[9,6]	f[9,7]	f[9,8]	0	...	f[9,N]
...	...	...	...	...	...	...	...	...	...	...	...
N	f[N,1]	f[N,2]	f[N,3]	f[N,4]	f[N,5]	f[N,6]	f[N,7]	f[N,8]	f[N,9]	...	0

Sequentielle Implementierung

Institut für Informatik

	2	3	4	5	6	7	8	9	...	N
1	f[1,2]	f[1,3]	f[1,4]	f[1,5]	f[1,6]	f[1,7]	f[1,8]	f[1,9]	...	f[1,N]
2		f[2,3]	f[2,4]	f[2,5]	f[2,6]	f[2,7]	f[2,8]	f[2,9]	...	f[2,N]
3			f[3,4]	f[3,5]	f[3,6]	f[3,7]	f[3,8]	f[3,9]	...	f[3,N]
4				f[4,5]	f[4,6]	f[4,7]	f[4,8]	f[4,9]	...	f[4,N]
5					f[5,6]	f[5,7]	f[5,8]	f[5,9]	...	f[5,N]
6						f[6,7]	f[6,8]	f[6,9]	...	f[6,N]
7							f[7,8]	f[7,9]	...	f[7,N]
8								f[8,9]	...	f[8,N]
...									...	f[9,N]

Sequentielle Implementierung


```
foreach delta[t] ( delta[t] ∈ {Zeitintervalle}) {
    // calculate new movement vector
    foreach K[i] (i ∈ 1..N) {
        foreach K[ j] ( j ∈ 1..N, j !=i ) {
            if (i < j) {
                f[i,j] = calculate_forces(K[i], K[ j]);
            } else {
                f[i,j] = f[ j,i];
            }
        }
    }
    // move particles
    ...
```

Parallele Implementierung

- n Prozessoren/Prozesse, N Körper/Partikel
- Ein gemeinsamer Speicher

➔ arbeiten auf der gleichen Speichermatrix!

- Berechnung der Kräfte parallel,
- danach Berechnung der Bewegungen parallel

➔ Dürfen sich gegenseitig nicht ins Gehege kommen!

Idee: Kräftetabelle auf Prozesse aufteilen

Parallele Implementierung

Beispiel: 4 Prozesse: P1, P2, P3, P4, mit Blockverfahren

	2	3	4	5	6	7	8	9	...	N
1	f[1,2]	f[1,3]	f[1,4]	f[1,5]	f[1,6]	f[1,7]	f[1,8]	f[1,9]	...	f[1,N]
2		f[2,3]	f[2,4]	f[2,5]	f[2,6]	f[2,7]	f[2,8]	f[2,9]	...	f[2,N]
3			f[3,4]	f[3,5]	f[3,6]	f[3,7]	f[3,8]	f[3,9]	...	f[3,N]
4				f[4,5]	f[4,6]	f[4,7]	f[4,8]	f[4,9]	...	f[4,N]
5					f[5,6]	f[5,7]	f[5,8]	f[5,9]	...	f[5,N]
6						f[6,7]	f[6,8]	f[6,9]	...	f[6,N]
7							f[7,8]	f[7,9]	...	f[7,N]
8								f[8,9]	...	f[8,N]
...									...	f[9,N]

P1: 15 f

P2: 11 f

P3: 7 f

P4: 3 f

Parallele Implementierung

Institut für Informatik

4 Prozesse: P1, P2, P3, P4, mit "Striping"-Verfahren

	2	3	4	5	6	7	8	9	...	N
1	f[1,2]	f[1,3]	f[1,4]	f[1,5]	f[1,6]	f[1,7]	f[1,8]	f[1,9]	...	f[1,N]
2		f[2,3]	f[2,4]	f[2,5]	f[2,6]	f[2,7]	f[2,8]	f[2,9]	...	f[2,N]
3			f[3,4]	f[3,5]	f[3,6]	f[3,7]	f[3,8]	f[3,9]	...	f[3,N]
4				f[4,5]	f[4,6]	f[4,7]	f[4,8]	f[4,9]	...	f[4,N]
5					f[5,6]	f[5,7]	f[5,8]	f[5,9]	...	f[5,N]
6						f[6,7]	f[6,8]	f[6,9]	...	f[6,N]
7							f[7,8]	f[7,9]	...	f[7,N]
8								f[8,9]	...	f[8,N]
...									...	f[9,N]

P1: 12 f

P2: 10 f

P3: 8 f

P4: 6 f

Parallele Implementierung

Institut für Informatik

4 Prozesse: P1, P2, P3, P4, mit "Reverse Striping"

	2	3	4	5	6	7	8	9	...	N
1	f[1,2]	f[1,3]	f[1,4]	f[1,5]	f[1,6]	f[1,7]	f[1,8]	f[1,9]	...	f[1,N]
2		f[2,3]	f[2,4]	f[2,5]	f[2,6]	f[2,7]	f[2,8]	f[2,9]	...	f[2,N]
3			f[3,4]	f[3,5]	f[3,6]	f[3,7]	f[3,8]	f[3,9]	...	f[3,N]
4				f[4,5]	f[4,6]	f[4,7]	f[4,8]	f[4,9]	...	f[4,N]
5					f[5,6]	f[5,7]	f[5,8]	f[5,9]	...	f[5,N]
6						f[6,7]	f[6,8]	f[6,9]	...	f[6,N]
7							f[7,8]	f[7,9]	...	f[7,N]
8								f[8,9]	...	f[8,N]
...									...	f[9,N]

P1: 9 f

P2: 9 f

P3: 9 f

P4: 9 f

Parallele Implementierung

P_k:

```
foreach delta[t] ( delta[t] ∈ {Zeitintervalle}) {
    // calculate new movement vector
    for( i = k to N step (N/n) ) {          for( j = i+1 to N ) {
                    f[i,j] = f[ j,i] = calculate_forces(K[i], K[ j]);
    }}
    <<barrier for all n processes>>
    // move particles
    for( i = 1 to N ) {
        v[i] = Summe der f[i,j], j ∈ 1..N
        move_particle(K[i], v[i]);
    }
    <<barrier for all n processes>>
}
```

Verteilte Implementierung

- n Prozessoren/Prozesse, N Körper/Partikel
- n verschiedene Speicher
- Kommunikation durch Message-passing

➔Verschiedene Paradigmen:

- Manager / Workers
- Heartbeat - Algorithmus
- Pipeline - Algorithmus

Verteilte Implementierung

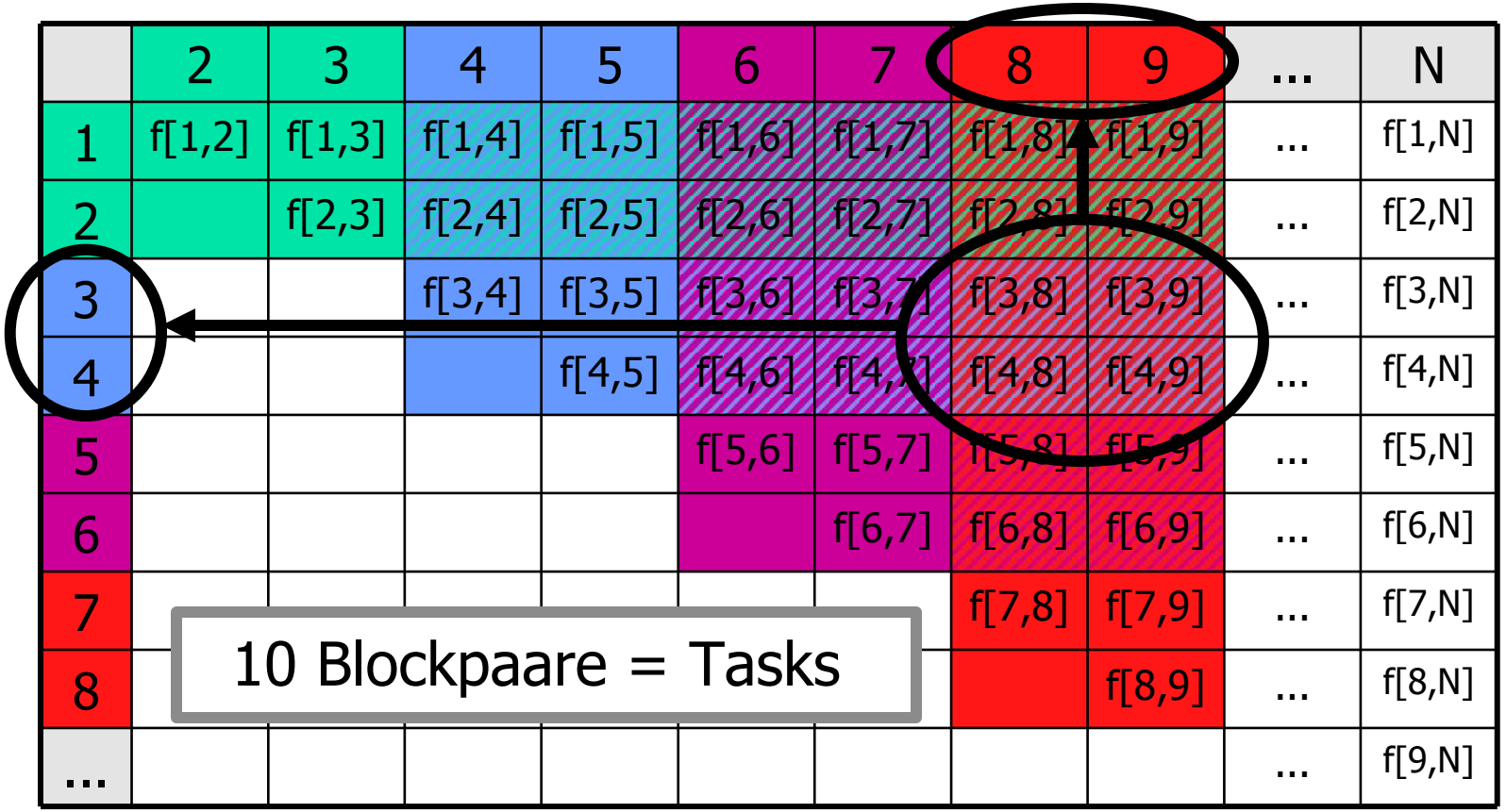
Manager / Workers:

- n+1 Prozesse, N Körper
- Ein Prozess als *"Manager"*, n als *"Workers".*
- Der Manager verteilt die Zuständigkeiten,
- die Workers berechnen die Kräfte,
- der Manager sammelt die Ergebnisse...
- ...und verteilt sie dann (Broadcast),
- die Workers berechnen die neuen Körper-Positionen und Geschwindigkeiten,
- der Manager sammelt die neuen Positionen und Geschwindigkeiten und verteilt sie dann.

Verteilte Implementierung

Institut für Informatik

4 Blöcke: B1, B2, B3, B4, mit Manager / Workers



Verteilte Implementierung

Manager / Workers:

- Manager verwaltet "Bag of Tasks"
- Worker holen sich Tasks zur Bearbeitung ab, bis keine mehr da sind.
- Körperbewegungs-Phase wird verteilt:

  Worker i bewegt alle Körper in Block i
- Danach Broadcast der neuen Positionen

Optimierungen:

- Jeder Worker erhält zu Anfang gesamte Positionsmatrix => Task-Nachrichten werden kleiner
- Statt zum Manager sendet jeder Worker berechnete Kräfte zu dem Worker, der die zugehörigen Körper später bewegen soll.

Verteilte Implementierung

Heartbeat – Algorithmus:

Grundidee: Für die Berechnung der neuen Position von Körper i:

1. erfrage Kräftewirkung der Körper 1 bis i-1,
2. erfrage Positionen und Massen der Körper i+1 bis N,
3. berechne Kräftewirkung für Körper i,
4. sende Kräftewirkung von Körper i auf Körper i+1 bis N an Diese.

➔ andere Werte sind nicht erforderlich!

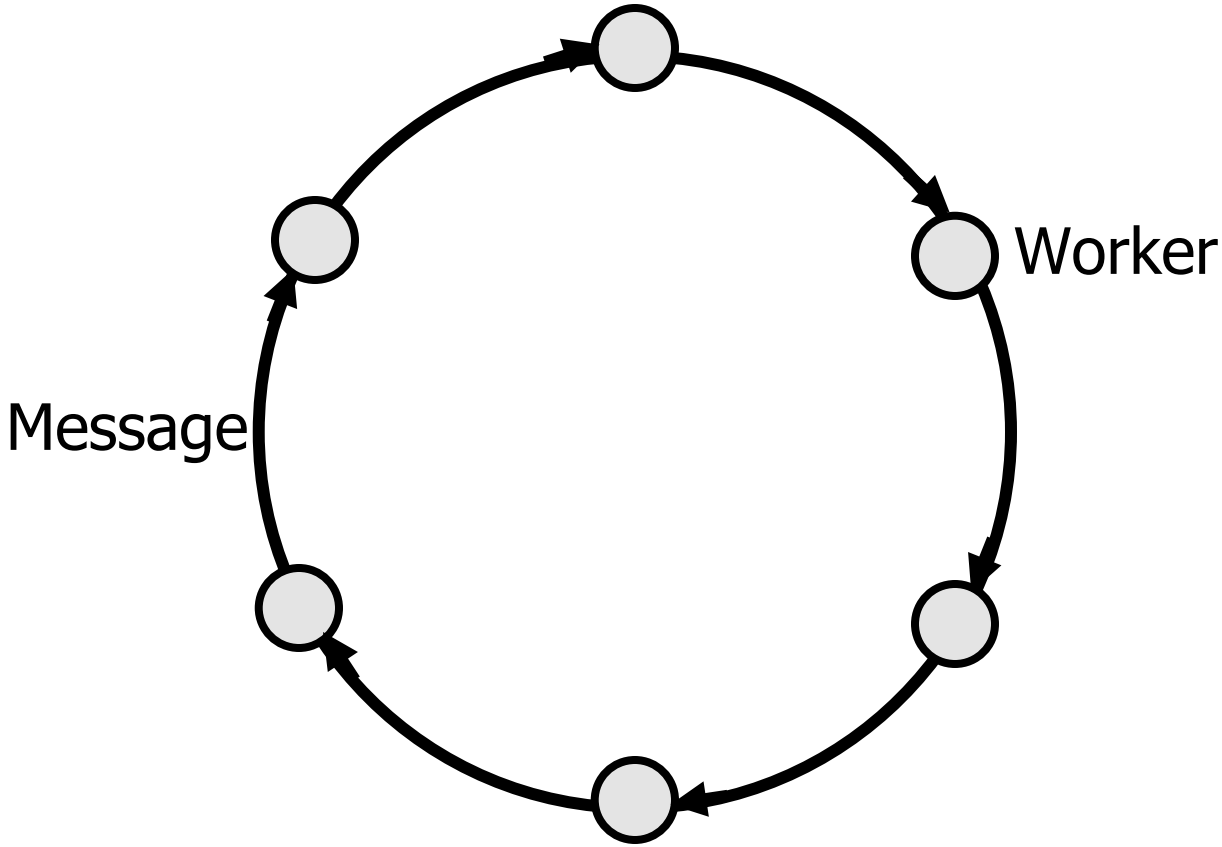
Verteilte Implementierung

Heartbeat – Algorithmus:

- Zuordnung von Körpern auf Prozesse (blockweise, "striping", "reverse striping", ...)
- Jeder Prozess ...
  - ... sendet Positionen und Geschwindigkeiten nach unten,
  - empfängt Positionen und Geschwindigkeiten von oben,
  - empfängt Kräftewirkungen von unten
  - berechnet eigene Kräftewirkungen
  - sendet Kräftewirkungen nach oben
  - bewegt seine Körper

Verteilte Implementierung

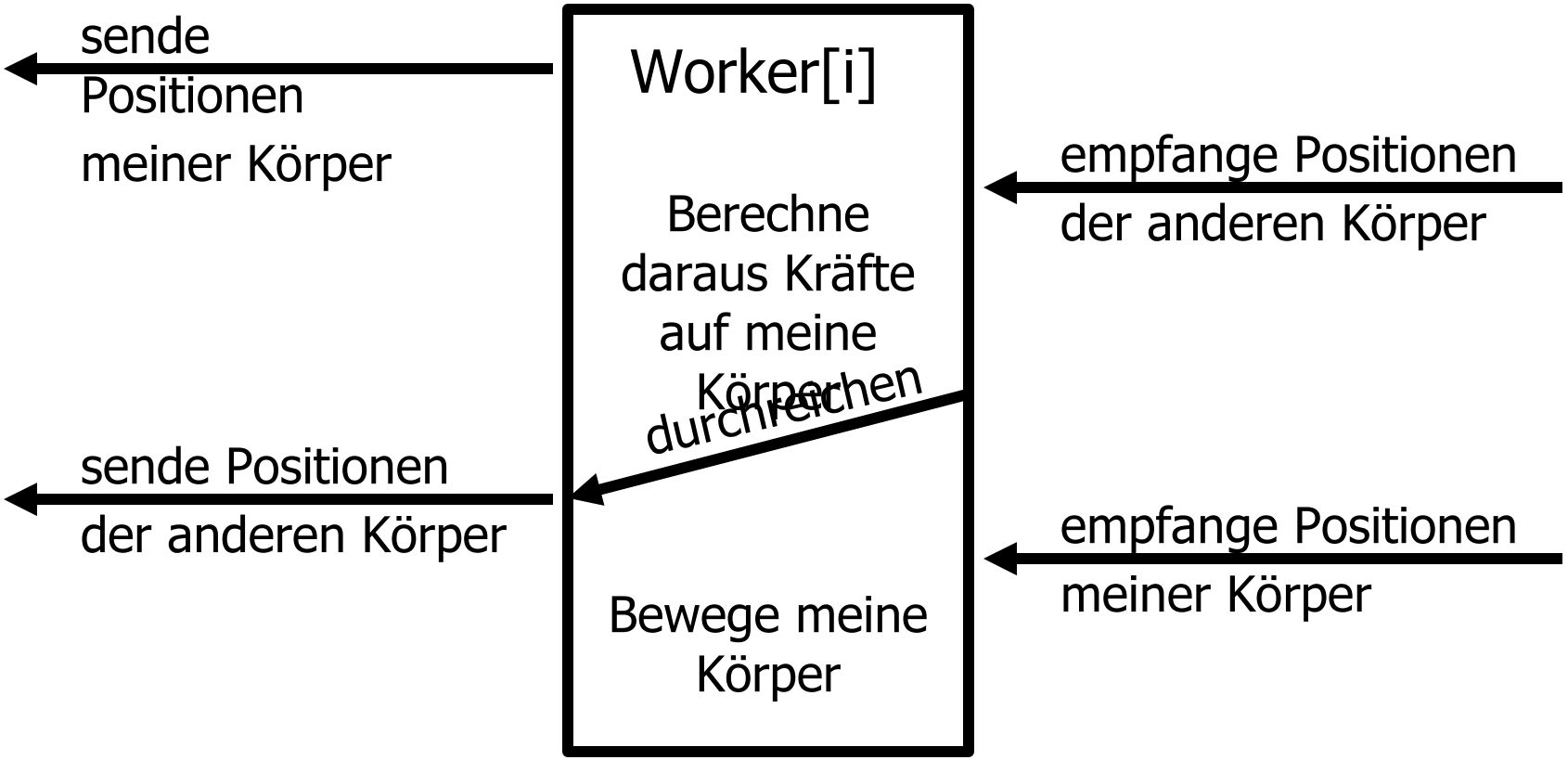
Pipeline – Algorithmus:



Verteilte Implementierung

Institut für Informatik

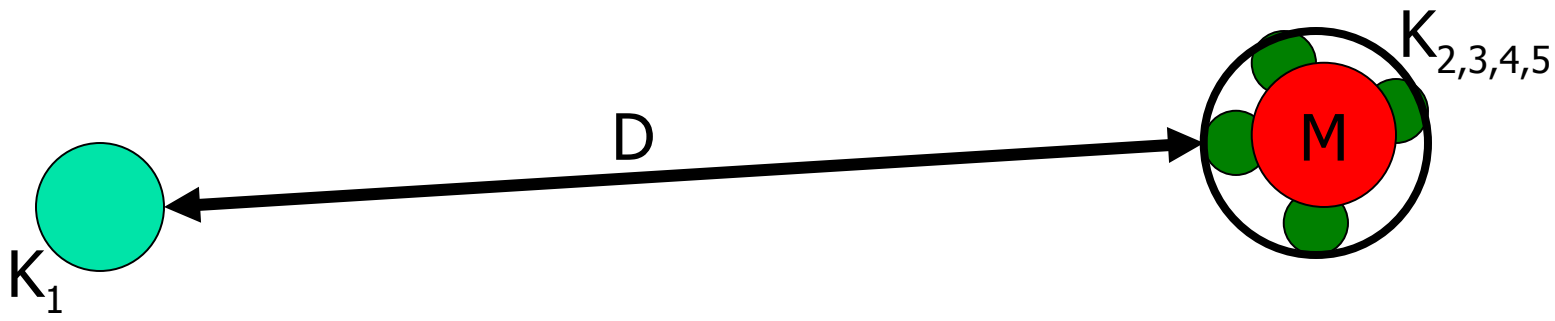
Pipeline – Algorithmus:



Optimierung des Algorithmus

Barnes-Hut-Approximation

Barnes-Hut-Approximation



$D > X \Rightarrow f[1,2]+f[1,3]+f[1,4]+f[1,5] \approx f[1,Ø2345]$

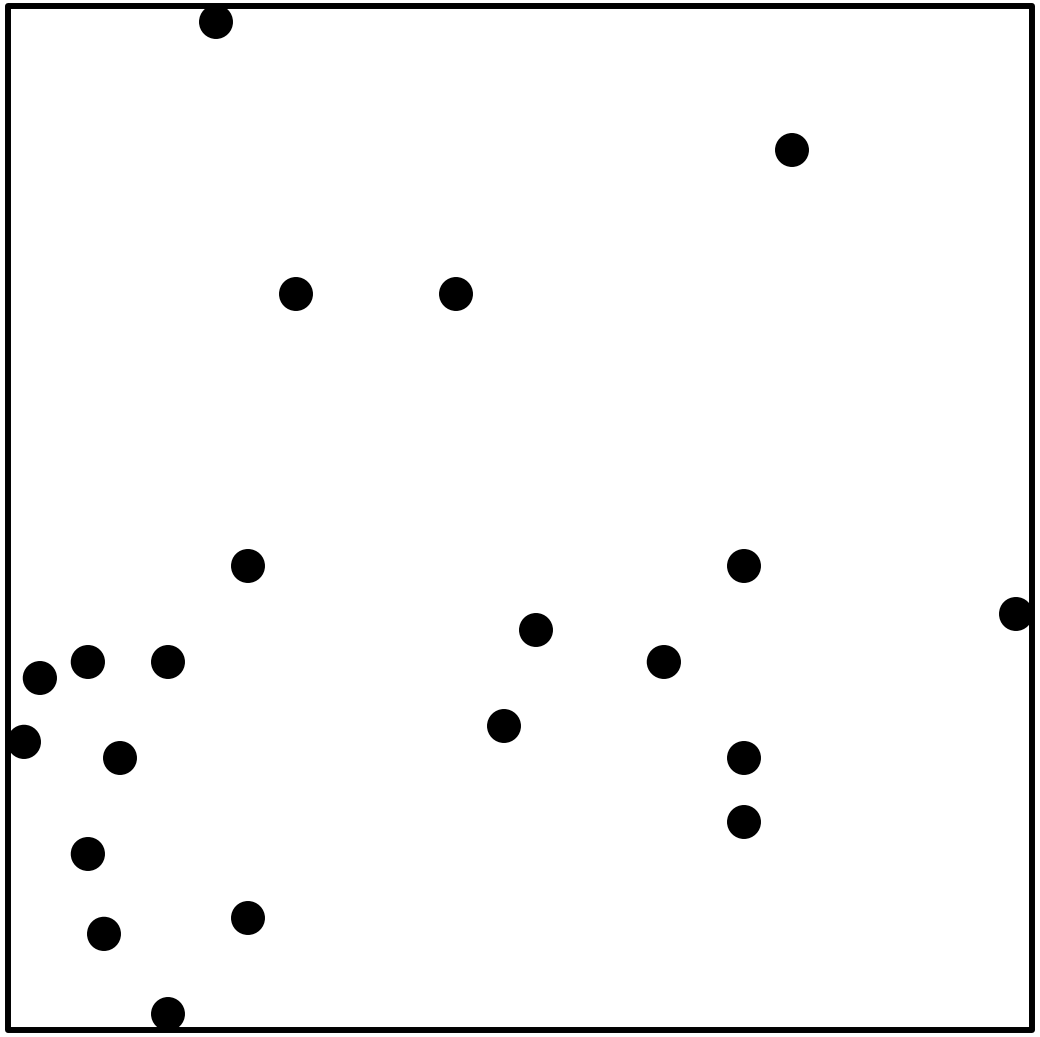
➔ Berechne neuen Partikel M mit:

$mass(M) := mass(K_2)+mass(K_3)+mass(K_4)+mass(K_5)$

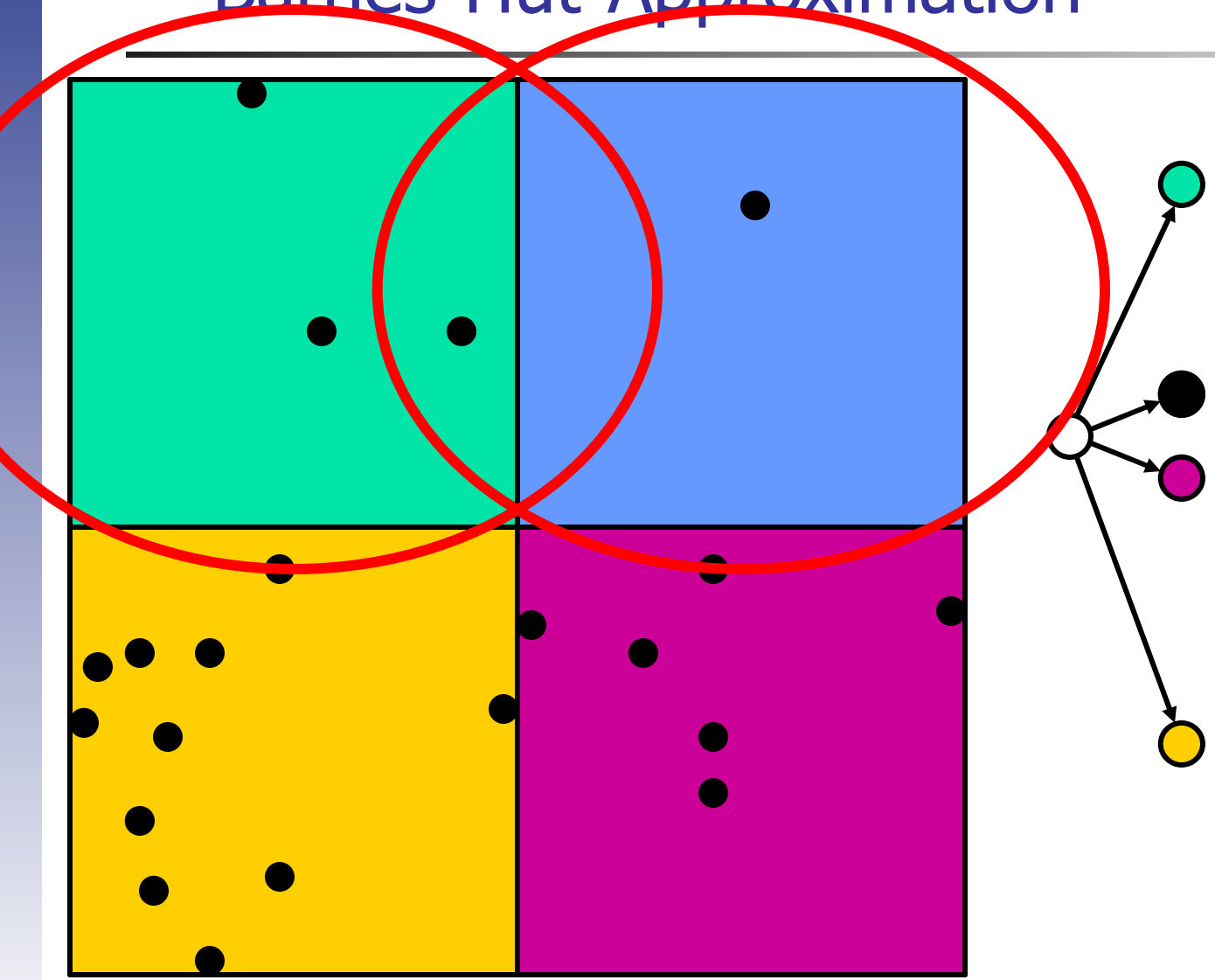
$pos(M) := (pos(K_2)+pos(K_3)+pos(K_4)+pos(K_5)) / 4$

und berechne statt f[1,2], f[1,3], f[1,4] und f[1,5] nur f[1,M]

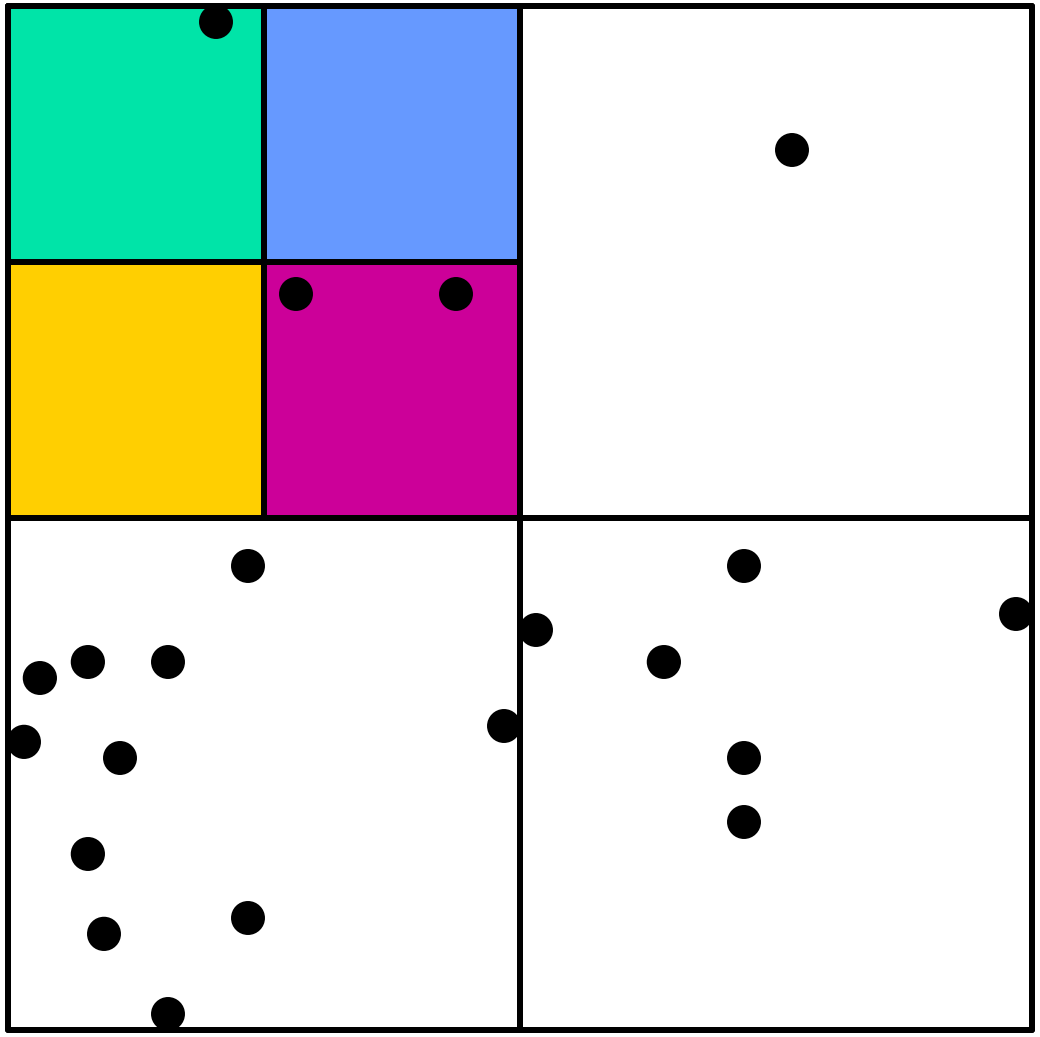
Barnes-Hut-Approximation

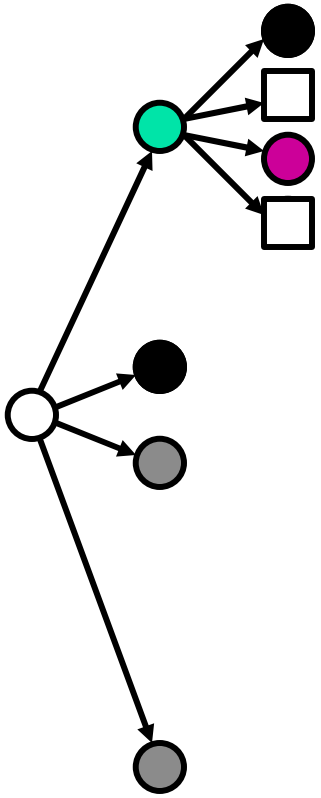


Barnes-Hut-Approximation

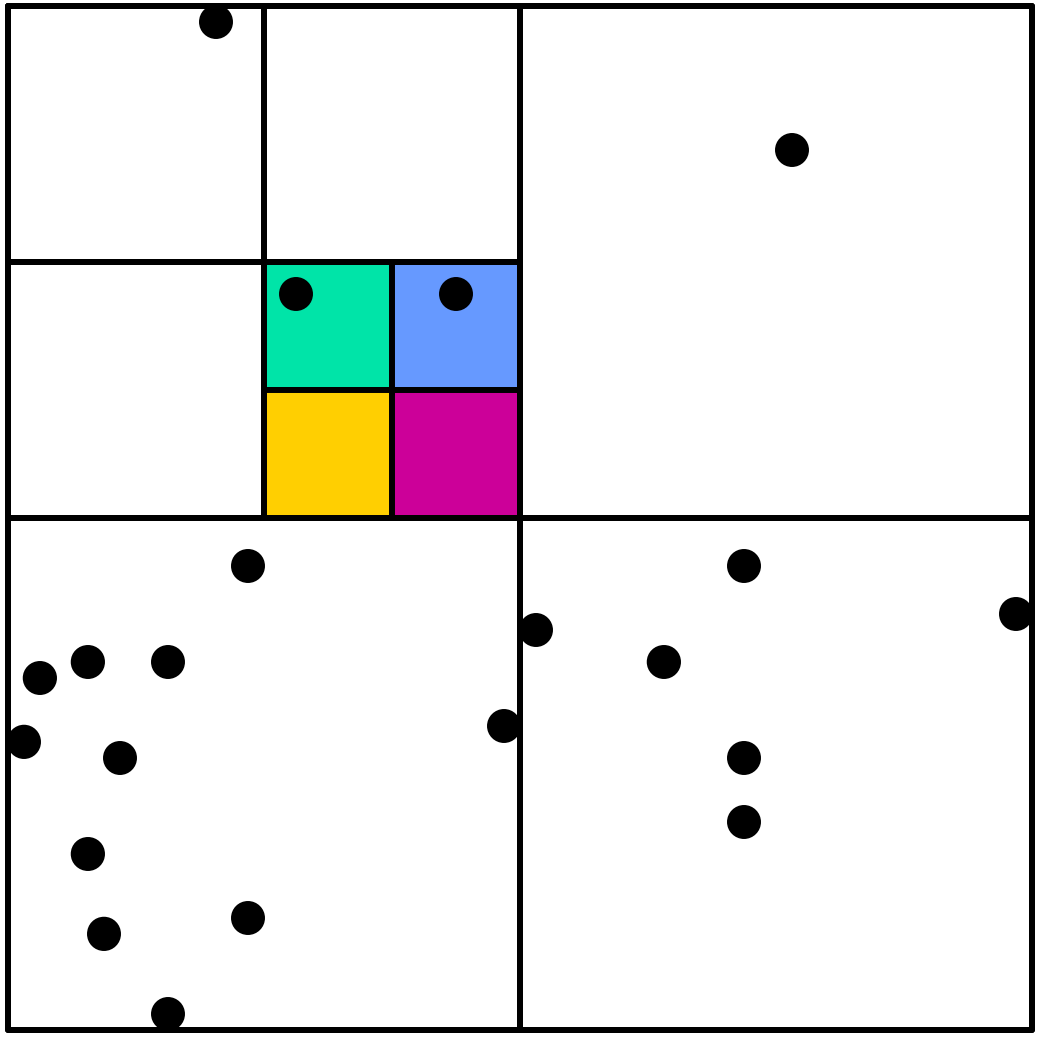


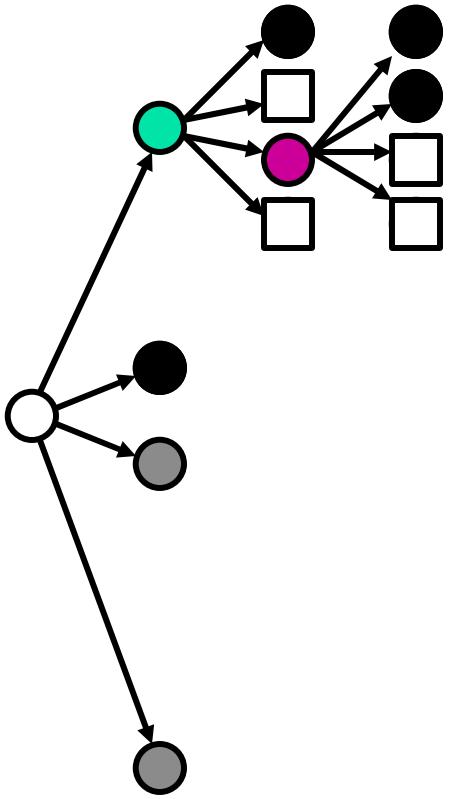
Barnes-Hut-Approximation



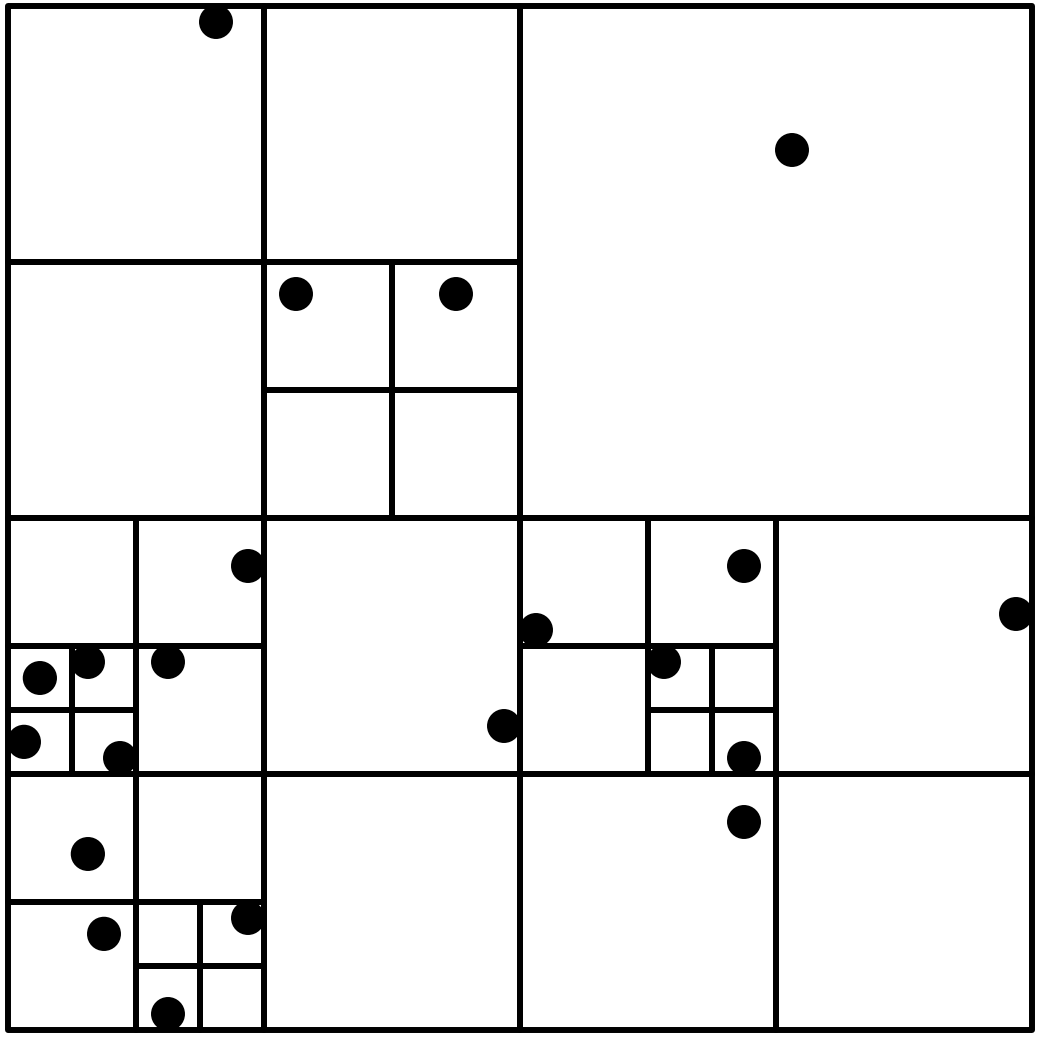


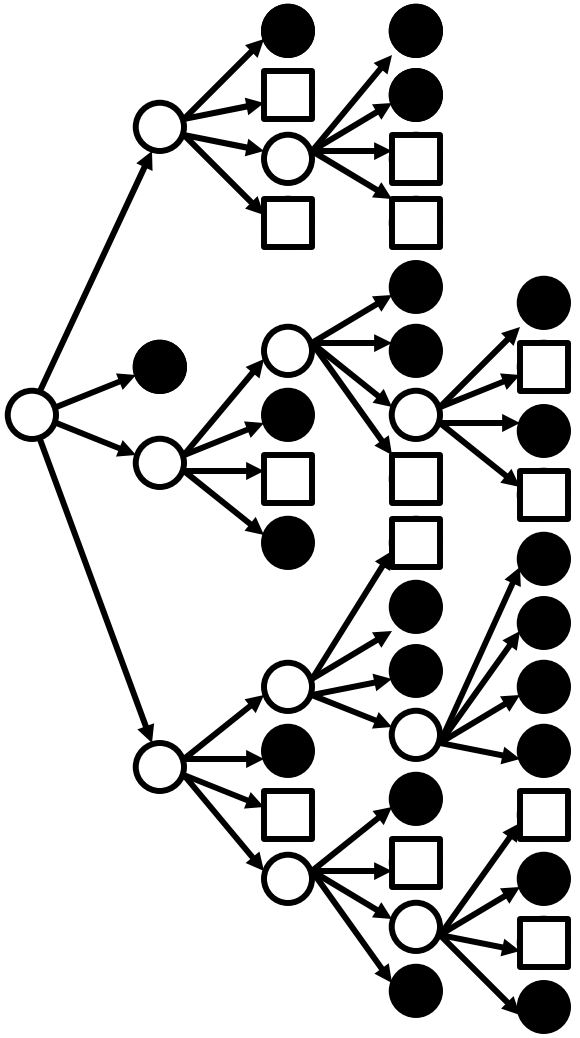
Barnes-Hut-Approximation





Barnes-Hut-Approximation

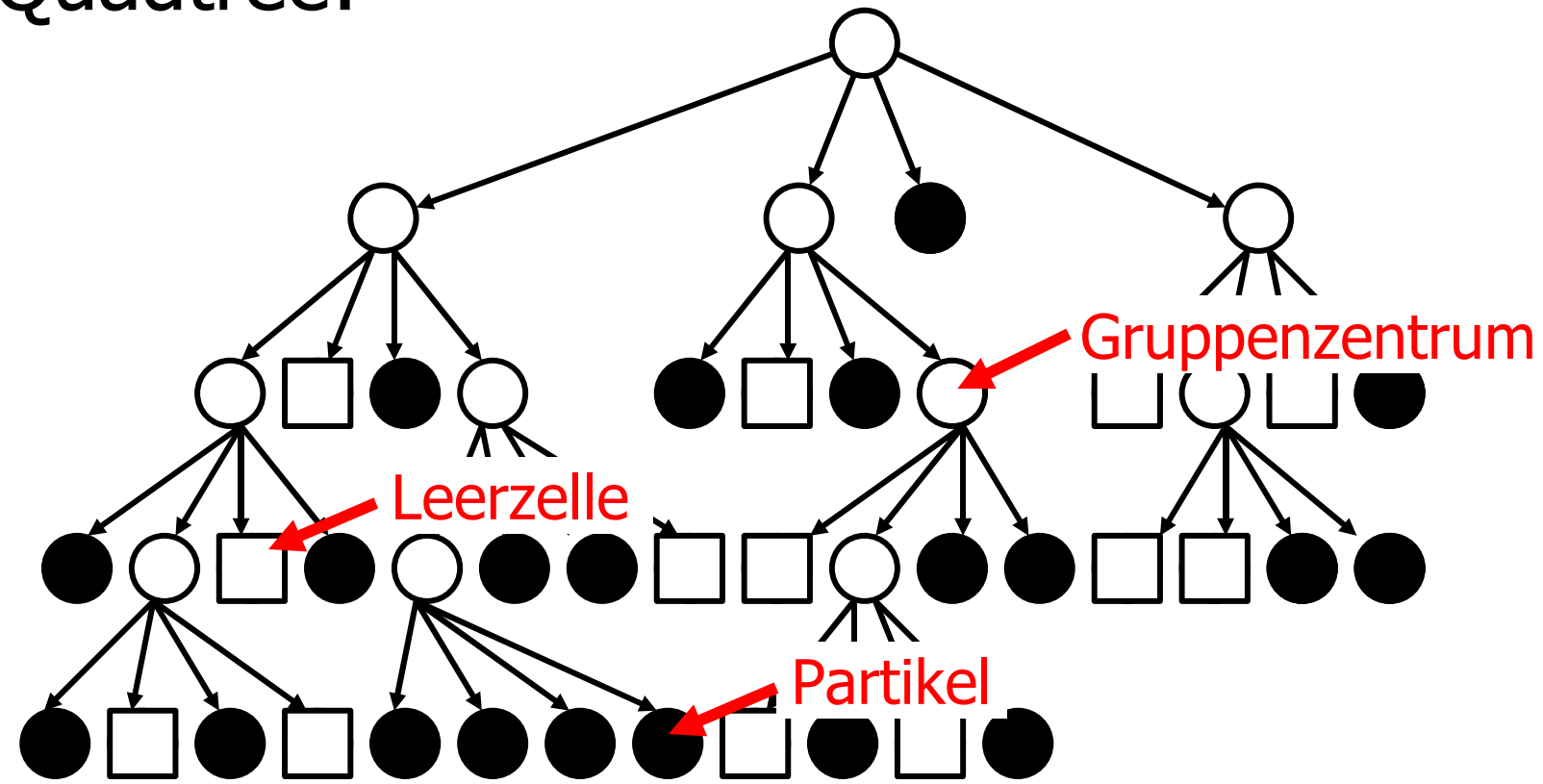




Barnes-Hut-Approximation

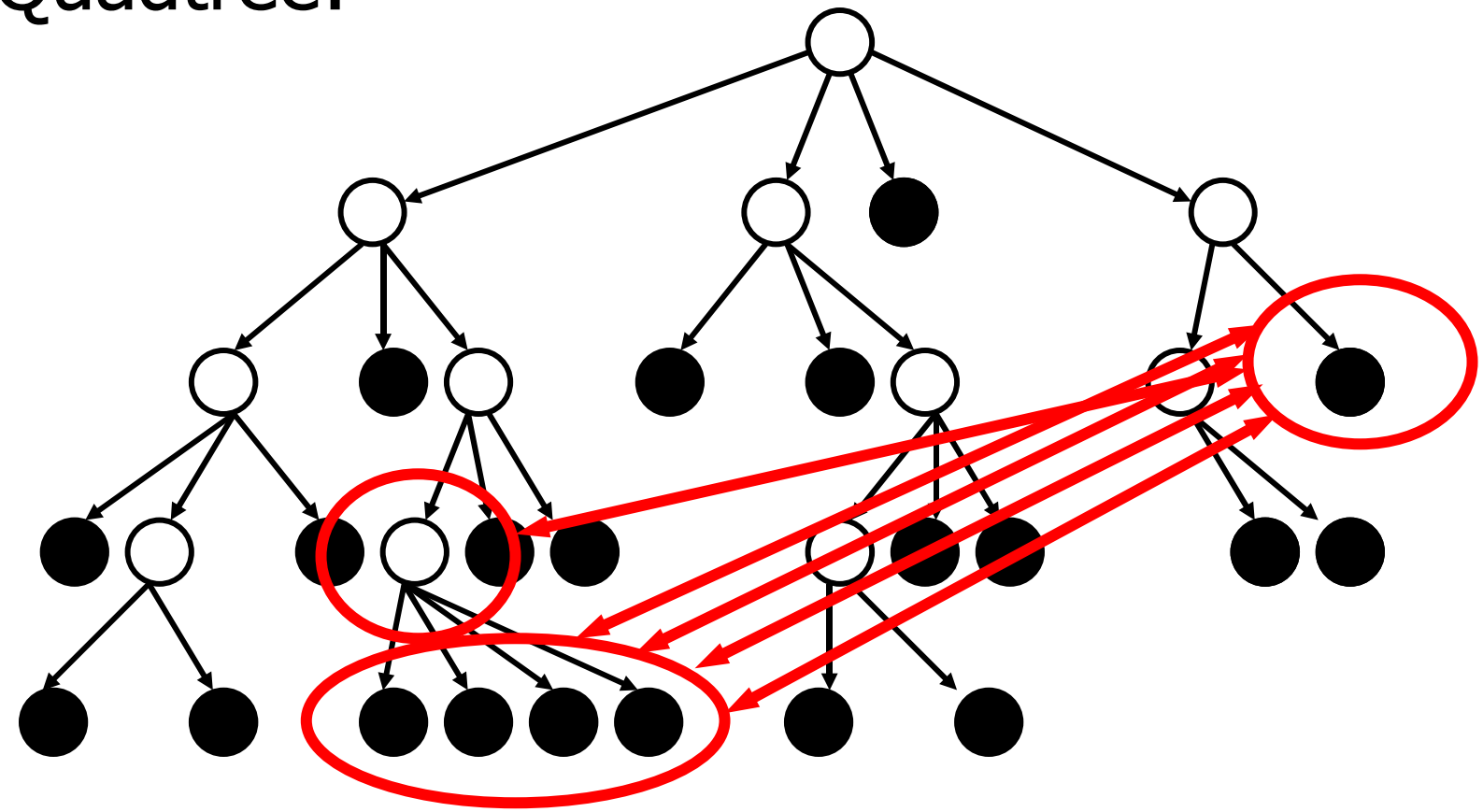
Institut für Informatik

Quadtree:



Barnes-Hut-Approximation

Quadtree:



Barnes-Hut-Approximation

- Gruppierung der Partikel mittels "Quadtree" (bzw. "Octtree" in 3D-Räumen)
- Berechnung der Massen und Positionen für die Gruppenzentren (Summe bzw. Mittelwert)
- Sofern Distanz zwischen Partikel und Gruppenzentrum groß genug:
- Berechne Kräfte zwischen Partikel P und Gruppenzentrum G ...
- ...und nimm diese Kraftwirkung für alle Partikel im Baum unterhalb von G

Fazit

- *Scientific computing* befasst sich mit der Lösung/Approximation immens großer Probleme
- Diese Lösung kann durch Einsatz paralleler/verteilter Algorithmen deutlich beschleunigt werden,
- aber immer nur in konstanter Abhängigkeit von der Zahl der Prozessoren!
- Effiziente Verbesserung kann nur durch die Wahl effizienterer Algorithmen erreicht werden,
- nicht durch Einsatz von mehr/schnelleren Prozessoren!

Nachdenken statt Nachrüsten!

Vielen Dank
für die Aufmerksamkeit!

Quellen

- Andrews, Gregory R.: *Foundations of Multithreaded, Parallel, and Distributed Programming*, Addison-Wesley, 2000
- Barnes, J. und Hut, P.: *A hierarchical O(n log n) force-calculation algorithm*, Nature, 324 (1986), pp. 446-449
- Graps, A.: *Amara's Recap of Particle Simulation Methods*, http://www.amara.com/papers/nbody.html, 2000
- Prins, J.F. und Simons, M.: *The Barnes-Hut Algorithm as a case study for parallel programming models*, Dagstuhl Seminar on High-level parallel programming models, April 1999
- de.wikipedia.org