**Smoothed Particle Hydrodynamics**

# SPH
# Implementierung für das Software-Praktikum im SS 2008

Johannes Willkomm
Martin Bücker
Michael Lülfesmann
Arno Rasch
Andreas Wolf

# SPH Überblick

- Smoothed Particle Hydrodynamics (SPH) wird zur Diskretisierung von partiellen Differenzialgleichungen verwendet.
- Die Methode kommt  ohne das sonst übliche festes Gitter aus
  - Phasengrenzen daher kein Problem
- Lagrange-Ansatz (statt Euler-Gleichungen)
  - Partikel mit fester Masse: diese bleibt automatisch erhalten
- Kernfunktion ist verschmierte Dirac-Funktion
  - Partikel innerhalb der Einflußlänge *h* beeinflussen sich
- Nachteile
  - Da die Partikel sich frei bewegen können wenig Kontrolle über die Auflösung an einem bestimmten Ort
  - Physikalische Korrektheit?

# Unser Ansatz

- Weitgehend wie in Monaghan „Simulating Free Surface Flows with SPH“
  - *d*-dimensional, *d*=2, *d*=2
  - Mit Spline-Kernfunktion (aus Liu & Liu)

$$W(r)=\begin{cases}\alpha_d\left(\frac{2}{3}-\left(\frac{r}{h}\right)^2+\frac{1}{2}\left(\frac{r}{h}\right)^3\right) & 0<r<h \\ \alpha_d\left(2-\frac{r}{h}\right)^3 & h<r<2\mathrm{h} \\ 0 & sonst\end{cases}$$

  - Variablen: Position **x**, Geschwindigkeit **v**, Dichte ρ, Wärme u
  - Zustandsgleichung für Druck P (andere möglich)

$$P=B\left(\left(\frac{\rho}{\rho_0}\right)^{\gamma}-1\right)$$

# Unser Ansatz

- Geschwindigkeit

$$\frac{d\,\boldsymbol{v}_i}{d\,t} = -\sum_{j \in Wasser(N(i))} m_j \left( \frac{P_i}{\rho_i} + \frac{P_j}{\rho_j} + \Pi_{ij} \right) \nabla_i W_{ij} + \sum_{j \in LJ(N(i))} LJ_{ij} + \boldsymbol{F}_i$$

  - Wasser($N(i)$): Wasser-Partikel in Nachbarschaft von *i*
  - LJ($N(i)$): Randpartikel in Nachbarschaft von *i*
  - $\boldsymbol{F}_i$ Kräfte (Schwerkraft)
  - $\nabla_i W_{ij}$ Ableitung der Kernfuktion (Gradient) bzgl. Ort von *i*
  - $\Pi_{ij}$ Viskosität verhindert Durchdringung der Partikel
  - $LJ_{ij}$ Beschleunigung durch Randpartikel *j*

- Dichte $\frac{d\,\rho_i}{d\,t} = \sum_{j \in Wasser(N(i))} m_j (\boldsymbol{v}_i - \boldsymbol{v}_j) \nabla_i W_{ij}$

- Wärme $\frac{d\,u_i}{d\,t} = \frac{1}{2} \sum_{j \in Wasser(N(i))} m_j \left( \frac{P_i}{\rho_i} + \frac{P_j}{\rho_j} + \Pi_{ij} \right) (\boldsymbol{v}_i - \boldsymbol{v}_j) \nabla_i W_{ij}$

# Lennard-Jones Randpartikel

- Verlässt ein Partikel das Simulationsgebiet Ω wird es nicht mehr berücksichtigt
- Randpartikel bilden Wände für die Flüssigkeit (Gefäße)
  - Sie üben nur eine ablenkende Beschleunigung aus
  - Abgeschnittenes Lennard-Jones-Potential:

$$LJ_{ij} = \begin{cases} D\left(\left(\frac{r_0}{r}\right)^{2p_1} - \left(\frac{r_0}{r}\right)^{p_1}\right)\frac{\boldsymbol{x}_i - \boldsymbol{x}_j}{r}, & r < r_0 \\ 0, & r \geq r_0 \end{cases}$$

  - D, $p_1$ und $r_0$ sind Parameter und $r = \|\boldsymbol{x}_i - \boldsymbol{x}_j\|$
  - Wegen der Nachbarsuche verlangen wir, daß der Einfluß der Randpartikel nicht weiter reicht als die der Kernfunktion:

$$r_0 \leq 2\mathrm{h}$$

# Örtliches Raster

- Das Rechengebiet Ω wird definiert über zwei Punkte $P_1$, $P_2$
  - Länge in jeder Dimension: $\boldsymbol{L} = P_2 - P_1$
- Da die *smoothing length h* fest und für alle Partikel gleich ist, verwenden wir ein globales Raster der Länge $H=2h$
  - Jedes Partikel erhält dadurch einen *d*-dimens. Integer Index $\boldsymbol{I} = floor\left(\frac{\boldsymbol{x} - P_1}{H}\right)$
  - Die Anzahl Rasterzellen in jeder Dimension ist $\boldsymbol{N} = ceil\left(\frac{\boldsymbol{L}}{H}\right)$
  - und insgesamt:

$$n_R = \prod \boldsymbol{N} \sim O\left(\left(\frac{1}{h}\right)^d\right)$$

# Örtliches Raster

- Die Raster-Zellen werden durchnummeriert „wie üblich“
  - Jede Zelle erhält dadurch einen 1-D Integer-Index *i*
- Berechnung:

$$i = \boldsymbol{I} * \boldsymbol{W}$$

mit

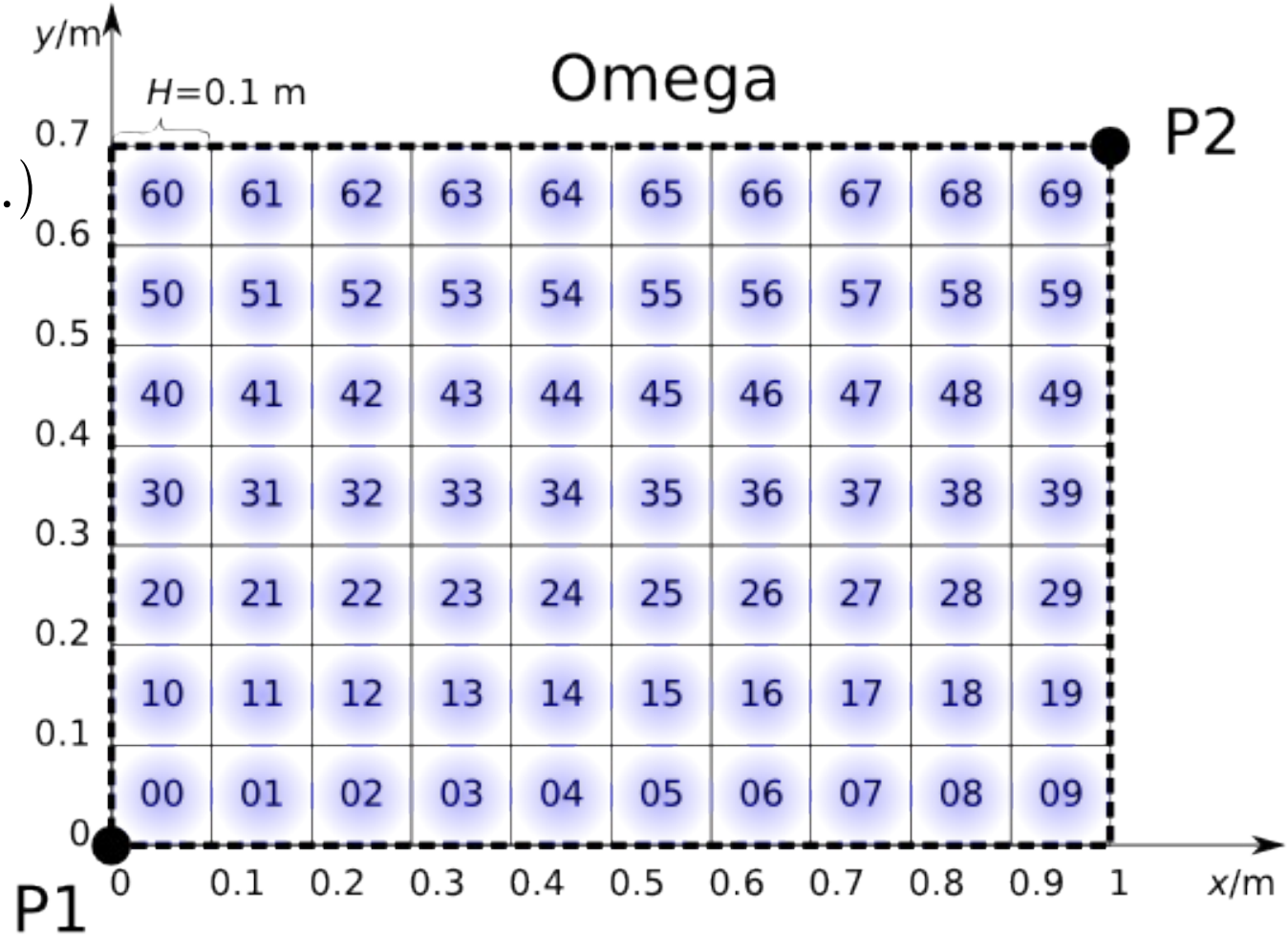
$$\boldsymbol{W} = (1, N_0, N_0 N_1, \ldots)$$

  - hier:

$$N_0 = 10$$

$$N_1 = 7$$

$$\boldsymbol{W} = (1, 10)$$

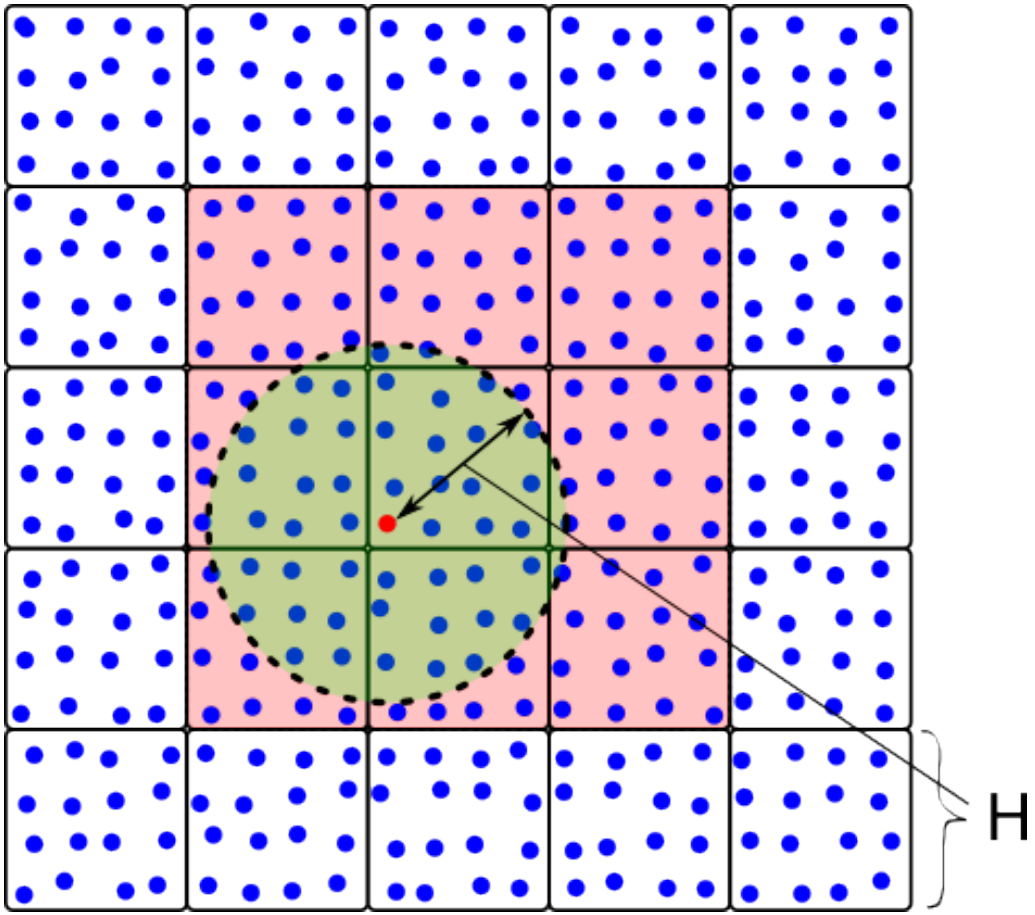


# Summation der Einflüsse (einfach)

- Für alle (beweglichen) Partikel:
  - Teste alle Partikel in gleicher und umliegenden Zellen ob Abstand < 2*h*
  - Alle anderen sind auf jeden Fall weiter entfernt
- Anzahl umliegender Zellen:

$$n_d = 3^d, \quad n_1 = 3, \quad n_2 = 9, \quad n_3 = 27$$

- Bild: Nachbar-Zellen in 2D

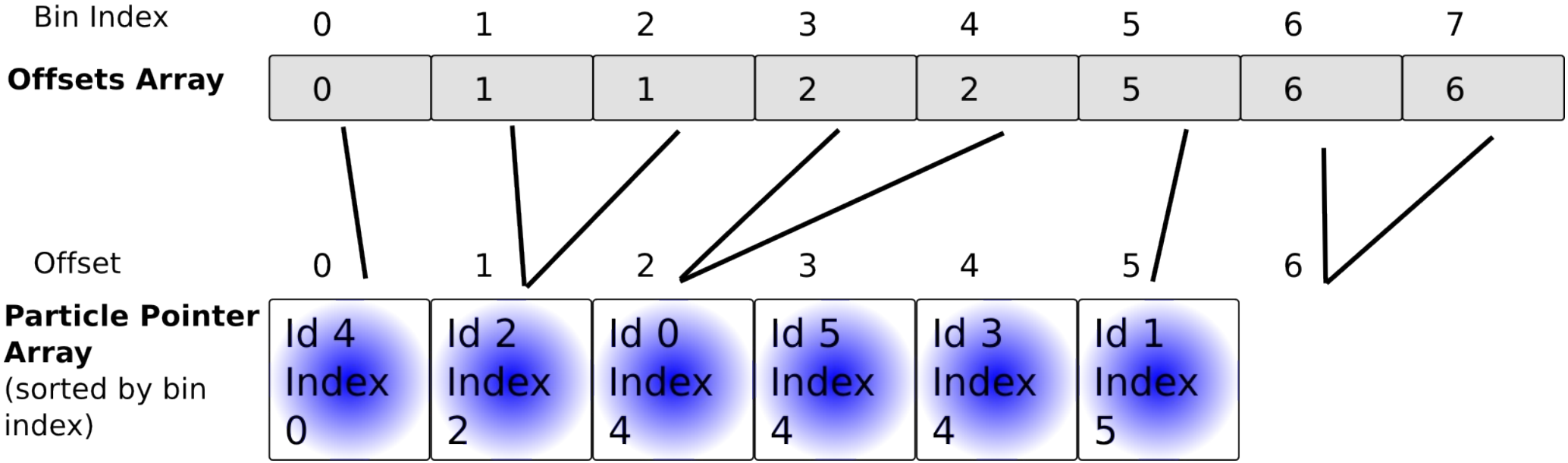


# Lookup Zelle->Partikel

- Grundidee: Eine Liste je Rasterzelle
  - Jede Liste enthält Zeiger auf Partikel in der Zelle
- Klasse SortedParticleIndex enthält
  - Vektor pointers mit Zeigern auf alle Partikel, sortiert nach 1D-Integer-Index (Länge *n*)
    - Partikel in gleicher Zelle liegen nebeneinander (*runs*)
  - Vektor offsets (Länge $n_R$) enthält Begin des *runs* je Zelle
- Liste von Zelle *i* = *run* der Partikel mit Index *i* in pointers
- Anzahl der Partikel in Zelle i = offsets[i+1] - offsets[i]

# Lookup Zelle->Partikel

- Beispiel: $n_R$=8 Raster-Zellen, $n$=6 Partikel
- Beginn der Liste Zelle *i*: offsets[i], Länge: offsets[i+1] - offsets[i]




- 1 Partikel in Zelle 0
- 1 Partikel in Zelle 2
- 3 Partikel in Zelle 4
- 1 Partikel in Zelle 5

# Aktualisieren der Datenstruktur

- Sortiere Liste pointers mit parallelem Integer-Sortieralgorithmus
  - Schneller als allgemeines vergleichsbasiertes Sortieren
  - Hier anwendbar: Maximal $n_R$ mögliche Indizes
- Paralleles Counting Sort nach

  „Practical parallel algorithms for personalized communication and integer sorting“ (1996)

  David A. Bader, David R. Helman and Joseph J'aj'a

  ACM Journal of Experimental Algorithmics
- Erstellt Histogramm hist (zähle Partikel in jeder Zelle)
- Dann Präfixsummen pref davon:

$$\mathrm{pref}[i] = \sum_{j=0}^{i} \mathrm{hist}[j]$$

  - offsets = shift(pref, 1)
  - Was vorher in einem Postprocessing-Schritt erstellt werden mußte ist hier Nebenprodukt des Sortieralgorithmus!

# Paralleles Histogram

- N = Anzahl Daten, m_valueRange = Anzahl mögl. Werte
  - omp parallel:

```
myLocalCount = localCounts + myid;
myLocalCount->resize(m_valueRange);

#ifdef _OPENMP
#pragma omp for
#endif
      for(long i = 0; i < N; ++i) {
        value_type v = m_valueGetter(*(beg + i));
        ++(*myLocalCount)[v];
      }

      for(size_t j = 0; j < nthreads; ++j) {
#ifdef _OPENMP
#pragma omp for
#endif
        for(long i = 0; i < long(m_valueRange); ++i) {
          m_counts[i] += localCounts[j][i];
        }
      }
```

# Parallele Präfixsummen

- myLowN, myHighN: Bereich der Daten von Thread
  - omp parallel:

```
      for(size_t i = myLowM + 1; i < myHighM; ++i) {
        m_histogram[i] += m_histogram[i-1];
      }

      totals[myid] = m_histogram[myHighM - 1];

#ifdef _OPENMP
#pragma omp barrier
#endif
      if (myid == 0) {
        for(size_t i = 1; i < nthreads; ++i) {
          totals[i] += totals[i - 1];
        }
      }

#ifdef _OPENMP
#pragma omp barrier
#endif
      if (myid > 0) {
        size_t const myTotal = totals[myid-1];
        for(size_t i = myLowM; i < myHighM; ++i) {
          m_histogram[i] += myTotal;
        }
      }
```

# 3-Schritt Paralleles Counting Sort

- Es wird anhand der Präfixsummen in drei Schritten umsortiert
  - „An h-balanced personal communication“
  - h = $N/p^2 + p/2$
  - Zwei Schritte in denen sich die Prozessoren gegenseitig Aufgaben zuweisen
  - Dazu können zwei Arrays der Größe $p$ x $p$ x h verwendet werden
  - Im 2. und 3. Schritt maximal $p * h$ Werte für jeden Prozessor zu behandeln (im 3. werden die Werte umsortiert)
- Auf folgender Folie zunächst eine vereinfachte Fassung mit nur einem Zwischenschritt

# 2-Schritt Paralleles Counting Sort

- omp parallel:

```
    if (myid == 0) {
      tmp.resize(nthreads);
    }

#pragma omp barrier
    tmp[myid].resize(nthreads);

    for(size_t i = myLowN; i < myHighN; ++i) {
      value_type v = m_valueGetter(*(beg + i));
      long offset = prefixSums[v] - 1;
      long const targetProc = offset / myRangeN;
      tmp[myid][targetProc].push_back(*(beg + i));
    }

#pragma omp barrier
    ; // "this pragma must immediately precede a statement"
    long offset;
    for(size_t i = 0; i < nthreads; ++i) {
      std::vector<it_value_type> const &bin = tmp[i][myid];
      for(size_t j = 0; j < bin.size(); ++j) {
        it_value_type vind = bin[j];
        value_type v = m_valueGetter(vind);
        offset = --prefixSums[v];
        *(beg + offset) = vind;
      }
    }
```

Nicht bekannt wie viele Werte jedes tmp[myid][targetProc] erhalten wird

Hier nicht balancierte Last wenn Werte nicht gleichmäßig über Wertebereich verteilt

# 3-Schritt Paralleles Counting Sort

- omp parallel:

```
      std::valarray<unsigned> sendToProc(nthreads);

      for(size_t i = myLowN; i < myHighN; ++i) {
        value_type v = m_valueGetter(*(beg + i));
        long offset = prefixSums[v] - 1;
        long const targetProc = offset / myRangeN;
        tmp[myid][(myid + targetProc + sendToProc[targetProc]) % nthreads].push_back(*(beg + i));
        ++sendToProc[targetProc];
      }

#pragma omp barrier

      for(size_t i = 0; i < nthreads; ++i) {
        FixedSizedArray<it_value_type> const &bin = tmp[i][myid];
        for(size_t j = 0; j < bin.size(); ++j) {
          it_value_type vind = bin[j];
          value_type v = m_valueGetter(vind);
          long offset = prefixSums[v] - 1;
          long const targetProc = offset / myRangeN;
          tmp2[myid][targetProc].push_back(vind);
        }
      }

#pragma omp barrier

      long offset;
      for(size_t i = 0; i < nthreads; ++i) {
        FixedSizedArray<it_value_type> const &bin = tmp2[i][myid];
        for(size_t j = 0; j < bin.size(); ++j) {
          it_value_type vind = bin[j];
          value_type v = m_valueGetter(vind);
          offset = --prefixSums[v];
          *(beg + offset) = vind;
        }
      }
```

# Lookup Zelle->Partikel

- Vorteile
  - Lookup sehr schnell $O(1)$
  - Paralleles sortieren theoretisch nur $O(n)$ statt $n*\log(n)$
    - Kein Faktor $\log(n)$ wie bei Baumstruktur
  - Speicherbedarf für $n_R$ „Listen“ ist minimiert: $n_R * 4$ Byte
    - doppelt verkettete Liste: $n_R * 2 * 8$ Byte
    - nicht mehr als $2^{32}$ Partikel
  - Sortieren verschiebt außerdem Partikel die das Gebiet verlassen haben ans Ende der Liste
- Nachteile
  - Neuerstellen der gesamten Datenstruktur in jedem Schritt
    - Denkbar wäre bei anderer Datenstruktur umsortieren nur von den Partikeln die ihre Zelle gewechselt haben

# Hashing

- Bilde elementweise Rest $2^m$ des Integer-Index
  - $\boldsymbol{I}_H = \boldsymbol{I} \bmod 2^m$
  - Anzahl Hash-Zellen: $n_H = (2^m)^d = 2^{md}$
  - Berechung effizient durch **bitand** mit $(2^m - 1)$
  - 1D-Hash Integer

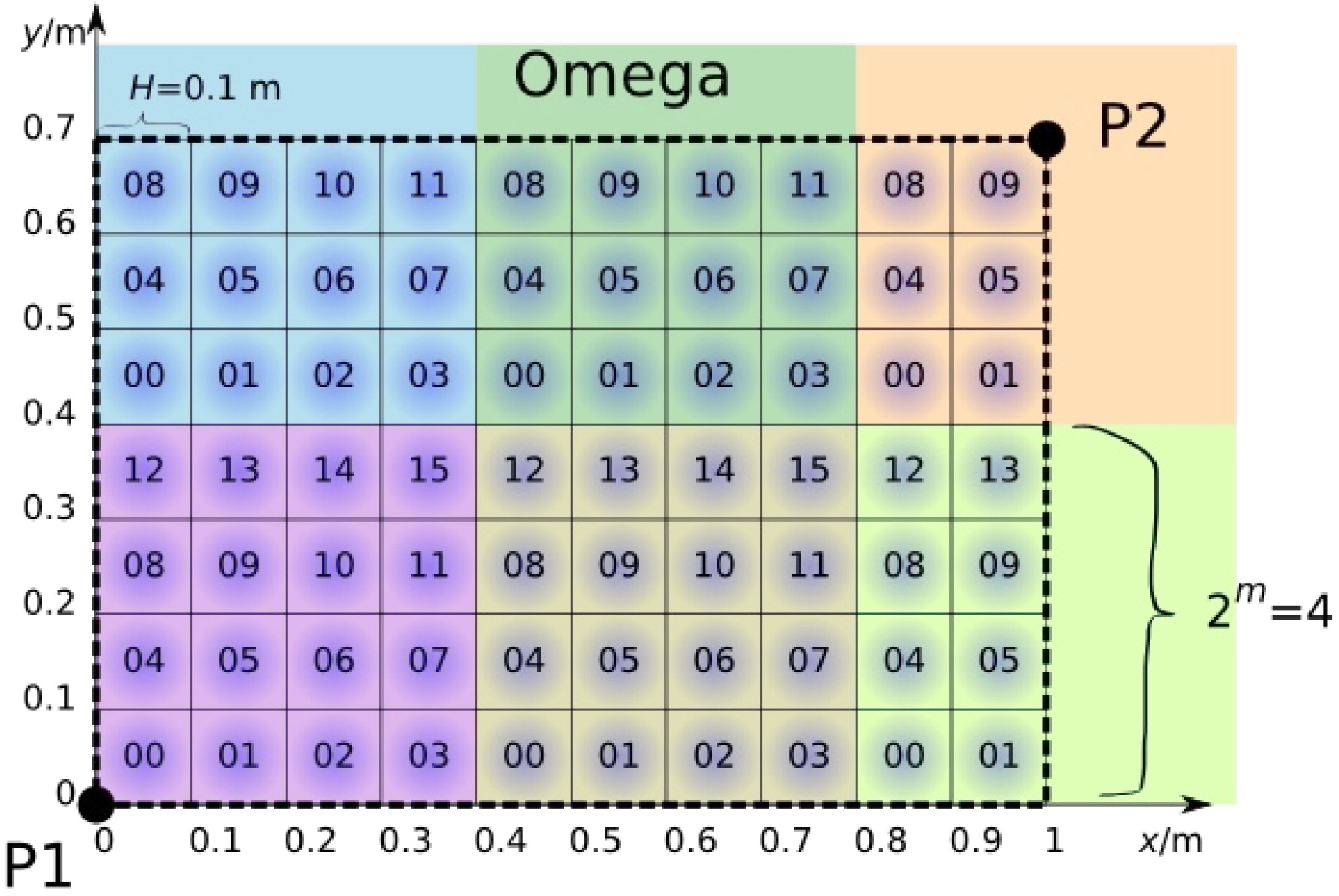
    $$i_H = \boldsymbol{I}_H * \boldsymbol{W}_H, \quad \boldsymbol{W}_H = \left(1, 2^m, 2^{2m}, \ldots\right)$$

  - Berechung effizent durch elementweises Linksschieben und Kombination mit **bitor**:

    $$i_H = \boldsymbol{bitor}\left(\boldsymbol{I}_H \ll \boldsymbol{S}_H\right), \quad \boldsymbol{S}_H = \left(0, m, 2m, \ldots\right)$$

- Begrenzt Speicherbedarf: Array offsets von SortedParticleIndex hat nun nur Länge $n_H$
- Auch andere Hashfunktionen möglich
  - Bessere Lastbalancierung mit randomisierendem Hashing?

# Hashing


y/m
H=0.1 m
Omega
P2
P1
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1
x/m
08 09 10 11 08 09 10 11 08 09
04 05 06 07 04 05 06 07 04 05
00 01 02 03 00 01 02 03 00 01
12 13 14 15 12 13 14 15 12 13
08 09 10 11 08 09 10 11 08 09
04 05 06 07 04 05 06 07 04 05
00 01 02 03 00 01 02 03 00 01
$2^m=4$

# Zeit-Integration

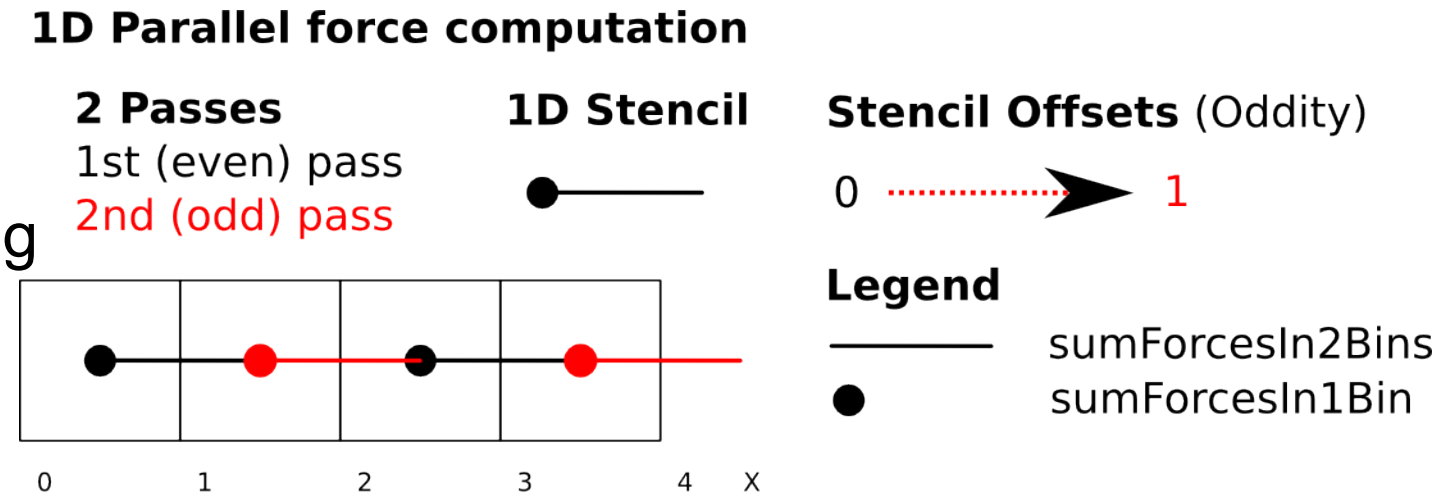
- Separates Projekt „integrate“, verschiedene Verfahren als C++-Templates implementiert
  - Verwendbar mit Skalaren und Vektoren
  - Typparameter „ForceFunctor“ berechnet Einflüsse
    - Klasse mit Funktionen operator() und update
- **Einzelschrittverfahren** verwenden zusätzliche Stützstellen zwischen $t$ und $t + \mathrm{d}t$
  - Explizites Euler-Verfahren (Sonderfall, keine Stützstellen)
  - Runge-Kutta-, Trapez-, Heun-Regeln
    - Verfahren bestimmt durch Ordnung $p$, Koeffizienten-Vektoren $a$, $c$ und Koeffizientenmatrix $b$
  - $p$ Funktionsauswertungen je Schritt
  - Quelle: Bronstein

# Zeit-Integration

- Mehrschrittverfahren verwenden gewichtete Zustände aus der Vergangenheit
  - Adams-Bashforth-Methode (ABM)
    - Bis 7. Ordnung, weitere möglich
    - Eine Funktionsauswertung je Schritt
  - Adams-Moulton-Methode (AMM)
    - Implizites Verfahren, erfordert Schätzwert in der Zukunft, durch impliziten Schritt (nicht implementiert)
  - Verwenden Einzelschritt-Methode für die ersten Schritte
- Prädiktor-Korrektor-Verfahren (PK)
  - AMM mit ABM als Schätzer
  - Zwei Funktionsauswertungen je Schritt
- Quelle: Artikel Mehrschrittverfahren in der Wikipedia

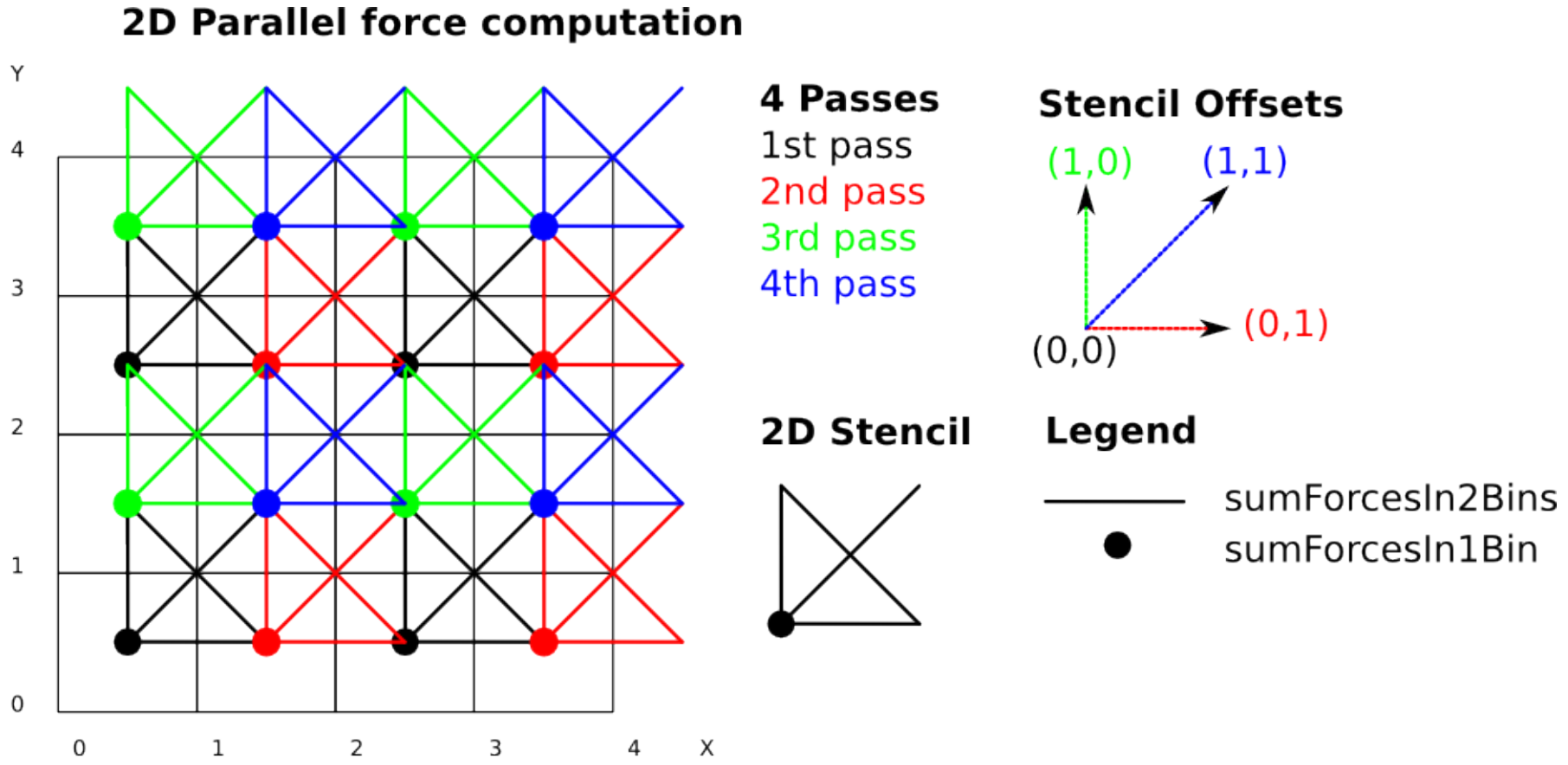
# Gebiets-Parallelisierung

- Idee: Einflüsse sind (fast) symmetrisch
  - Einfluss jedes Partikelpaars nur einmal berechnen
- Parallelisierung erfordert Aufteilung des Gebiets um race conditions zu verweiden
  - Paare von Partikel müssen einem Thread  zugeteilt werden
- Ansatz: Teile Paare von Rasterzellen Threads zu
  - Hilfsfunktionen sumForcesIn1Bin, sumForcesIn2Bins
- Ansatz in 1D:
  - 2 Durchläufe
  - Parallelisierung der einzelnen Durchläufe



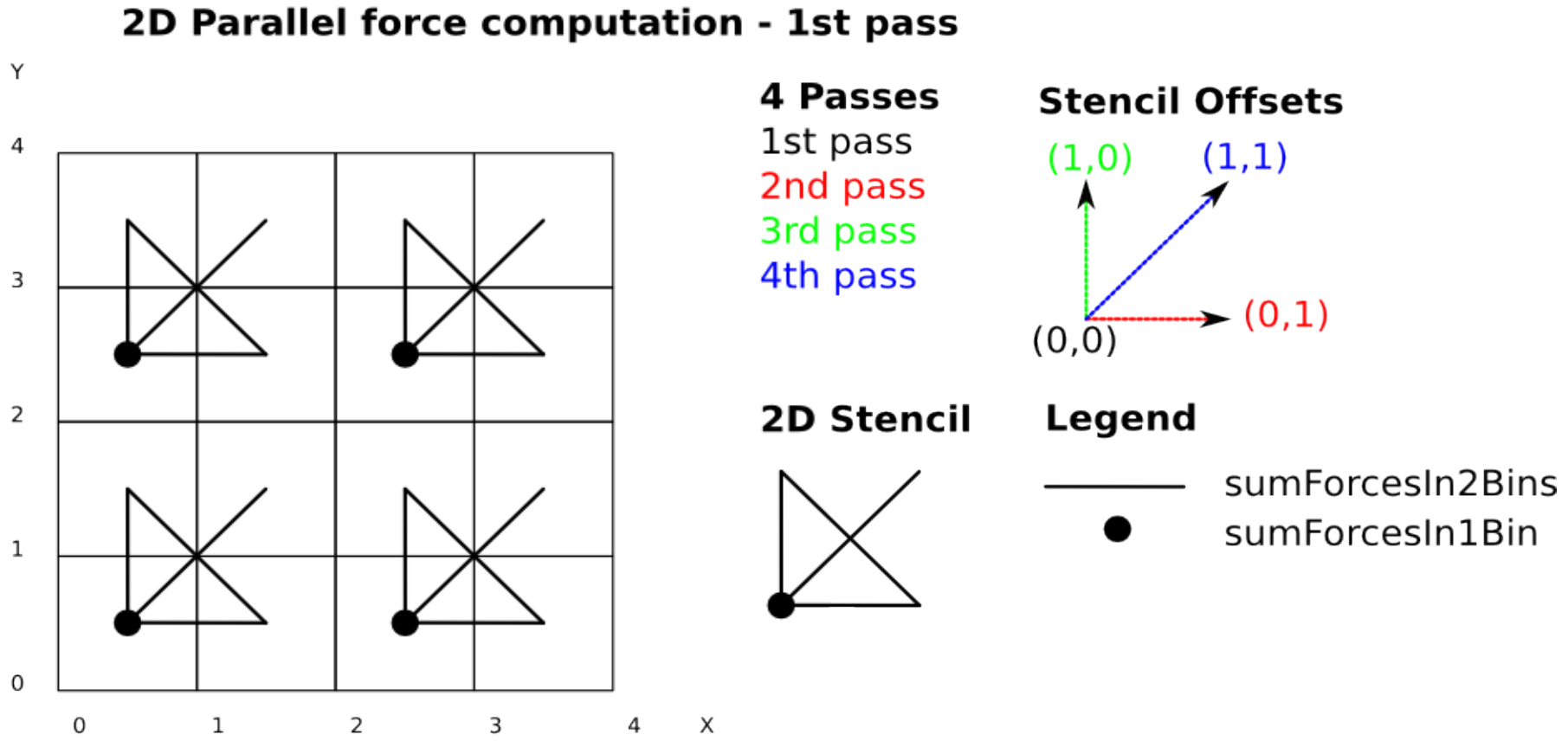
# Gebiets-Parallelisierung 2D

- Verallgemeinerung auf 2D, 3D?
- Graphische Lösung 2D:



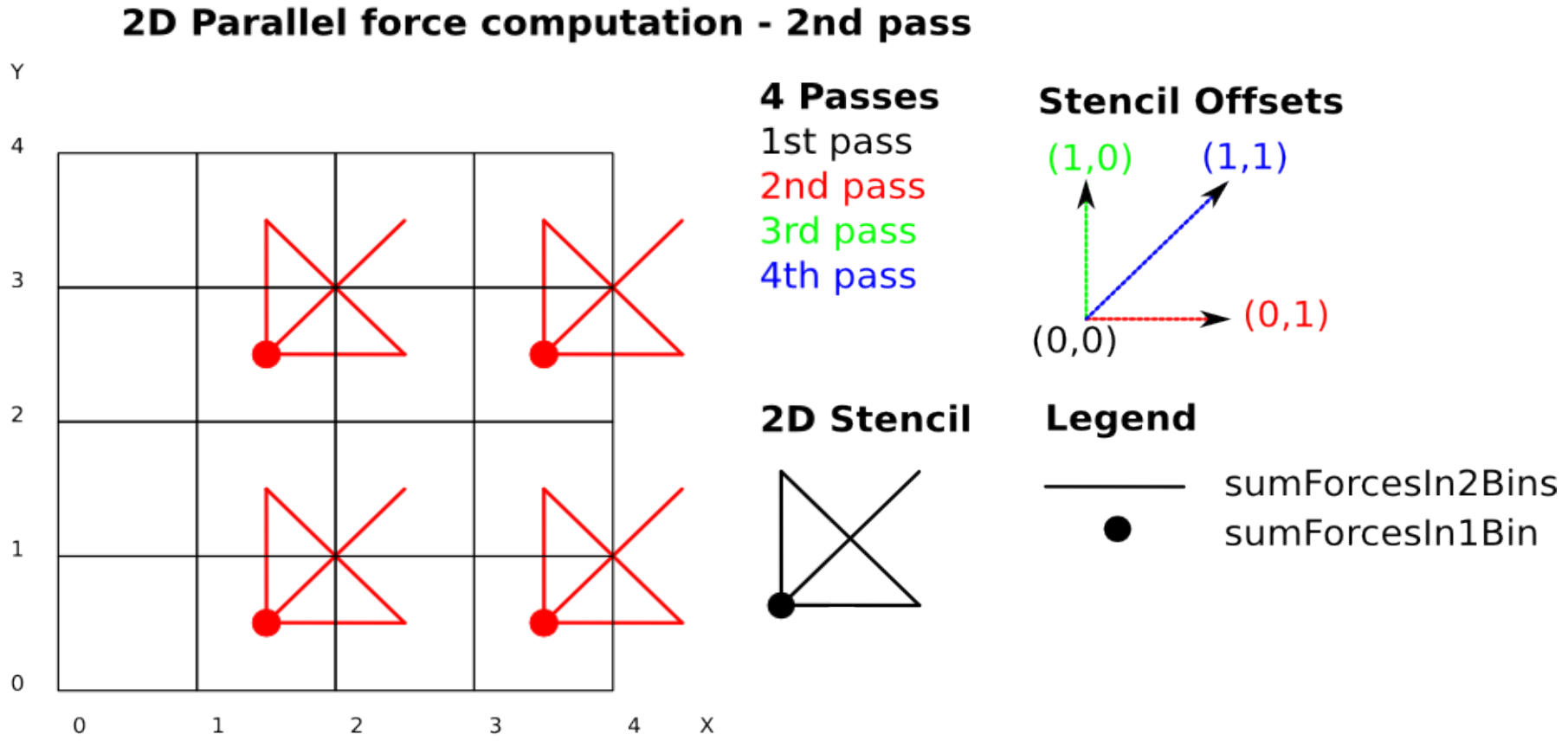
# Gebiets-Parallelisierung 2D

- Graphische Lösung 2D:



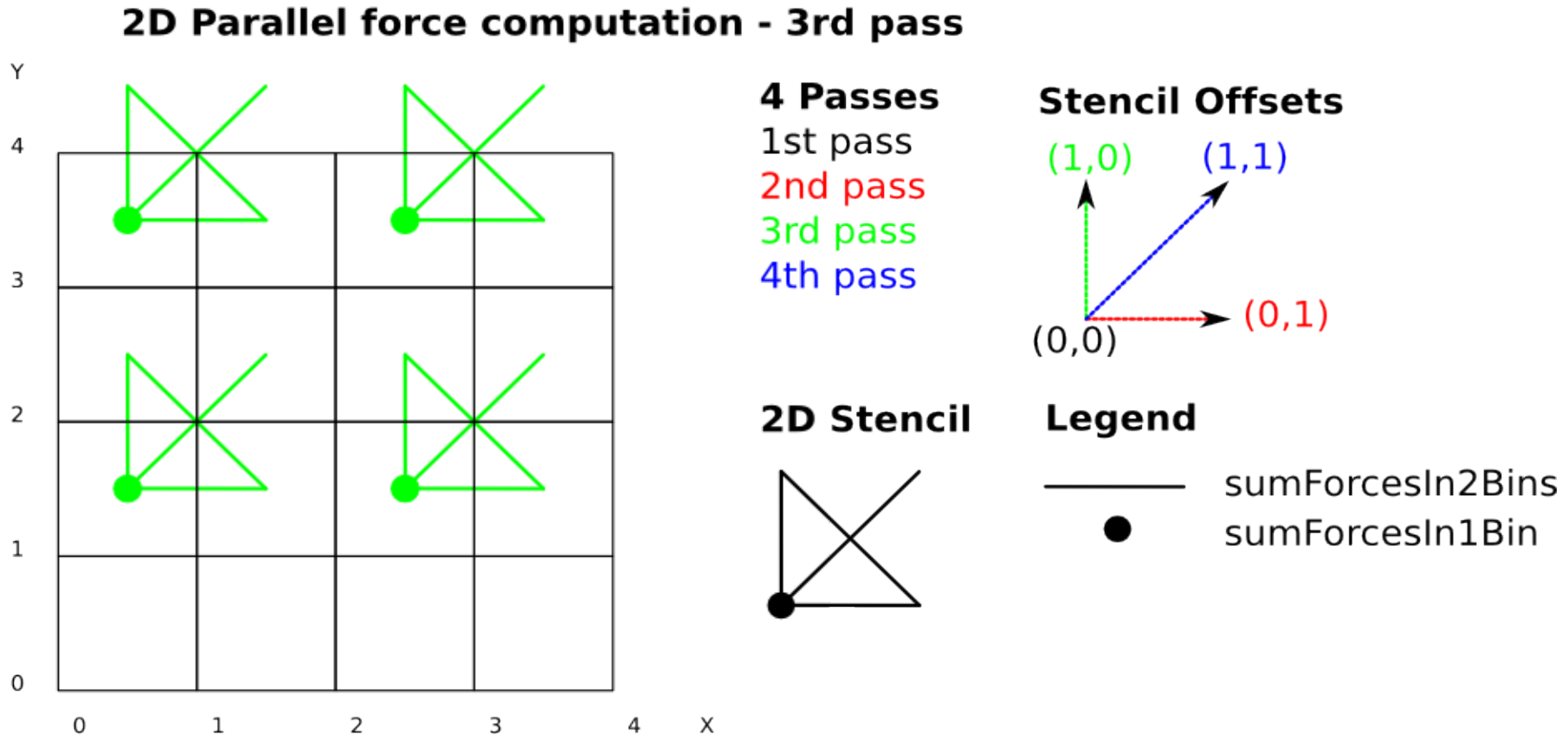
# Gebiets-Parallelisierung 2D

- Graphische Lösung 2D:



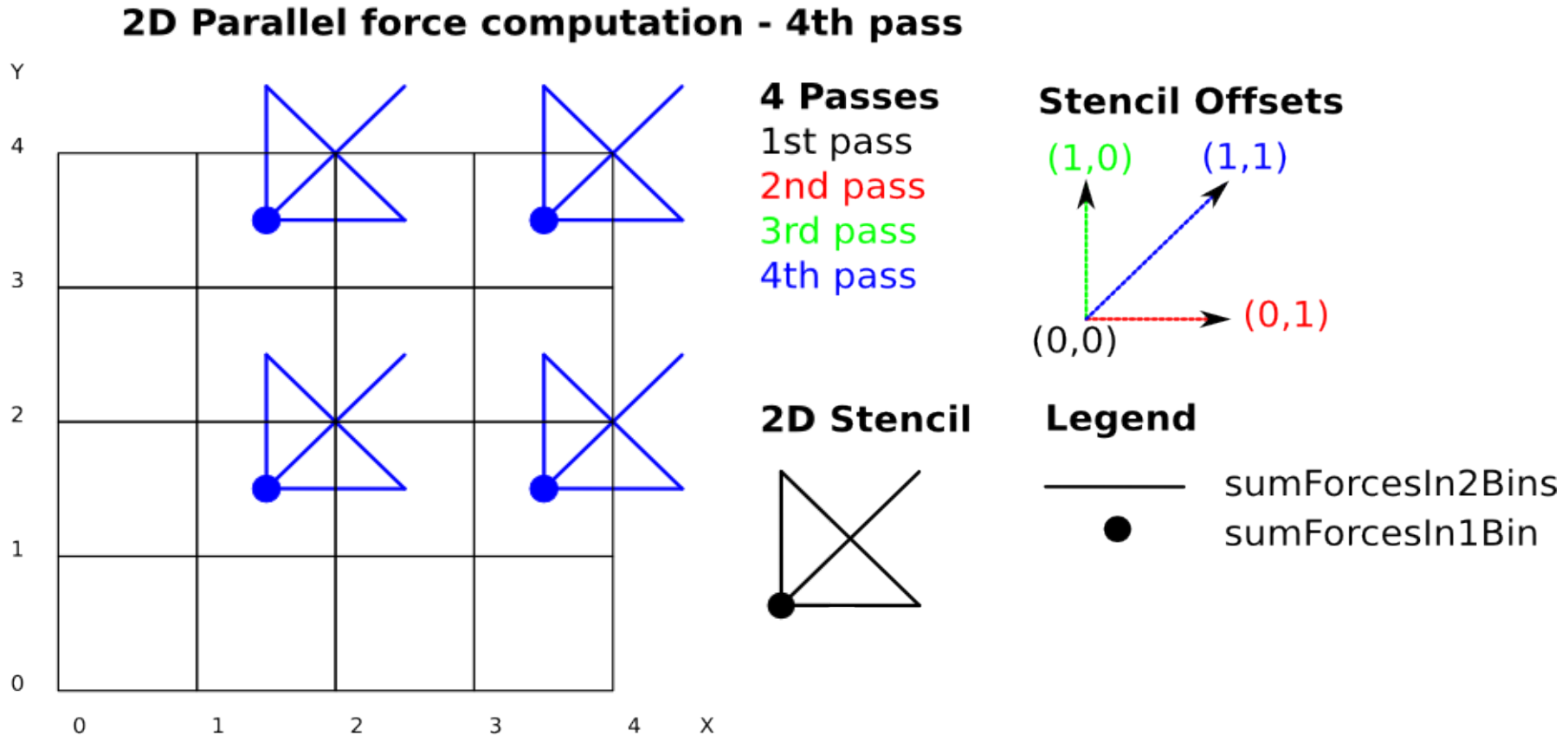
# Gebiets-Parallelisierung 2D

- Graphische Lösung 2D:



# Gebiets-Parallelisierung 2D

- Graphische Lösung 2D:



# Gebiets-Parallelisierung

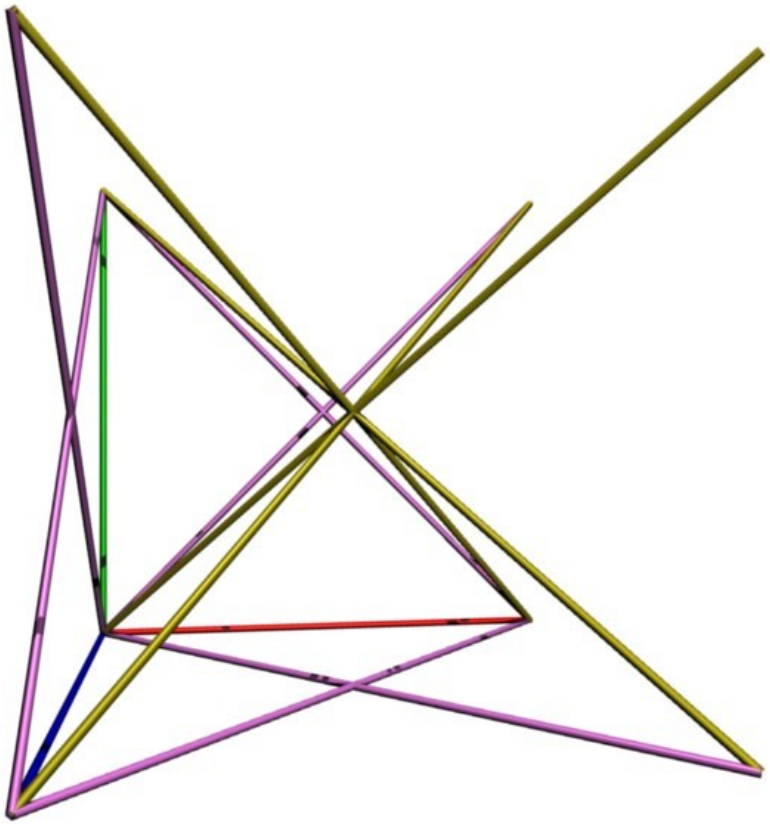
- Grund-Raster: Alle Zellen mit nur geradem Integer-Index
- Verschiebung (Offset):
  - 1D: gerade und ungerade Zellen 2
  - 2D und 3D: $2^d$ Möglichkeiten
  - Definiere Feld P mit allen möglichen Offsets:
    - Aufzählung von $\{0,1\}^d$

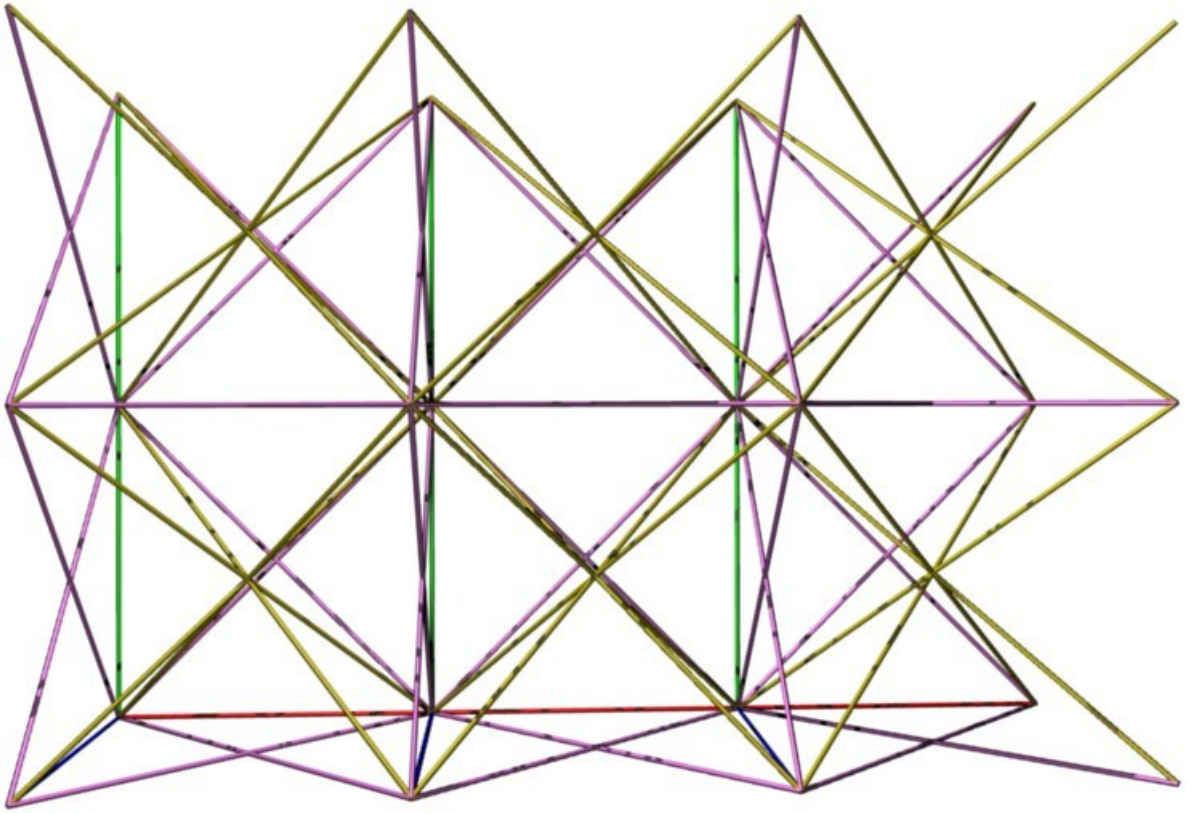
P bei d=2:
- (0, 0)
- (0, 1)
- (1, 0)
- (1, 1)

- Das Muster (Stencil) muß so aussehen, daß
  - Alle Relationen zwischen Zellen berücksichtigt werden
  - Die Stencils eines Durchlaufs sich nicht überlappen (race conditions)
  - SumForcesIn1Bin: Anwenden auf Zelle (0,0,...) des Stencils
  - SumForcesIn2Bins: Anwenden auf alle Kombinationen (P[ *i* ] P[ *j* ]) für die P[ *i* ] * P[ *j* ] = 0

# Gebiets-Parallelisierung 3D

- In 3D sieht der Stencil so aus:





# Gebiets-Parallelisierung

- Algorithmus
  - **for** oddity in P: *// $2^d$ Durchläufe*
    - Parallel **for** pos in even$^d$ (hashed) raster positions:
      - sumForcesInOneBin(hash(pos + oddity))
      - **for** *i* in [ 1, $2^d$ – 1 ]
        - **for** *j* in [ 0, *i* - 1 ] *// alle Kombinationen (i,j)*
          - If (*i* **bitand** *j*) == 0
            - sumForcesIn2Bins(hash(pos + oddity + P[ *i* ]), hash(pos + oddity + P[ *j* ]))
- Annahme: ( *i* **bitand** *j* ) = 0 <==> (P[ *i* ] * P[ *j* ] = 0)
  - P[ *i* ] enthält *i* in Binärdarstellung als {0,1}-Vektor
- Parallele Schleife mit schedule(dynamic) oder schedule(static,10) um Last gleichmäßig zu verteilen

# Gebiets-Parallelisierung

- Wie viele Relationen zwischen Zellen gibt es in dem Stencil?
  - $I_1=1$, $I_2=4$, $I_3=13$
  - $I_d=(3^d-1)/2$
  - Sequence A003462 aus der AT&T Integer Sequence Encyclopedia
- Warum gerade die *i*,*j* mit P[ *i* ] * P[ *j* ] == 0?
  - Wenn P[ *i* ] * P[ *j* ] != 0 dann haben beide (mindestens) eine 1 in einer/mehreren Koordinate(n)
    - Diese Zellen-Kombination wird in anderem Durchlauf berücksichtigt: in dieser/n Dimension(en) 1 abziehen
      - Wird in dem Durchlauf berücksichtigt in dem oddity gerade diesen Wert hat.

# Gebiets-Parallelisierung

- Verbesserung von sumForcesIn1Bin, sumForcesIn2Bins:
  - Sortiere in SortedParticleIndex mit anderem Prädikat:
    - Sortiere nach Raster-Index
    - Dann nach beweglich/unbeweglich
    - Nun Partikel mit gleichem Index nebeneinander, erst bewegliche, dann unbewegliche
      - Lexikographische Ordnung nach (isOut, index, isBoundary)
  - Speichere Offsets bewegliche/unbewegliche Partikel separat
    - Verdoppelt Länge von offsets
  - Dann z.B. in sumForcesIn2Bins(*a*,*b*):
    - Kombiniere bewegliche aus *a* mit allen aus *b*
    - Kombiniere unbewegliche aus *a* mit beweglichen aus *b*
    - Tests p.isBoundary(), o.isBoundary() in Schleife entfallen

# Gebiets-Parallelisierung

- Probleme
  - Berechnung der Indizes ist nicht umsonst
    - Verwende 2er-Potenzen und bitweise Operationen
    - Effizienteste Aufzählung?
  - Immer Durchlaufen aller Rasterzellen auch wenn viele leer
    - Parameter *m* darf nicht zu klein und nicht zu groß sein für gute Laufzeiten
    - Vergleich naive Summation: Parameter *m* so groß wie es von Speicher her vertretbar ist

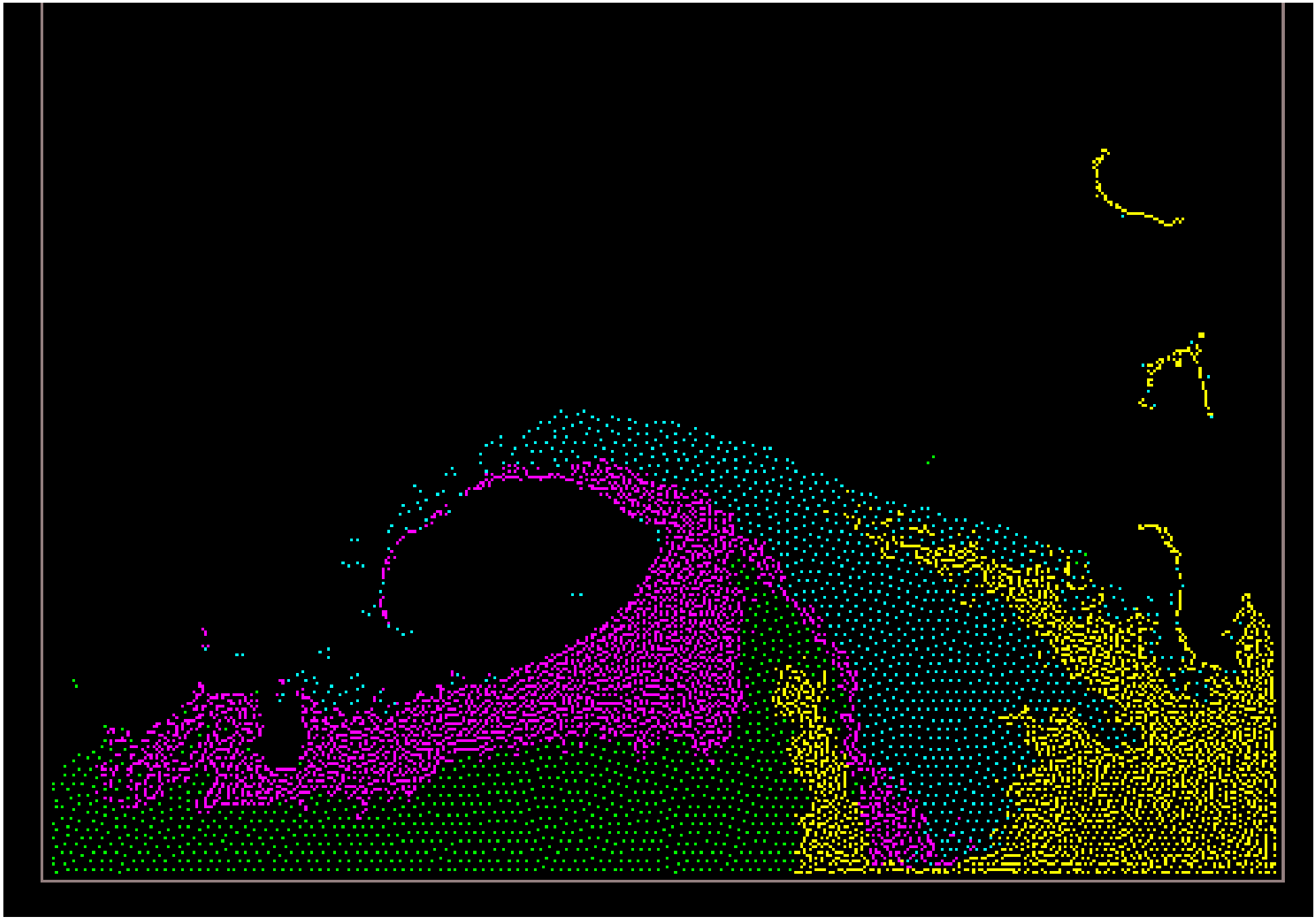
# Vergleich mit naivem Ansatz

- Parallelisierung über Partikel
  - Symmetrie nicht ausgenutzt ($2r$ Relationen berechnet)
  - Nur wo Partikel sind wird auch gerechnet
  - $n * 3^d$ Lookups in Zelle->Partikel Mapping
- Parallelisierung über Gebiet
  - Symmetrie genutzt ($r$ Relationen berechnet)
  - Alle Einflüsse zwischen zwei Rasterzellen auf einen Schwung berechnet
    - gut für Cache
  - In jedem Durchlauf Lookup jeder Raster-Zelle und aller Partikel
    - schlecht für Cache
    - $2^d * 2^{md} = 2^{d(m+1)}$ Lookups insgesamt

# Initialisierung von Partikeln

- Aufgabe: Erstellen von Szenarien
- Lade Randpartikel aus Datei (3 Koordinaten je Partikel)
  - Generiert mit Skript oder Modeler (Blender via X3D Export)
- Wasser: Angabe von num „Gebieten“
  - Je Gebiet Angabe von Gebiet (zwei Punkte), Anzahl Partikel, Dichte, Geschwindigkeit, Epsilon
    - Kollisionen, Explosionen
    - Flächen- und Linienbelegungen (entlang Achsen)
  - Bilde equidistantes Gitter mit Störung durch Epsilon
  - Masse = Volumen * Dichte / Anzahl
  - Ermöglicht unterschiedlich hoch „aufgelöste“ Wasser
    - dadurch Partikel mit verschiedener Masse!
  - Eine „Farbe“ je Gebiet

# Verschiedene Auflösungen



# Initialisierung von Partikeln

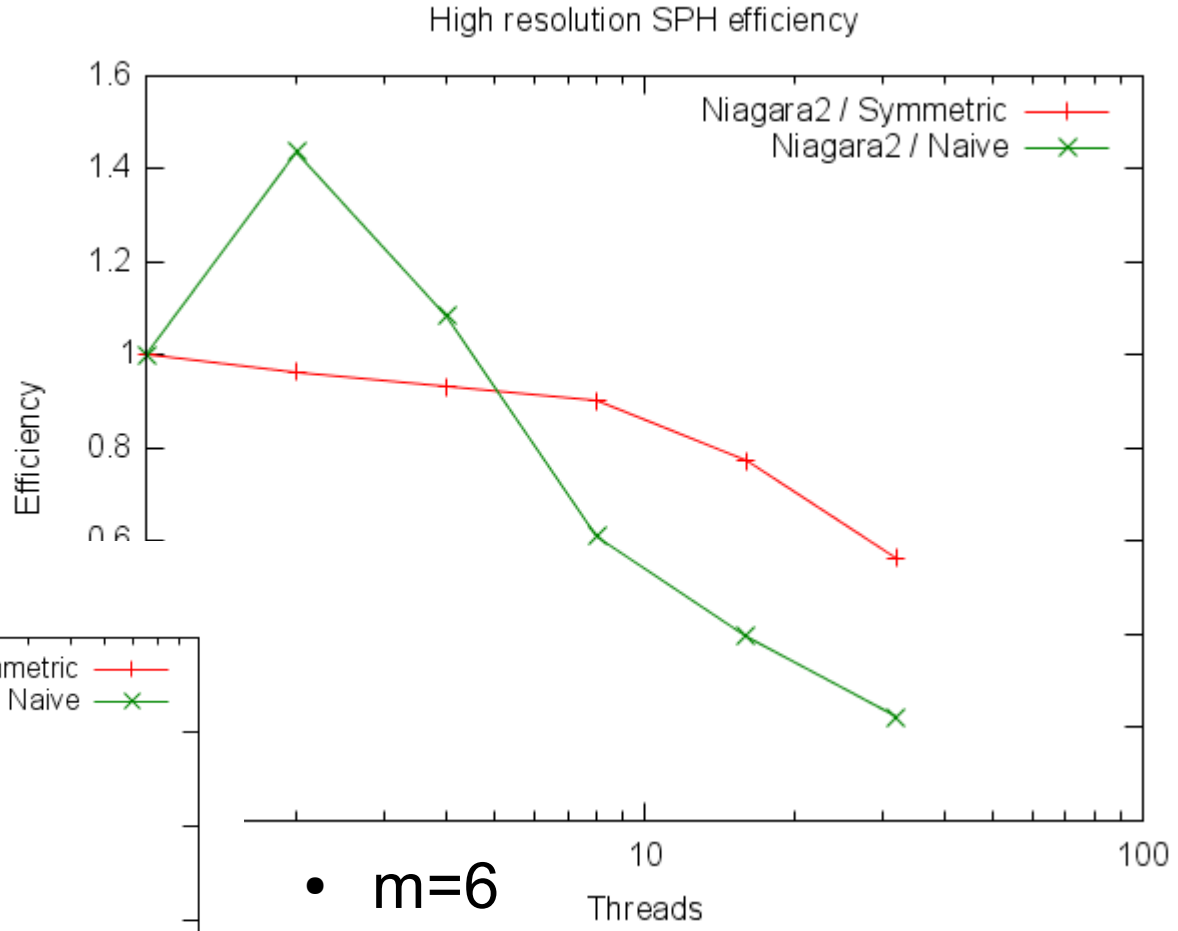
- Weiterer Parameter je „Gebiet“ *i*: Free
  - freischwebend ja oder nein
- Ja: Dichte(Partikel) = Dichte(Gebiet)
- Nein: Dichte(Partikel) = Dichte(Tiefe, Gebiet)
  - Löse $$P(\rho) = \rho_{0,i}\, g\, h$$
    $$B\left(\left(\frac{\rho}{\rho_{0,i}}\right)^{\gamma} - 1\right) = A = \rho_{0,i}\, g\, h$$
  - Maxima sez: $$\rho = \left(\frac{\rho_{0,i}^{\gamma} \overbrace{\rho_{0,i}\, g\, h}^{A}}{B} + \rho_{0,i}^{\gamma}\right)^{\frac{1}{\gamma}}$$
    - Anders als bei Monaghan
  - Tiefe abhängig von Gitterposition und Schwerkraft
    - Todo: relativ zu definiertem „Boden“ (Stapeln von Gebieten in Gefäß)

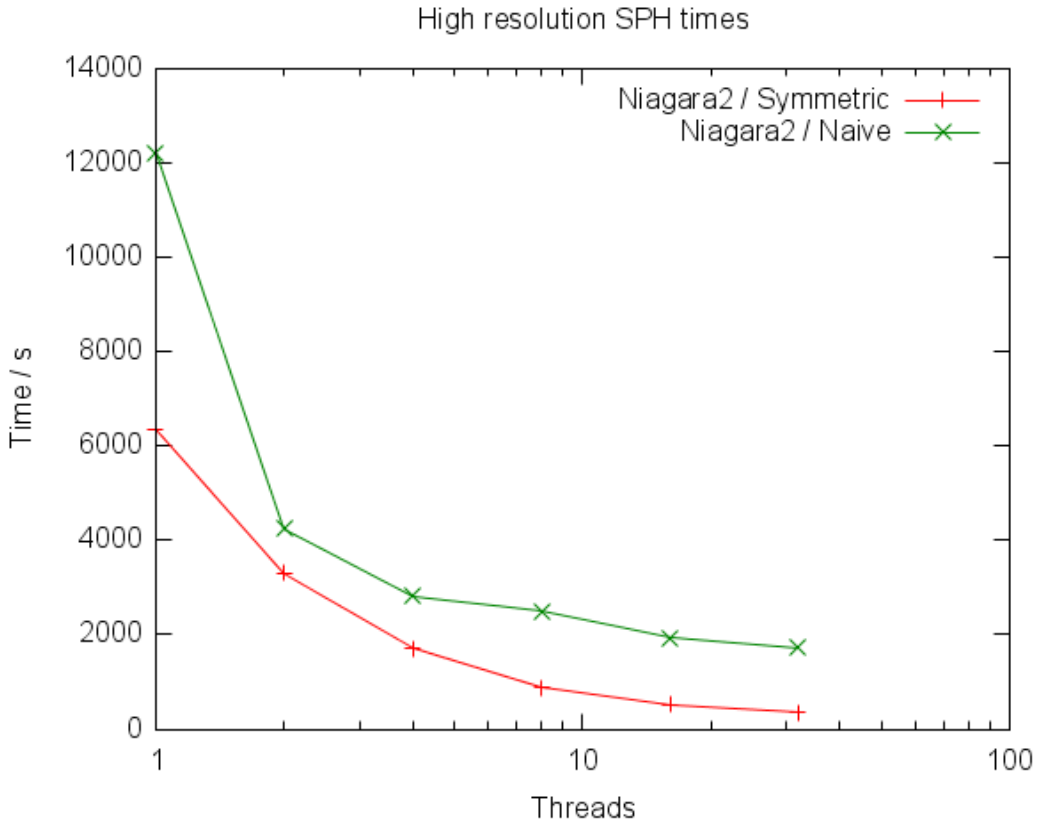
# Einfluß des Hashings auf Distanztests

- 2D, 2460 Randpartikel, H=0,01, Omega=1 m²
  - Testfall settings/schwapp-2d.sh
  - 4096 Partikel verteilt auf 0,4 $m^2$:
    - m=4: 2,47 Relationen/Partikel, 73 Distanztests/Partikel
    - m=5: 2,47 Rels/P, 17,6 Dtests/P
    - m=6: 2,47 Rels/P, 5,3 Dtests/P
    - m=7: 2,47 Rels/P, 4,2 Dtests/P
  - 16384 Partikel verteilt auf 0,4 $m^2$:
    - m=4: 15,3 Relationen/Partikel, 289 Distanztests/Partikel
    - m=5: 15,3 Rels/P, 70,2 Dtests/P
    - m=6: 15,3 Rels/P, 24,4 Dtests/P
    - m=7: 15,3 Rels/P, 23 Dtests/P

# Leistung: Zeiten und Effizienz

- 3D
- PK-Integrator
- Partikel n = 1,3 Mio.
- Davon 185 000 Rand
- 50 Zeitschritte




- m=6
- H=2,5 mm
- ca. 7 Rels/P, 100 Dtests/P
- Zellen $n_H$= 64 Mio
- Speicher 1,2 GB

# Zusammenfassung

- Funktionierender SPH Kode
  - Todo: physikalische Kalibrierung / realistische Tests
  - Todo: Quellen/Injektion von Partikeln
  - Vorbereitet für weitere Phasen / Partikelarten
- SortedParticleIndex effiziente Indexstruktur
  - Todo: Vergleich mit anderen Indexstrukturen
- Hashing reduziert Speicherplatz
- Gebietsparallelisierung spart Rechenaufwand und skaliert besser als naive Summation