# Integercodierte adaptive CFK-Gitter in zwei Dimensionen

## Mathematische Konstruktion, reproduzierbare Teilchenversuche und Diagnose der Parallelisierung

Forschungsprotokoll des Projekts CFK

Versuchsserie und Berichtsstand: 16. September 2026

Untersucht wird ein binärer Bisektionsbaum auf einem rechtwinklig-gleichschenkligen Ursprungsdreieck. Positive Integer codieren die gesamte Geometrie. Aus Teilchenkoordinaten werden notwendige Teilungen und anschließend der kleinste konforme Baum mit höchstens einem zugeordneten Teilchen je Blatt bestimmt. Die Referenzserie umfasst 125 Punktmengen mit 10 bis 100 000 Teilchen. Eine Integerimplementierung reproduziert sämtliche Gitter und reduziert bei 100 000 Teilchen die mittlere Aufbauzeit von 18,064 auf 3,876 s. Zwölf Prozesse allein verbessern dieses Ergebnis nicht. Eine anschließende Diagnose mit 60 Messungen zeigt den Nutzen verteilter Konformitätsabschlüsse, aber auch den dominanten seriellen Aufwand der Eingabenormalisierung. Vier vorbereitete Prozesse benötigen explorativ 2,770 s. Die Millioner-Serie wurde ausdrücklich nicht begonnen; die ursprünglich gesuchte Ein-Minuten-Grenze ist somit nicht bestimmt. Definitionen, Beweise, Messgrenzen, Rohdaten und ausführbare Quellen werden gemeinsam archiviert. Dieser Bericht dokumentiert einen Python-Prototyp, keine bereits validierte CFD-Anwendung.

# Inhaltsverzeichnis

# 1 Gegenstand und Abgrenzung

Die Arbeit folgt der Idee adaptiver Coxeter–Freudenthal–Kuhn-Triangulierungen (CFK) und der mit Bisektion verknüpften $J_1$-Struktur des vorliegenden Manuskripts von Persiano, Comba und Barbalho [1]. Die Quelle behandelt allgemeinere Konstruktionen. Hier wird ausschließlich eine feste zweidimensionale Hierarchie implementiert. Insbesondere werden Wurzelgebiet, Eckreihenfolge, führendes Bit und die Teilchen-Zuordnungsregel explizit festgelegt; diese Konventionen sind Teil unserer Implementierung und nicht als wörtlich übernommene Definitionen des Manuskripts zu verstehen. Alle Aussagen zur Minimalität beziehen sich auf *diese* Hierarchie, nicht auf sämtliche möglichen Triangulierungen einer Punktmenge.

Das lokale PDF trägt den Dateinamen `adptri93.pdf`. Ein bibliographisch gesichertes Publikationsjahr oder Publikationsorgan wird daraus nicht abgeleitet. Das Original wird mit Prüfsumme archiviert. Der Bericht unterscheidet geometrische Definitionen, beweisbare Eigenschaften und empirische Laufzeitbefunde.

## 1.1 Einordnung in CFK und $J_1$

Die reguläre CFK-Konstruktion zerlegt kartesische Würfel in Simplizes. In zwei Dimensionen ist ein solcher Würfel ein Quadrat, das durch eine Diagonale in zwei Dreiecke geteilt wird. Gleichgerichtetes Verschieben der Grundzerlegung liefert die im Manuskript als $K_1$ bezeichnete Anordnung; Spiegeln an gemeinsamen Quadratseiten liefert $J_1$. Die Diagonalrichtungen benachbarter Quadrate wechseln dabei. Die Implementierung beginnt mit nur einem der beiden Dreiecke und braucht deshalb keinen Wald mit mehreren Wurzeln. Sie stellt einen Ausschnitt der durch Spiegelung kompatibel fortsetzbaren Hierarchie dar.

Bei gleichmäßiger Bisektion aller Dreiecke wechselt in zwei Dimensionen die $J_1$-Anordnung zur um 45 Grad gedrehten $S_1$-Anordnung; eine weitere gleichmäßige Bisektion ergibt wieder ein feineres $J_1$-Gitter [1, Abschnitte 1–3]. Die in drei Dimensionen zusätzlich auftretende $R_1$-Stufe und die drei Tetraederformen sind nicht Bestandteil unserer 2D-Implementierung. Ein adaptiver Baum enthält Zellen verschiedener Tiefen; er ist daher kein einzelnes reguläres $J_1$-Gitter, sondern eine konforme lokale Verfeinerung dieser Hierarchie.

# 2 Geometrie aus einem Integer

## 2.1 Wurzel und orientierte Bisektion

Das Rechengebiet ist

$$T_1 = \mathrm{conv}\{A = (0,0), B = (2,0), C = (0,2)\}.$$

Es enthält das Einheitsquadrat $Q = [0,1]^2$, ist aber nicht mit ihm identisch: $|T_1| = 2$, $|Q| = 1$. Die Teilchenversuche besetzen nur $Q$; auch der unbesetzte Rest des Wurzeldreiecks bleibt Bestandteil des Gitters. Wir speichern die Ecken gegen den Uhrzeigersinn, mit rechtem Winkel an $A$. Mit $M = (B + C)/2$ lauten die geordneten Kinder

$$T_{2k} = (M, A, B), \qquad T_{2k+1} = (M, C, A). \tag{1}$$

Damit bleibt die Eckkonvention unter jeder Bisektion erhalten. Es handelt sich stets um die Halbierung der Hypotenuse, also der längsten Kante. Alle entstehenden Dreiecke sind ähnlich; die $J_1$-Struktur beschreibt die Anordnung und Verfeinerung, nicht eine Vielfalt von Dreiecksformen.

## 2.2 Codierung

Die Wurzel erhält Code 1. Die nachfolgenden Bits sind die Entscheidungen in (1). Für $k = (1b_1 \cdots b_d)_2$ gilt

$$d(k) = \lfloor \log_2 k \rfloor = \texttt{bit\_length}(k) - 1, \quad \text{parent}(k) = k \gg 1, \quad \text{child}_b(k) = (k \ll 1) \mid b.$$

Für $k > 1$ ist das Geschwister $k \mathbin{\hat{}} 1$. Die Wurzel hat weder Elternteil noch Geschwister innerhalb des Gebiets. Der Code bestimmt eine Zelle der impliziten Hierarchie unabhängig davon, ob diese Zelle in einem konkreten adaptiven Baum existiert. Ein Vorfahr auf Tiefe $j \leq d$ ergibt sich durch Rechtsverschiebung um $d - j$ Bits. So sind Präfixbeziehungen ohne Geometrieprüfung entscheidbar.

Die Eckkoordinaten werden durch wiederholte Anwendung von (1) rekonstruiert. Sie sind dyadisch rational und werden mit `Fraction` exakt dargestellt. Bei Tiefe $d$ gelten

$$|T_k| = 2^{1-d}, \qquad \ell_{\text{Kathete}} = 2^{1-d/2}, \qquad \ell_{\text{Hypotenuse}} = 2^{3/2-d/2}. \tag{2}$$

Die lokalen Kanten $0, 1, 2$ sind $BC, CA, AB$. Exakte quadrierte Seitenlängen sind $(2^{3-d}, 2^{2-d}, 2^{2-d})$. Irrationale Längen werden nur für die Ausgabe als Gleitkommazahlen berechnet; extrem tiefe Codes können dabei Unterlauf verursachen. Eine physikalische Skalierung $x_{\text{phys}} = sx + x_0$, $s > 0$, multipliziert Längen mit $s$ und Flächen mit $s^2$. Eine allgemeine anisotrope affine Abbildung erhält die kombinatorische Konformität, aber nicht die genannten Längen- und Ähnlichkeitseigenschaften.

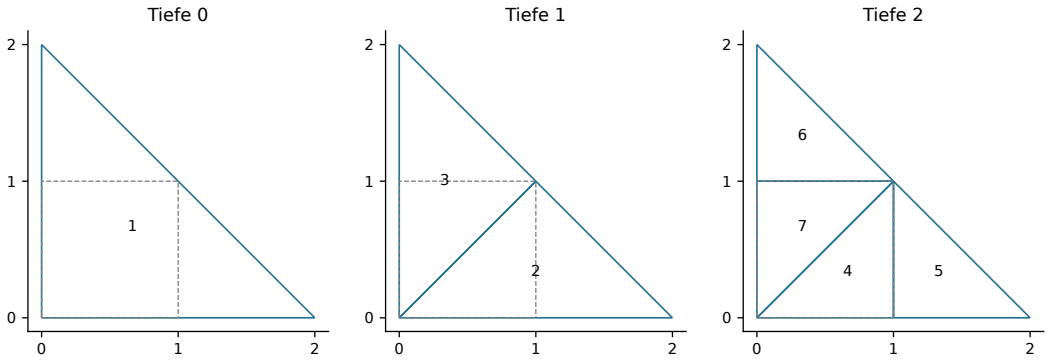


Abbildung 1: Die ersten drei Ebenen der festgelegten Hierarchie. Codes stehen in den Zellschwerpunkten; das gestrichelte Quadrat ist das Teilchengebiet. Es ist keine zusätzliche Randbedingung des Gitters.

## 2.3 Nachbarn ohne gespeichertes Gitter

$\nu_e(k)$ bezeichnet den Nachbarn über Kante $e$ im vollständig gleich tief verfeinerten Wurzelgebiet, beziehungsweise $\perp$ am äußeren Rand. Die Funktion benötigt allein $k, e$. Folgende Beziehungen erlauben den Aufstieg zur ersten Geschwisterkante und den anschließenden Abstieg:

| Kindbit | Kindkante | Beziehung |
|---|---|---|
| 0 | 0 | vollständige Elternkante 2 |
| 0 | 1 | Hälfte der Elternkante 0 nahe Eltern-Ecke $B$ |
| 0 | 2 | Geschwisterkante 1 |
| 1 | 0 | vollständige Elternkante 1 |
| 1 | 1 | Geschwisterkante 2 |
| 1 | 2 | Hälfte der Elternkante 0 nahe Eltern-Ecke $C$ |

Beim Abstieg muss die gegenüberliegende lokale Kante mitgeführt werden. Die gegenläufige Orientierung gemeinsamer Kanten vertauscht die beiden Hälften. Genau diese Information speichert der kleine Arbeitsstapel in `neighbor`. Die Zahl der Auf-/Abstiegsschritte ist $O(d)$. Diese Aussage behandelt Integeroperationen als elementar; ihre tatsächlichen Kosten wachsen bei beliebig langen Codes mit der Bitlänge.

**Wichtige Schnittstellenabgrenzung:** $\nu_e(k)$ ist nicht automatisch ein aktives Blatt im adaptiven Netz. Für spätere Flussberechnungen ist eine eigene Abfrage aktiver Kantennachbarn erforderlich. Eltern-/Nachfahrenbeziehungen müssen dabei mit dem gespeicherten Baum abgeglichen werden. Insbesondere darf ein nicht existierender gleich tiefer Code nicht als vorhandene Zelle behandelt werden.

# 3 Baum, Konformität und minimale Abschlüsse

## 3.1 Knoten, Teilungen und Blätter

Ein vollständiger binärer Baum enthält mit einem geteilten Knoten stets beide Kinder. Sei $I$ die Menge seiner inneren, also geteilten Knoten. Dann sind

$$N(I) = \{1\} \cup \{2k, 2k+1 : k \in I\}, \qquad L(I) = N(I) \setminus I.$$

Damit dies einen Baum beschreibt, muss $I$ unter Elternbildung abgeschlossen sein. Für endliche Bäume gelten

$$|L| = |I| + 1, \qquad |N| = 2|L| - 1, \qquad \sum_{k \in L} 2^{1-d(k)} = 2. \tag{3}$$

Die Flächenidentität folgt induktiv aus der flächentreuen Bisektion. Sie allein prüft weder Überlappungsfreiheit noch Konformität.

## 3.2 Abhängigkeiten der Bisektion

Ein konformes Dreiecksgitter besitzt keine hängenden Knoten: Zwei Zellen schneiden sich nur in einer gemeinsamen vollständigen Kante, einer gemeinsamen Ecke oder gar nicht. Innerhalb dieser kompatiblen Bisektionshierarchie lässt sich das durch folgende Regeln erzwingen:

$$k \in I,\ k > 1 \implies \lfloor k/2 \rfloor \in I, \qquad k \in I,\ \nu_0(k) \neq \perp \implies \nu_0(k) \in I. \tag{4}$$

Die erste Regel erzeugt notwendige Vorfahren; die zweite teilt die gegenüberliegende Hypotenuse mit.

**Lemma 3.1** *Ein vollständiger endlicher Baum dieser Hierarchie ist genau dann konform, wenn seine Teilungsmenge die zweite Regel in* (4) *erfüllt.*

*Beweis.* Eine Bisektion erzeugt nur einen neuen Kantenpunkt, den Hypotenusenmittelpunkt. An einer inneren Kante muss die gegenüberliegende Seite diesen Punkt ebenfalls erzeugen. In der festen Hierarchie geschieht dies durch Teilung des gleich tiefen Hypotenusenpartners; eventuell fehlende Vorfahren sind zuvor zu erzeugen. Umgekehrt kann man die endliche Teilungsmenge nach Tiefe ordnen und innere Hypotenusenpartner jeweils gemeinsam teilen. Jeder Schritt erhält Konformität; Randhypotenusen benötigen keinen Partner. Elternregeln sichern, dass die zu teilenden Dreiecke vorhanden sind. □

Für eine endliche Anforderungsmenge $S$ bezeichnet $\mathcal{C}(S)$ den kleinsten Abschluss unter beiden Regeln. Eine Arbeitsliste fügt neue Anforderungen so lange hinzu, bis keine mehr fehlen. Eltern

verringern die Tiefe, Hypotenusenpartner erhalten sie; deshalb bleibt der Vorgang in der endlichen Menge aller Codes bis $\max_{k\in S} d(k)$. Die entstehenden Blätter können eine Ebene tiefer liegen. Es gilt $\mathcal{C}(\varnothing) = \varnothing$, der zugehörige Baum enthält nur die Wurzel.

**Satz 3.2 (Minimalität)** *$N(\mathcal{C}(S))$ ist der eindeutige kleinste konforme Baum, der alle Codes aus $S$ teilt, geordnet nach Inklusion seiner Knotenmenge.*

*Beweis.* Jeder zulässige Baum muss jede durch (4) erzwungene Teilung enthalten. Die Arbeitsliste fügt ausschließlich solche Teilungen hinzu. Ihr Fixpunkt ist selbst abgeschlossen und damit zulässig. □

### 3.3 Warum globale Teilbäume vereinigt werden dürfen

**Satz 3.3 (Vereinigungseigenschaft)** *Für endlich viele Anforderungsmengen gilt*

$$\mathcal{C}\left(\bigcup_i S_i\right) = \bigcup_i \mathcal{C}(S_i). \tag{5}$$

*Beweis.* Die Regeln haben jeweils genau eine Voraussetzung $k \in I$. Ist $k$ in einer Vereinigung abgeschlossener Mengen enthalten, so liegt es in mindestens einer dieser Mengen, die bereits alle von $k$ erzwungenen Codes enthält. Die Vereinigung ist deshalb abgeschlossen. Minimalität und Monotonie des Abschlusses liefern beide Inklusionen. □

Daraus folgt

$$N\left(\mathcal{C}\left(\bigcup_i S_i\right)\right) = \bigcup_i N(\mathcal{C}(S_i)).$$

Die Arbeiter können also aus ihren Anforderungen *global* konforme Knotenmengen berechnen; die Mengenvereinigung benötigt keinen weiteren Konformitätsabschluss. Die Teilmengen überlappen und dürfen nicht als disjunkte geometrische Teilgebiete verstanden werden. Ein nur an einem Bucket-Rand abgeschnittenes, lokal konformes Netz erfüllt die Voraussetzung nicht. Ebenso genügt nicht die Vereinigung der Blattmengen, da ein Blatt eines Teilbaums im anderen bereits geteilt sein kann.

Grobe gemeinsame Teilungsanforderungen müssen zusätzlich berücksichtigt werden. Eine paarweise Merge-Hierarchie ist mathematisch möglich (Assoziativität der Vereinigung), wurde jedoch nicht implementiert: zusätzliche Transfers und Kopien könnten den relativ kleinen zentralen Merge verteuern. Zuerst wird dessen Anteil gemessen.

## 4 Teilchen bestimmen notwendige Präfixe

### 4.1 Eindeutige Zuordnung und Minimalitätsziel

Es seien $P = \{p_1, \ldots, p_n\} \subset T_1$ paarweise verschiedene Punkte. Geometrisch geschlossene Dreiecke überlappen auf ihren Rändern; deshalb braucht „höchstens ein Teilchen pro Blatt“ eine eindeutige Besitzregel. Mit lokalen baryzentrischen Koordinaten

$$p = (1-b-c)A + bB + cC, \qquad b, c \geq 0, \quad b + c \leq 1$$

wird bei $b \geq c$ Kind 0 gewählt, sonst Kind 1. Randgleichheit gehört somit immer Kind 0. Diese Regel wird bei Codeerzeugung und Blattsuche identisch verwendet. Für die Wurzel ist $b = x/2, c = y/2$.

Sei $P_k$ die Menge der Punkte, deren unendlicher Entscheidungspfad Präfix $k$ besitzt. Notwendig sind genau die Teilungen

$$S(P) = \{k : |P_k| \geq 2\}. \tag{6}$$

Dies ist eine koordinatenbasierte Präfixkonstruktion, kein wiederholtes Durchsuchen eines bereits erzeugten Gitters. Intern werden Punktgruppen dennoch nach Präfixen partitioniert und ihre Größe geprüft; die Bedingung $|P_k| \geq 2$ wird nicht umgangen.

**Satz 4.1** *Für endlich viele verschiedene Punkte ist $S(P)$ endlich. Der kleinste konforme Baum mit höchstens einem zugeordneten Punkt pro Blatt ist $N(\mathcal{C}(S(P)))$.*

*Beweis.* Die Dreiecksdurchmesser gehen nach (2) gegen null. Für endlich viele verschiedene Punkte ist der kleinste paarweise Abstand positiv. Ab hinreichender Tiefe kann kein Dreieck zwei Punkte enthalten, also endet die Präfixkonstruktion. Jeder Präfix mit mindestens zwei Punkten muss geteilt werden; weitere konforme Verfeinerung kann die Belegung nur verringern. Der minimale Abschluss liefert daher genau den gesuchten Baum. □

Identische Punkte lassen sich durch keine endliche Verfeinerung trennen und werden als ungültige Eingabe zurückgewiesen. Eine gesetzte Maximaltiefe führt bei weiterhin mehrfach besetzter Zelle zu einem expliziten Fehler, nicht zu einem stillschweigend unzulässigen Netz. Sehr nahe Punkte können große Tiefen erzwingen; $n$ allein beschränkt die Arbeit nicht scharf.

### 4.2 Transformation ohne geometrische Tests

Die baryzentrischen Koordinaten im gewählten Kind sind

$$(b', c') = \begin{cases} (1 - b - c,\ b - c), & b \geq c, \\ (c - b,\ 1 - b - c), & b < c. \end{cases} \tag{7}$$

Einsetzen der Kinderecken aus (1) beweist die Formeln. Die neue Koordinate an der rechten Ecke ist entsprechend $2c$ bzw. $2b$. Es werden weder Punkt-in-Dreieck-Determinanten noch trigonometrische Operationen benötigt.

In der Referenzversion sind $b, c$ rationale Zahlen. Die schnelle Version setzt für $x = u/r, y = v/s$, $q = \mathrm{kgV}(r, s)$,

$$D = 2q, \qquad B = u(q/r), \qquad C = v(q/s), \qquad b = B/D, \quad c = C/D.$$

Der Nenner $D$ bleibt entlang des gesamten Pfades dieses Punktes konstant. Gleichung (7) wird zu

$$(B', C', D') = \begin{cases} (D - B - C,\ B - C,\ D), & B \geq C, \\ (C - B,\ D - B - C,\ D), & B < C. \end{cases} \tag{8}$$

Das spart wiederholte rationale Kürzungen. Es ist kein gemeinsamer Nenner für alle Teilchen nötig. Binäre Gleitkommaeingaben werden als ihre exakt gespeicherten rationalen Werte interpretiert, nicht als idealisierte Dezimalzahlen. Geometrische Exaktheit bezieht sich auf diese Eingaben.

## 5 Implementierung und Schnittstellen

### 5.1 Schichten und Datenmodell

Die Geometrie ist eine reine Funktion des Codes. Der Baum speichert `set[int]` für alle Knoten, einschließlich Wurzel und innerer Knoten. Ein explizites Objekt mit Zeigern für jedes Dreieck

ist nicht erforderlich. Die öffentliche Schnittstelle liefert unveränderliche Momentaufnahmen als `frozenset`; diese erzeugen Kopien und sind daher nicht kostenlos. Blätter werden durch einen Durchlauf über alle Knoten bestimmt. Die Teilchenzuordnung liegt getrennt von der Geometrie.

| Funktion/Objekt | Bedeutung und Einschränkung |
|---|---|
| `depth(k)` | Tiefe; positive Python-Integer, keine Bool-Werte. |
| `vertices(k)` | Drei exakte rationale Ecken, gegen den Uhrzeigersinn, rechter Winkel zuerst. |
| `area(k)` | Exakte rationale Fläche. |
| `squared_side_lengths(k)` | Exakte Quadrate der Längen in Kantenreihenfolge $0, 1, 2$. |
| `side_lengths(k)` | Gleitkommalängen, nicht für exakte Entscheidungen vorgesehen. |
| `neighbor(k,e)`, `neighbors(k)` | Gleich tiefe implizite Nachbarn, am Wurzelrand `None`; keine Blattabfrage. |
| `TriangleTree(nodes)` | Prüft einen vollständigen konformen Baum bei der Konstruktion. |
| `TriangleTree.from_splits(S)` | Minimaler konformer Baum, der alle Anforderungen teilt; Anforderungen dürfen zunächst fehlen. |
| `tree.refine(k)` | Teilt vorhandenen Knoten samt notwendigen Partnern; gibt sortierte neu geteilte Codes zurück. Bereits geteilter Knoten: keine Änderung. |
| `tree.nodes`, `tree.leaves` | Unveränderliche Momentaufnahmen; Blattberechnung scannt den Baum. |
| `tree.is_leaf(k)` | Blattstatus eines existierenden Knotens; fehlender Code ist ein Fehler. |
| `tree.validate()` | Vollständige Struktur- und Konformitätsprüfung anhand der Integerregeln. |
| `point_code(p,d)` | Code auf festgelegter Tiefe, einschließlich eindeutiger Randzuordnung. |
| `required_splits(points)` | Notwendige Teilungen $S(P)$, noch ohne zusätzlichen Konformitätsabschluss. |
| `locate_point(p,tree)` | Zugeordnetes aktives Blatt eines vorhandenen Baums. |
| `build_particle_mesh(points)` | Baum und Teilchen-Blatt-Zuordnung; Referenz- und Integerfassung mit gleicher Bedeutung. |
| `build_profiled(...)` | Instrumentierte Diagnosefassung; Rückgabe von Knotenmengen, Anforderungen, Phasen und Jobdaten. Nicht dieselbe Rückgabestruktur wie `ParticleMesh`. |
| `warm_pool(workers)` | Bereitet Prozesse vor, wartet auf alle Bereitschaftsmeldungen und liefert Pool samt Startzeit. Aufrufer muss den Pool schließen. |

## 5.2 Kernalgorithmen

Die Nachbarsuche benötigt keine gespeicherten Koordinaten. Ihr vollständiger, unverändert archivierter Quelltext steht in `code/geometry.py`. Die folgenden Ausschnitte sind direkt aus dem archivierten Quelltext eingebunden, damit Bericht und tatsächliche Implementierung nicht auseinanderlaufen:

Der Konformitätsabschluss benutzt eine Arbeitsliste und eine Menge bereits geplanter Teilungen. Fehlende Vorfahren und Partner werden hinzugefügt. Erst nach Planung wird der Baum erweitert, sodass abgeschlossene öffentliche Operationen einen konformen Zustand hinterlassen. Die nachträgliche Sortierung bestimmt eine reproduzierbare Rückgabereihenfolge, ist für die mathematische

```python
    descent: list[tuple[bool, int]] = []
    while code != 1:
        child = code & 1
        sibling_edge = 2 if child == 0 else 1
        if edge == sibling_edge:
            opposite = code ^ 1
            opposite_edge = 1 if child == 0 else 2
            break

        # Child hypotenuse is a full parent leg. Its other outer edge
        # is one half of the parent hypotenuse.
        full_leg = edge == 0
        descent.append((full_leg, child))
        edge = (2 if child == 0 else 1) if full_leg else 0
        code >>= 1
    else:
        return None

    for full_leg, child in reversed(descent):
        if full_leg:
            # Parent edge CA belongs to child 1, AB to child 0.
            opposite = 2 * opposite + (1 if opposite_edge == 1 else 0)
            opposite_edge = 0
        else:
            # Opposite CCW triangles traverse their shared edge backwards.
            opposite = 2 * opposite + (1 - child)
            opposite_edge = 2 if child == 0 else 1
    return opposite
```

Listing 1: Integer-Nachbarsuche: Aufstieg bis zur Geschwisterkante und Abstieg.

```python
        pending = list(codes)
        split: set[int] = set()
        while pending:
            current = pending.pop()
            if current in split or 2*current in self._nodes:
                continue
            split.add(current)

            # Creating a missing triangle requires bisecting its parent.
            if current not in self._nodes:
                pending.append(current // 2)
            mate = neighbor(current, 0)
            if mate is not None:
                pending.append(mate)

        result = tuple(sorted(split))
        for current in result:
            self._nodes.update((2*current, 2*current + 1))
        return result
```

Listing 2: Planung und Übernahme zusätzlicher Teilungen.

Mengenoperation aber nicht notwendig. Die Mengen erlauben im üblichen Hashmodell schnelle Mitgliedschaftsabfragen; eine allgemeine konstante Laufzeit für beliebig große Integer wird nicht behauptet.

Die Referenzfassung in `particles.py` verwendet rationale Arithmetik entlang jedes Pfades. `particles_fast.py` übernimmt Validierung und Randkonvention, ersetzt aber die inneren Pfadtransformationen durch (8). Beide Fassungen führen die Eingabe zunächst durch `_points`: Koordinatenkonversion, Endlichkeits- und Gebietsprüfung, Erkennen identischer Punkte. Dieser gemeinsame Schritt ist wichtig für Robustheit, verursacht aber einen großen seriellen Anteil.

## 5.3 Buckets und Prozesse

Die bisherige schnelle Fassung hat bereits geometrische Buckets: Die jeweils größte mehrfach besetzte Präfixgruppe wird geteilt, bis bis zu $4p$ Gruppen vorliegen oder die größte Gruppe höchstens 2048 Punkte besitzt. Ein Heap wählt die größte Gruppe. Die Zahlen bezeichnen Zielwerte, keine Garantie gleicher Laufzeiten. Gleich viele Teilchen können durch unterschiedliche lokale Abstände unterschiedlich viele weitere Teilungen verursachen. Singleton-Gruppen müssen nicht weiterbearbeitet werden. Reine Routing-Präfixe dürfen nicht vorsorglich als zusätzliche Teilungen in das Ergebnis gelangen.

`ProcessPoolExecutor` mit explizitem Startverfahren `spawn` verteilt Aufgaben dynamisch an freie Prozesse [2, 3]. Es sind nicht dauerhaft feste Buckets an bestimmte Kerne gebunden. Ergebnisse werden bisher in Abgabereihenfolge abgeholt. Die Berechnung späterer Jobs kann trotzdem parallel laufen, jedoch kann das Abholen an einem früheren Job warten. Die Daten müssen zwischen Prozessen serialisiert und übertragen werden; Python-Mengen sind keine gemeinsam genutzten Integerarrays. Prozesse umgehen den Interpreter-Lock für diese Python-Berechnung, beseitigen aber weder Transfer- noch Speicheraufwand.

U2 parallelisiert nur die Präfixarbeit und schließt anschließend zentral. U3 untersucht zusätzlich globale Abschlüsse pro Aufgabe gemäß (5). Grobe Anforderungen aus der Bucketbildung werden als eigene zusätzliche Aufgabe geschlossen. Der Hauptprozess vereinigt alle Knotenmengen. Diese Diagnosefassung ist bewusst separat implementiert: Die bestehende Produktionsschnittstelle bleibt unverändert.

## 5.4 Eigenschaften an Zellen und spätere Simulation

Eine getrennte Abbildung `dict[int, CellData]` ist ein geeigneter Ansatz für Materialwerte, Zustandsgrößen oder Kennzeichnungen. Der Code bleibt stabil, solange die Hierarchiekonvention unverändert bleibt. Die Speicherung solcher Simulationsfelder ist noch nicht implementiert. Vor ihrer Einführung sind Lebensdauer und Übertragung bei Verfeinerung festzulegen: Extensive Größen wie Masse müssen konservativ aufgeteilt, intensive Größen wie Dichte geeignet interpoliert werden. Innere Knoten und aktive Blätter dürfen dabei nicht verwechselt werden. Für CFD fehlen außerdem aktive Nachbarschaft, konsistente Flussorientierung, Randbedingungen, Diskretisierung und physikalische Verifikation.

Eine spätere C++-Portierung darf nicht stillschweigend auf 64-Bit-Codes beschränkt werden: Ein Code auf Tiefe $d$ benötigt $d + 1$ Bits. Auch Integerkoordinaten können bei allgemeinen rationalen Eingaben mehr als 64 Bit benötigen. Entweder wird eine explizite unterstützte Tiefe mit Überlaufprüfung festgelegt oder Mehrpräzisionsarithmetik beibehalten. Die Pythonfassung bleibt als exakt rechnende Vergleichsreferenz sinnvoll.

### 5.5 Visualisierung und Tests

`viewer.py` stellt das Gitter mit Matplotlib dar. Es unterstützt Interaktion zur Verfeinerung, Rücknahme und Rücksetzung sowie Teilchenanzeige und statischen Export. Eine GUI-Sitzung konnte in der Ausführungsumgebung nicht geöffnet werden; nichtinteraktive Darstellung und die zugehörigen Tests wurden ausgeführt. Das ist kein Nachweis einer getesteten Desktop-Interaktion.

Der aktuelle Testlauf umfasst 43 erfolgreiche Tests. Geometrische Tests prüfen Eckkonvention, Flächen, Seiten und Nachbarn gegen explizite Geometrie. Baumtests prüfen Vollständigkeit, Konformität und minimale Abschlüsse. Teilchentests umfassen Randpunkte, ungültige und identische Eingaben, Tiefenbeschränkung, Referenzvergleich und parallele Ausführung. Neue Diagnosetests prüfen die Vereinigung globaler Abschlüsse, zentrale/verteilte Ergebnisse und Wiederverwendung vorbereiteter Pools. Die Testquellen und das vollständige Ausführungsprotokoll liegen im Archiv.

Die Benchmarkvalidierung kontrolliert Struktur, Flächensumme, Kennzahlen und exakte Codesignaturen. Große Versuche wiederholen nicht unabhängig für jedes Teilchen eine geometrische Blattsuche. Diese Eigenschaft wird durch Konstruktion und separate Tests abgesichert. Ein Hashvergleich ist ein starker praktischer Gleichheitsnachweis, aber ersetzt keinen mathematischen Beweis oder unabhängige geometrische Referenztests.

## 6 Reproduzierbares Versuchsdesign

### 6.1 Zufallsmodell und Kontrollgrößen

Für $n = 10, 100, 1000, 10\,000, 100\,000$ wurden je 25 unabhängige Punktmengen erzeugt. Die Pseudozufallszahlen sind gleichverteilt auf dem diskreten Raster $\{0, \ldots, 2^{53} - 1\}/2^{53}$; dies approximiert die kontinuierliche Gleichverteilung auf $[0, 1)^2$. Verwendet wird `random.Random` (MT19937) mit Master-Seed 20260916. Für Wiederholung $r \in \{1, \ldots, 25\}$ ist der konkrete Seed

$$s_{n,r} = \mathrm{int}_{\mathrm{big}}\left(\mathrm{SHA256}\left(\texttt{cfk-uniform-v1:20260916:}n\texttt{:}r\right)\right).$$

Die Zeichenfolge verwendet ASCII und die dezimale Darstellung von $n, r$ ohne zusätzliche Leerzeichen. Pro Punkt werden nacheinander zwei Aufrufe von `getrandbits(53)` mit $2^{-53}$ multipliziert. Die vollständigen Dezimal-Seeds stehen in den JSON-Einzelprotokollen. Zufällige Duplikate wären auf dem endlichen Raster prinzipiell möglich; sie würden nicht stillschweigend entfernt, sondern einen Fehler auslösen. In den gemessenen Datensätzen trat dies nicht auf.

Der Eingabehash ist SHA-256 über aufeinanderfolgende Koordinatenpaare als Little-Endian-IEEE-754-Doubles (`struct.pack("<dd",x,y)`). Der Blatthash ist SHA-256 über aufsteigend sortierte dezimale Blattcodes, jeder gefolgt von einem Zeilenumbruch. Der Hash bestätigt somit nicht nur eine ähnliche Statistik, sondern denselben kodierten Baum.

### 6.2 Umgebung und Messfenster

Gemessen wurde unter Linux mit Python 3.13.9 (Anaconda, GCC 11.2.0) auf einem Intel Core Ultra 7 255U. Die Erfassung meldet 12 physische Kerne und 14 logische CPUs. Unterschiedliche Kerntypen und gemeinsame Leistungsgrenzen lassen keine einfache Annahme zwölf gleich schneller, unabhängig skalierender Recheneinheiten zu. Es wurden keine feste CPU-Affinität und keine kontrollierten Takt-, Temperatur- oder Hintergrundlastbedingungen eingerichtet. Vollständige Systemangaben stehen in den Metadaten.

Die primäre Größe ist verstrichene Zeit mit `time.perf_counter`. U1 und U2 messen

$$t_{\mathrm{build}} = t_{\mathrm{required\_splits}} + t_{\mathrm{from\_splits}}.$$

Das schließt Eingabeprüfung und Zahlenkonversion ein, aber nicht Zufallsziehung, Imports, Hashbildung, vorgelagertes `gc.collect`, abschließende Validierung, Kennzahlberechnung, Dateiausgabe oder nachträgliche Teilchen-Blatt-Zuordnung. Die Garbage Collection bleibt während des Messfensters aktiv. Es wird also nicht die gesamte Laufzeit von `build_particle_mesh` gemessen.

U1 startet für jeden Versuch einen frischen Messprozess; ein kleiner, nicht ausgewerteter Vorlauf wärmt Datei-/Importcaches. U2 startet ebenfalls frische Messprozesse und schließt den Start der inneren Arbeiterprozesse in die Aufbauzeit ein. Die Variantenreihenfolge wechselt zwischen Wiederholungen. U1 ist jedoch eine vorherige Messserie, kein gleichzeitig randomisierter Referenzarm. Gepaarte identische Daten kontrollieren Geometrieeffekte, nicht nachträgliche Änderungen der Systemlast.

### 6.3 Studienübersicht und Abweichungen

| Serie | Eingaben | Varianten | Messungen |
|---|---|---|---|
| U1 | fünf Größen, je 25 | rationale Referenz | 125 |
| Pilot | $10^4, 10^5$, je 3 | Integer, $p = 1, 2, 4, 8, 12$ | 30 |
| U2 | identische 125 U1-Eingaben | Integer, $p = 1, 12$ | 250 |
| U3 | $10^5$, U1-Wdh. 1–3 | $5 \times 2 \times 2$ Konfigurationen | 60 |

U1 sollte ursprünglich fortgesetzt werden, bis der Mittelwert einer Größenstufe 60 Sekunden überschreitet. Auf Nutzerwunsch wurde bei 100 000 gestoppt; die nächste Stufe $10^6$ blieb ungemessen. Eine Unterbrechung der ersten Serie ist in `AMENDMENT.md` dokumentiert: Ein noch nicht abgeschlossenes Ergebnis wurde mit demselben Seed erneut berechnet. Fertige Messungen wurden nicht nach Laufzeit verworfen. Der Kontrollzustand bleibt archiviert; die Ein-Minuten-Grenze ist unbekannt.

## 7 Ergebnisse der Referenz und Integerfassung

### 7.1 U1: Größen, Tiefen und Laufzeit

| $n$ | $\bar{t}$ [s] | $s_t$ [s] | $\overline{\lvert L \rvert}$ | Bereich $d_{\max}$ |
|---:|---:|---:|---:|---:|
| 10 | 0,001028 | 0,000289 | 33,32 | 6–11 |
| 100 | 0,008267 | 0,000683 | 364,92 | 12–19 |
| 1000 | 0,111419 | 0,017639 | 3443,68 | 18–24 |
| 10000 | 1,334166 | 0,077415 | 33352,36 | 25–32 |
| 100000 | 18,063919 | 0,262672 | 331324,24 | 32–38 |

Tabelle 2: U1: 25 Wiederholungen pro Größe. Zeit als Mittelwert und Stichprobenstandardabweichung; Tiefenspalte als kleinste bis größte beobachtete maximale Blatttiefe.

Die Blattzahl liegt bei den größeren gemessenen Größen nahe $3{,}3n$. Dies ist ein empirischer Befund für dieses Zufallsmodell einschließlich Konformitätsverfeinerung und unbesetztem Wurzelrest, keine allgemeine asymptotische Konstante. Bei 100 000 Teilchen liegen die maximalen Tiefen zwischen 32 und 38; einzelne nahe Punktpaare beeinflussen die größte Tiefe deutlich stärker als die typische Blattgröße.

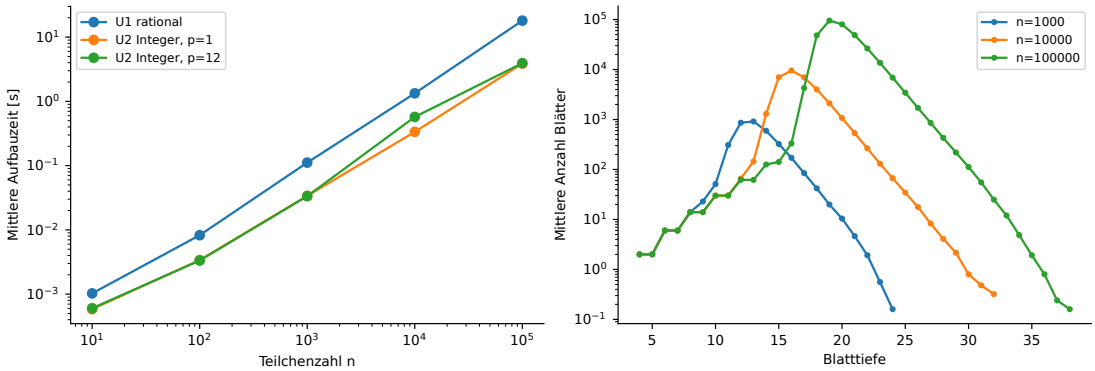


Abbildung 2: Links: mittlere Aufbauzeiten der abgeschlossenen 25er-Serien (logarithmische Achsen). Rechts: mittlere Blattzahlen je Tiefe in U1. Die vollständigen Histogramme jedes Versuchs sind gespeichert.

| $n$ | $p=1$ | $p=2$ | $p=4$ | $p=8$ | $p=12$ |
|---|---|---|---|---|---|
| 10000 | 0,307 | 0,502 | 0,497 | 0,530 | 0,518 |
| 100000 | 3,936 | 4,039 | 3,655 | 3,803 | 3,820 |

Tabelle 3: Pilot: Mittelzeiten in Sekunden, je drei identische Eingaben. Die kleine Stichprobe dient der Vorauswahl, nicht einer belastbaren Rangfolge benachbarter Prozesszahlen.

| $n$ | $p=1$ [s] | $p=12$ [s] | $S$ |
|---|---|---|---|
| 10 | 0,000583 | 0,000603 | 1,76 |
| 100 | 0,003397 | 0,003338 | 2,43 |
| 1000 | 0,033880 | 0,033504 | 3,29 |
| 10000 | 0,335200 | 0,572445 | 3,98 |
| 100000 | 3,875782 | 3,932713 | 4,66 |

Tabelle 4: U2: 25 Wiederholungen je Größe und Prozesszahl. Beschleunigung $S = \bar{t}_{\mathrm{U1}}/\bar{t}_{\mathrm{U2},p=1}$ ist das Verhältnis der Mittelwerte, nicht der Mittelwert einzelner Quotienten.

### 7.2 Pilot und U2: exakte Beschleunigung

Alle 30 Pilot- und 250 U2-Ergebnisse stimmen mit den U1-Referenzen in Eingabehash, Blatthash, Tiefenhistogramm und Teilungskennzahlen überein. Bei 100 000 Teilchen erreicht allein die Integerarithmetik Faktor 4,66. Zwölf Prozesse benötigen im Mittel 3,933 statt 3,876 s, liefern also keinen zusätzlichen Vorteil. Bei 10 000 sind sie mit 0,572 statt 0,335 s deutlich langsamer. Bis 1000 Teilchen greift die serielle Kleingruppenroute; `workers=12` bedeutet dort nicht zwölf tatsächlich gestartete Prozesse.

## 8 U3: Wo bleibt die Parallelität?

### 8.1 Kontrollierter Diagnosevergleich

U3 verwendet ausschließlich die ersten drei U1-Eingaben mit $n = 100\,000$. Faktoren sind $p \in \{1, 2, 4, 8, 12\}$, zentraler oder verteilter Abschluss und kalter oder vorbereiteter Pool. Jede Konfiguration läuft in einem frischen Hauptprozess. Ein vorbereiteter Pool wird für drei Punktmengen wiederverwendet; neue Bäume und Mengen werden für jeden Versuch neu aufgebaut. Die Bereitschaft aller angeforderten Prozesse wird vor der Zeitmessung abgewartet. Start und Ende des Pools sind separat in `sessions/` protokolliert. Bei einem Prozess gibt es keinen Pool, sodass der Unterschied zwischen kalt und vorbereitet dort lediglich unterschiedliche Messläufe ist.

Die Reihenfolge der vier Varianten wechselt zwischen Prozesszahlen, ist aber nicht vollständig randomisiert. Mit drei Wiederholungen ist U3 eine explorative Diagnose, keine neue bestätigende 25er-Studie. Die instrumentierte Fassung hält zusätzliche Ergebnisse und Zeitstempel fest; ihre Zeit ist nicht ohne Vorbehalt mit U2 gleichzusetzen.

Disjunkte Hauptprozessphasen sind Normalisierung, Integerpackung, Bucketbildung, Poolkonstruktion, Abgabe, Warten bzw. serielle Jobausführung, Merge, Shutdown, globaler Abschluss und unveränderliche Ergebnismengen. Ihre Summe entspricht der gemessenen Aufbauzeit. Arbeiterintervalle laufen parallel und dürfen nicht dazuaddiert werden. Vollständige Validierung und Payload-Schätzung erfolgen nach dem Messfenster.

Jeder Job liefert PID, Präfix, Punktzahl, Beginn, Ende, Präfix-/Abschlussdauer, CPU-Zeit und Ergebnisgröße. Wartezeit zwischen Abgabe und Beginn beinhaltet Warteschlange, Start und Transfer; Zeit zwischen Jobende und Abholung beinhaltet Transfer und gegebenenfalls Warten auf frühere Ergebnisse. Keine dieser Größen isoliert die reine IPC-Zeit. Geschätzte serialisierte Größen verwenden `pickle` mit höchstem Protokoll; sie sind keine gemessenen IPC-Bytezähler.

### 8.2 Zeitvergleich und Phasen

| $p$ | Zentral/kalt | Zentral/vorber. | Verteilt/kalt | Verteilt/vorber. |
|---|---|---|---|---|
| 1 | $3{,}768 \pm 0{,}013$ | $3{,}940 \pm 0{,}056$ | $3{,}960 \pm 0{,}143$ | $3{,}951 \pm 0{,}074$ |
| 2 | $3{,}860 \pm 0{,}233$ | $3{,}550 \pm 0{,}039$ | $3{,}377 \pm 0{,}146$ | $3{,}200 \pm 0{,}119$ |
| 4 | $3{,}575 \pm 0{,}043$ | $3{,}601 \pm 0{,}139$ | $2{,}940 \pm 0{,}041$ | $2{,}770 \pm 0{,}015$ |
| 8 | $3{,}854 \pm 0{,}195$ | $3{,}573 \pm 0{,}063$ | $2{,}820 \pm 0{,}045$ | $2{,}842 \pm 0{,}195$ |
| 12 | $3{,}805 \pm 0{,}052$ | $3{,}599 \pm 0{,}040$ | $3{,}009 \pm 0{,}113$ | $2{,}782 \pm 0{,}027$ |

Tabelle 5: U3: Mittelwert ± Stichprobenstandardabweichung in Sekunden, je drei Wiederholungen. Vorbereiteter Pool: Startkosten außerhalb des Messfensters. Alle 60 Resultate stimmen mit den Referenzen überein.

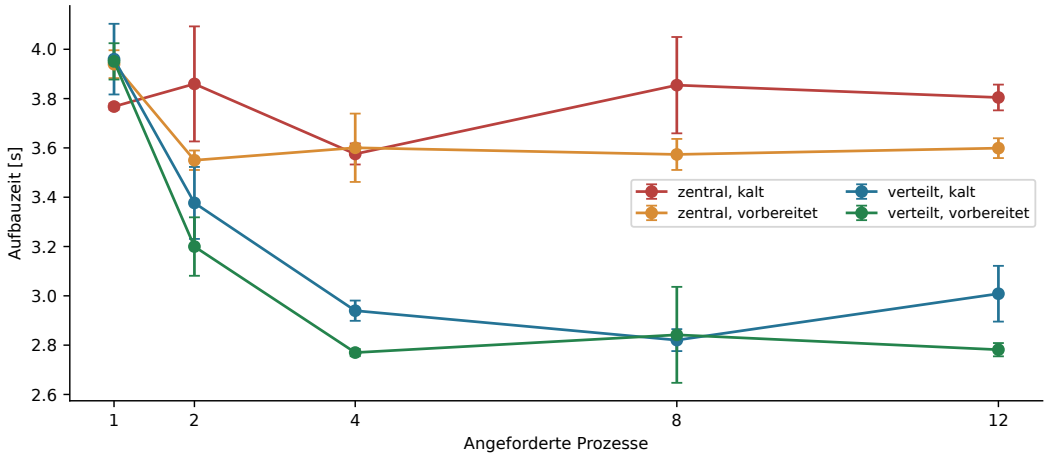


Abbildung 3: U3-Aufbauzeiten. Fehlerbalken sind Standardabweichungen aus drei Wiederholungen, keine Konfidenzintervalle. Der Unterschied bei einem Prozess ist Messstreuung ohne Pool-Effekt.

Bei vier vorbereiteten Prozessen sinkt die Zeit durch verteilten Abschluss von 3,601 auf 2,770 s, also um rund 23 Prozent. Gegenüber der zentralen seriellen Diagnose mit kaltem Etikett (3,768 s) ist dies Faktor 1,36, nicht Faktor vier oder zwölf. Der beste kalte Mittelwert liegt bei acht Prozessen (2,820 s); vorbereitet sind vier und zwölf Prozesse mit 2,770 bzw. 2,782 s praktisch gleichauf. Aus drei Versuchen folgt keine gesicherte optimale Prozesszahl.

| Variante | Norm. | Pack | Bucket | Warten | Merge | Abschluss | Rest |
|---|---|---|---|---|---|---|---|
| Z, 4 | 1,529 | 0,220 | 0,288 | 0,289 | 0,023 | 1,245 | 0,006 |
| V, 4 | 1,505 | 0,228 | 0,272 | 0,586 | 0,137 | 0,000 | 0,041 |
| V, 12 | 1,513 | 0,217 | 0,409 | 0,428 | 0,168 | 0,000 | 0,046 |

Tabelle 6: Ausgewählte mittlere Hauptprozessphasen in Sekunden. Z: zentral, V: verteilt; jeweils vorbereiteter Pool, Zahl: Prozesszahl. Rest fasst die übrigen disjunkten Phasen zusammen. Arbeiterzeiten sind nicht zusätzlich enthalten.

Die Normalisierung allein benötigt ungefähr 1,5 s; Integerpackung zusätzlich etwa 0,22 s. Bei zentralem Abschluss kommen rund 1,2 s seriell hinzu. Damit ist die schwache Skalierung der bisherigen Variante bereits ohne die Annahme besonders schlechter Lastverteilung verständlich.

Die verteilte Variante erhöht den Rückgabepayload: etwa 4,04 MB bei vier vorbereiteten Prozessen gegenüber 0,66 MB zentral. Die Summe zurückgegebener Knotenzahlen beträgt das 1,0485-Fache der endgültigen Knotenzahl; bei zwölf Prozessen steigt dies auf 1,0981. Die globale Konformität erzeugt somit moderate Überlappung, spart aber den großen seriellen Abschluss. Der zentrale Merge liegt bei vier vorbereiteten Prozessen im Mittel bei 0,137 s. Eine parallele Merge-Hierarchie ist deshalb nicht der erste nächste Optimierungsschritt.

## 8.3 Zeitliche Auslastung

Die Zeitstrahlen zeigen, warum ein Systemmonitor während erheblicher Teile des Aufbaus keine hohe parallele Auslastung anzeigt: Die Jobs beginnen erst nach Normalisierung, Packung und Bucketbildung. Innerhalb des Arbeiterabschnitts überlappen mehrere Jobs. Ungleiche Jobdauer, Transfers und Ergebnisabholung bleiben mögliche zusätzliche Ursachen; ihre Anteile lassen sich aus

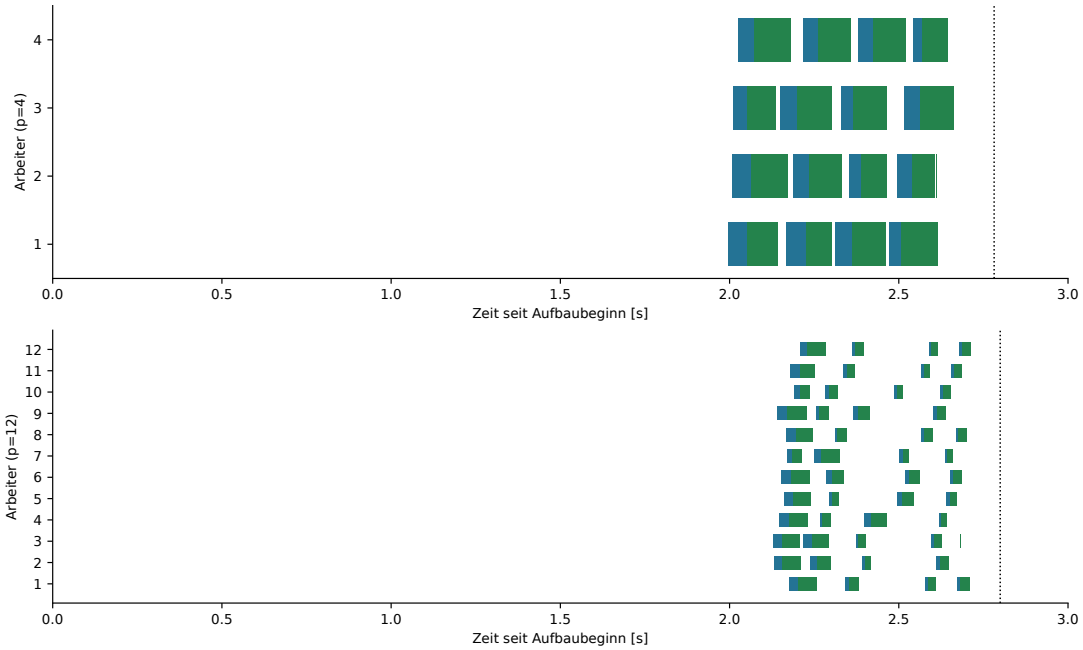


Abbildung 4: Jobintervalle des ersten U3-Versuchs mit verteiltem Abschluss und vorbereitetem Pool, oben vier, unten zwölf Prozesse. Blau: Präfixarbeit; Grün: globaler Abschluss im Job. Die Zeitachse beginnt mit dem gesamten Baumaufbau. Vor den Jobs liegt die serielle Vorbereitung. Leere Abschnitte sind keine isolierte Messung von Betriebssystem-Leerlauf.

dem Monitorbild allein nicht trennen. Die gespeicherten Einzelintervalle erlauben spätere, gezieltere Analysen.

# 9 Bewertung und nächste Entscheidungen

## 9.1 Gesicherte Aussagen und Grenzen

Die Integercodierung liefert eine kompakte, geometrisch vollständige Adressierung. Sie vereinfacht Eltern-, Kinder- und Präfixoperationen. Konformität ist jedoch mehr als das Vorhandensein beliebiger Codes: Die Eltern- und Partnerregeln müssen gemeinsam erfüllt werden. Die mathematische Abschlussstruktur erlaubt die exakte Vereinigung unabhängig global abgeschlossener Anforderungen. Alle untersuchten Implementierungen liefern für die Vergleichsdaten denselben Baum.

Der große Leistungsgewinn von U1 zu U2 entsteht überwiegend durch die Änderung der inneren Arithmetik. Die bisherige Parallelisierung betrifft nur einen Teil der Gesamtkosten. Als Orientierung dient

$$S(p) \leq \frac{1}{f + (1-f)/p},$$

wenn ein Anteil $f$ der seriellen Ausgangsarbeit unverändert seriell bleibt und Zusatzkosten ignoriert werden. Das ist ein Modell, keine hier angepasste Laufzeitformel. Mit Vorbereitung, Übertragung, Überlappung von Abschlüssen, ungleichen Kerntypen und Speicherverkehr ist die reale Situation ungünstiger. Die Erwartung Faktor zwölf allein aus zwölf Kernen ist daher nicht begründet.

Die U3-Phasenmessung zeigt unmittelbar: Ein bloßer anderer Scheduler beseitigt die serielle Normalisierung und den zentralen Abschluss nicht. Der verteilte Abschluss verbessert den zweiten

Punkt nachweislich in diesen Daten. Vier vorbereitete Prozesse sind vorläufig eine vernünftige Diagnosekonfiguration, aber noch kein generell optimaler Standard für jede Teilchenzahl und Verteilung.

### 9.2 Empfohlene Fortsetzung

Als nächster gezielter Eingriff bietet sich ein exakt validierender Direktpfad von binären Floatkoordinaten zu Integertripeln an. Er soll den Umweg über viele `Fraction`-Objekte vermeiden, ohne die allgemeine rationale Referenzschnittstelle abzuschaffen. Gebietsprüfung, Duplikaterkennung und Randentscheidung müssen nachweislich gleich bleiben. Die Verteilung auf Prozesse sollte erst danach erneut vermessen werden, weil sich der relative parallele Anteil verändert.

Anschließend sollte die uninstrumentierte verteilte Variante auf denselben 25 U1-Seeds bestätigt werden, mit gepaarter und möglichst randomisierter Variantenreihenfolge. Vorbereitete Pools sind sinnvoll für viele aufeinander folgende Gitteraufbauten; für Einzelaufrufe muss die kalte Gesamtzeit gelten. Mehr Buckets oder andere Schedulingregeln sind anhand der Jobdauerverteilung zu prüfen, nicht nur anhand der Teilchenzahl. Ein paralleler Merge folgt erst, wenn sein gemessener Anteil den zusätzlichen Aufwand rechtfertigt. Diese weiteren Änderungen wurden in diesem Arbeitsschritt nicht vorgenommen.

Für eine Forschungsarbeit wären außerdem nichtuniforme und adversarielle Punktverteilungen, sehr nahe Paare, Randkonzentration, leere Eingaben, Speicherverbrauch aller Prozesse und Skalierung über mehrere Maschinen zu untersuchen. Maximum und Mittelwert der Tiefe beantworten unterschiedliche Fragen. Konfidenzintervalle für Laufzeitverhältnisse sollten die Paarung der Seeds berücksichtigen; Fehlerbalken dieses Berichts sind ausschließlich beschreibende Standardabweichungen.

Die Übertragung nach C++ ist erst nach Stabilisierung der Semantik sinnvoll. Eine kompaktere Speicherrepräsentation könnte Vorteile bringen, aber die hier gemessenen Zeiten erlauben keine seriöse Vorhersage ihres Faktors. Die Millioner-Serie bleibt ausgesetzt. Es wurde kein Hintergrund- oder Nachtlauf eingerichtet.

## 10 Archiv und Reproduktion

### 10.1 Inhalt des datierten Berichtsordners

Der Ordnername bezeichnet den Beginn dieser Forschungsdokumentation; UTC-Zeitstempel der Einzelversuche sind maßgeblich für deren tatsächliche Ausführung. Die Berichtsquellen verwenden die bereitgestellte `Vorlage.tex`: KOMA-Script, A4, 10 Punkt, deutsche Sprache, Absatzabstand und Kopfzeile bleiben erhalten. Aufgaben-/Lösungsgenerierung, PSTricks-Raster und chemische Notation wurden entfernt. Tabellen, Literaturverwaltung und `listings` ergänzen die wissenschaftliche Dokumentation; `listings` benötigt keine externe Codeausführung.

| Pfad relativ zum Bericht | Inhalt |
|---|---|
| `Bericht.tex`, Kapiteldateien | Lesbarer Haupttext, Formeln, Quelltexteinbindung und Literatur. |
| `Bericht.pdf` | Kompilierte Fassung dieses Berichts. |
| `build_assets.py` | Aus Rohdaten abgeleitete Tabellen und Abbildungen; startet keine Experimente. |
| `data/uniform_v1/` | Alle 125 U1-Protokolle, CSV-Dateien, ursprüngliche Quellen, Metadaten und Ergänzungsprotokoll. |
| `data/integer_pilot/` | Pilotdaten und zugehörige Messquellen. |

| Pfad relativ zum Bericht | Inhalt |
|---|---|
| `data/uniform_v2/` | Alle 250 Vergleichsprotokolle, gepaarte Auswertung, Referenzkopie und Quellen. |
| `data/diagnostic_v3/` | 60 Einzelprotokolle, Jobintervalle, Poolsitzungen, Phasentabellen, Quellen und Metadaten. |
| `code/` | Aktueller Python-Stand einschließlich Tests; Messserien behalten zusätzlich ihre jeweils eigenen Quellkopien. |
| `sources/` | Unveränderte LaTeX-Vorlage und Originalmanuskript. |
| `input_manifest.json` | SHA-256 aller archivierten Daten-, Code- und Quelldateien zum Zeitpunkt der Archivierung. |
| `analysis_check.json` | Prüfstatus der Eingabesignaturen, Phasensummen und 60 exakten U3-Übereinstimmungen; Analyseversionen. |
| `test_results.txt` | Ausgabe des abschließenden Testlaufs. |
| `figures/`, `tables/` | Automatisch erzeugte PDF-/PNG-Abbildungen und LaTeX-Tabellen. |

Abgeleitete Dateien lassen sich neu erzeugen. Primär sind die einzelnen JSON-Protokolle, nicht gerundete Tabellen im PDF. CSV-Dateien ermöglichen eine spätere Analyse ohne Auslesen des Berichts. Laufzeiten sind nicht bitweise reproduzierbar; Eingaben und Gitter dagegen sollen identisch sein. Die Metadaten enthalten zusätzlich die Prüfsummen der tatsächlich zur Messung verwendeten Quellen. Das Archiv ist eine Momentaufnahme, keine vollständig containerisierte Betriebssystemumgebung.

## 10.2 Bericht neu erzeugen

Die folgenden Befehle werden im datierten Berichtsordner ausgeführt. Benötigt werden Python mit Matplotlib und eine LaTeX-Installation mit den im Hauptdokument genannten Paketen, BibTeX und `latexmk`. Der Schalter `-snapshot` dient ausschließlich der erstmaligen Archivierung aus

```
MPLCONFIGDIR=/tmp/cfk-matplotlib python3 build_assets.py
latexmk -pdf -interaction=nonstopmode -halt-on-error Bericht.tex
python3 -m unittest discover -s code -p 'test_*.py' -v
```

Listing 3: Analyse und PDF ohne neue Messserie.

dem Projektverzeichnis. Bestehende Datenordner werden dabei absichtlich nicht ersetzt. Normale Regeneration prüft alle archivierten Eingaben gegen das Manifest und arbeitet nur auf dieser Kopie.

## 10.3 Messungen wiederholen

Für neue Messungen sind neue Ausgabeordner zu wählen; die hier dokumentierten Originaldaten dürfen nicht überschrieben werden. Die archivierten Messquellen werden verwendet, nicht ein später veränderter Entwicklungsstand. Die U1-Steuerung erhält eine explizite Obergrenze; dadurch wird keine Millioner-Serie versehentlich begonnen. Der erste Befehl verwendet die aktuelle U1-Steuerung mit Obergrenzenoption; für eine exakte historische Messimplementierung ist die in den U1-Metadaten referenzierte Quellkopie maßgeblich. Einzelne historische Versuche lassen sich direkt mit `data/uniform_v1/source/benchmark.py -worker N R SEED` ausführen. Die vollständigen Seeds stehen in den Einzelprotokollen. Diese Direktaufrufe schreiben ihre JSON-Ergebnisse auf

```
python3 CFK/2D/benchmark.py \
  --output /tmp/cfk-u1-replay --max-n 100000 \
  --repetitions 25 --master-seed 20260916

python3 CFK/Berichte/2026-09-16/data/diagnostic_v3/source/benchmark_diagnostic.py \
  --baseline CFK/Berichte/2026-09-16/data/uniform_v1 \
  --output /tmp/cfk-u3-replay
```

Listing 4: Beispielbefehle ab Projektwurzel; neue Ausgabeziele.

die Standardausgabe. Die Messcontroller prüfen die Integrität ihrer archivierten Quellen. Der U2-Vergleich kann aus dem datierten Berichtsordner wie folgt erneut gestartet werden; mit zusätzlichem `-pilot` und einem anderen Ausgabeordner wird stattdessen die kleine Pilotkonfiguration gewählt:

```
python3 data/uniform_v2/source/benchmark_compare.py \
  --baseline data/uniform_v1 --output /tmp/cfk-u2-replay
```

## 10.4 Kleines Anwendungsbeispiel

Dieses Beispiel wird im Ordner mit den Modulen ausgeführt. Bei Prozessparallelität muss der Programmeinstieg wegen `spawn` geschützt sein.

```
from geometry import area, vertices, neighbors
from particles_fast import build_particle_mesh

if __name__ == "__main__":
    mesh = build_particle_mesh(
        [(0.1, 0.2), (0.7, 0.8), (0.11, 0.21)],
        workers=1,
    )
    mesh.tree.validate()
    for k in sorted(mesh.tree.leaves):
        print(k, area(k), vertices(k), neighbors(k))
```

Listing 5: Baumaufbau und Zugriff auf Geometrie.

Für interaktive oder bildbasierte Kontrolle dient zusätzlich `viewer.py`; seine Kommandozeilenhilfe beschreibt die verfügbaren Ein-/Ausgabepfade. Der Benchmark misst absichtlich nicht die Darstellung.

## 10.5 Empfohlene Zitier- und Nachweispraxis

Bei Übernahme in eine Forschungsarbeit sind die konkrete Hierarchiekonvention, die Randregel, das Zufallsmodell, die Messfenster und die Algorithmusversion mit anzugeben. Eine bloße Angabe „CFK, Python, zwölf Kerne“ reicht nicht zur Reproduktion. Aussagen zur allgemeinen CFK-Theorie sind mit dem Originalmanuskript abzugleichen; die hier bewiesenen Minimalitäts- und Vereinigungsresultate beziehen sich ausdrücklich auf den beschriebenen Abschlussoperator. Aussagen zur Leistung sind auf die gemessenen Größen, Eingaben und Hardware zu beschränken.

## Literatur

[1] Ronaldo Marinho Persiano, João Luiz Dihl Comba, and Valéria Barbalho. An adaptive triangulation refinement scheme and construction. Lokal bereitgestelltes Manuskript, 8 Seiten, Datei adptri93.pdf. Publikationsjahr und Publikationsorgan nicht gesichert. Archivkopie und SHA-256 im Berichtsordner.

[2] Python Software Foundation. Python 3.13 documentation: concurrent.futures. https://docs.python.org/3.13/library/concurrent.futures.html, . Zugriff am 16. September 2026; ProcessPoolExecutor.

[3] Python Software Foundation. Python 3.13 documentation: multiprocessing. https://docs.python.org/3.13/library/multiprocessing.html, . Zugriff am 16. September 2026; insbesondere Startverfahren und Datenaustausch.