CFK: Schnellere Kraftbaumvorbereitung

Algorithmische Verbesserungen und OpenMP im Millionenfall

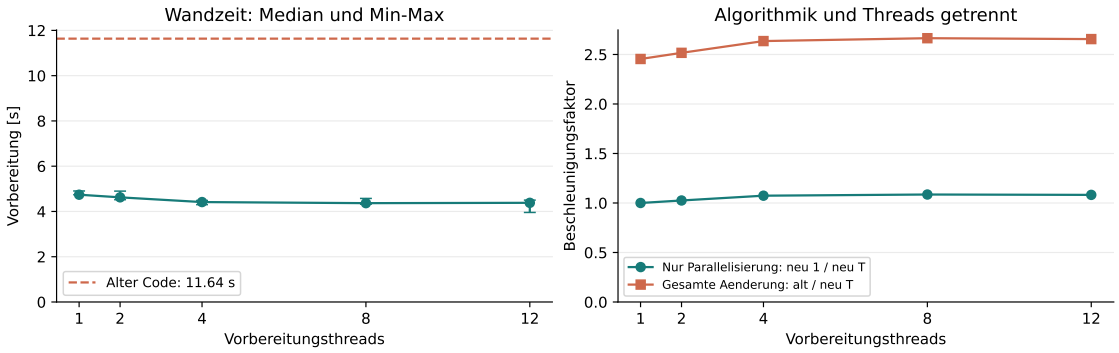
17. September 2026

1 Ergebnis auf einen Blick

Die Vorbereitung sinkt von 11.635 s (alter Code) auf 4.740 s (neuer Code, ein Thread) und 4.382 s (zwoelf Threads). Die gesamte Rechnung sinkt von 27.92 auf 20.32 s, also um 27.2 Prozent. Der reine Threadgewinn der Vorbereitung betraegt Faktor 1.08; die gesamte Aenderung erreicht Faktor 2.66.

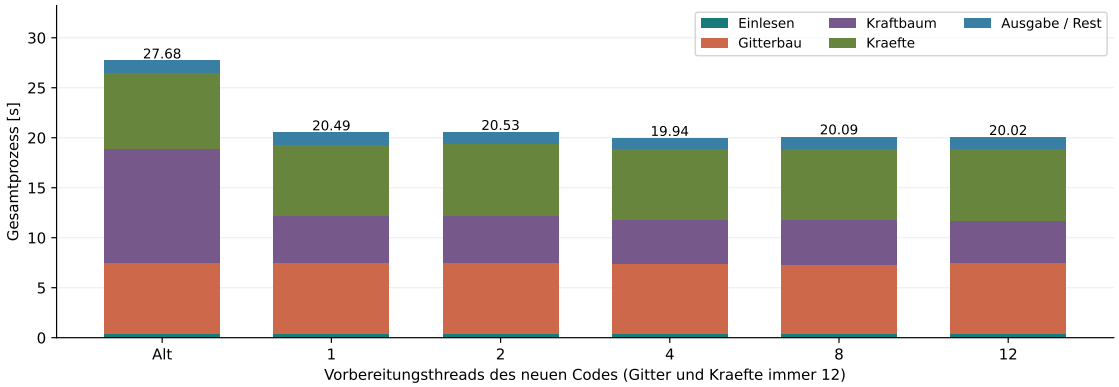
Threads	Vorbereitung [s]	Min–Max [s]	Gesamt [s]	Faktor	MiB
Alt	11.635	10.963–11.895	27.92	1.00	747.7
1	4.740	4.740–4.905	20.54	2.45	665.0
2	4.623	4.520–4.899	20.66	2.52	665.2
4	4.415	4.301–4.486	19.97	2.64	665.4
8	4.367	4.320–4.576	20.02	2.66	665.2
12	4.382	3.956–4.501	20.32	2.66	665.3

Mediane aus je drei Aufrufen. Alt bezeichnet das unveraenderte Programm vor CPP_G6. Gitterzuordnung und Kraftrechnung verwenden immer zwoelf Threads. Gesamt umfasst den ganzen Prozess, einschliesslich JSON-Ein-/Ausgabe; MiB bezeichnet den maximalen residenten Speicher des C++-Prozesses.

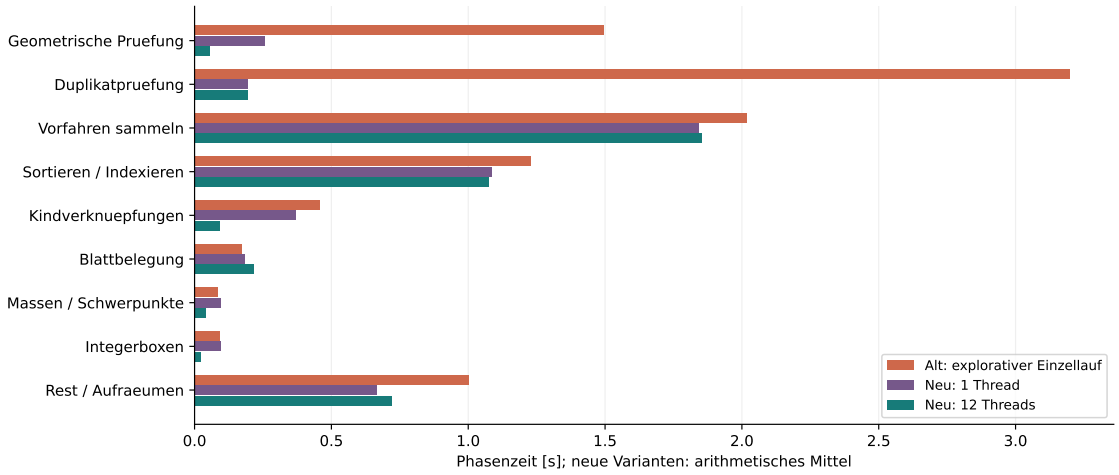


Den kleinsten Vorbereitungsmedian hatte hier die Einstellung mit 8 Threads. Kleine Unterschiede innerhalb der Laufzeitstreuung begruenden keine belastbare Rangfolge. Der Vergleich neu/ein Thread gegen neu/mehrere Threads isoliert die Parallelisierung; alt gegen neu enthaelt zusaetzlich Algorithmik und Datenlayout.

2 Wo die Zeit bleibt



Die gestapelten Balken zeigen arithmetische Mittel, damit die Bestandteile zum mittleren Gesamtwert addieren. Die Tabelle auf Seite 1 zeigt dagegen Mediane. Ausgabe/Rest ist die Differenz zur externen Prozesszeit; sie enthaelt auch Destruktoren ausserhalb der Zeitfenster und die Rundung von GNU time.



Das alte Phasenprofil ist ein einzelner, explorativer instrumentierter Lauf, teilweise waehrend der Fertigstellung eines Testbuilds. Es dient der Lokalisierung teurer Arbeit, nicht als praezise Speedup-Kontrolle. Die alten Gesamtzeiten auf Seite 1 stammen dagegen aus drei neu gemessenen, eingestreuten Kontrolllaeufen. Rest/Aufraeumen innerhalb der Vorbereitung umfasst vor allem lokale Containerdestruktoren.

3 Mathematik und Implementierung

3.1 Exakte Koordinaten statt pauschaler Langinteger-Arithmetik

Die Wurzel hat die Ecken $(0,0),(2,0),(0,2)$ und Code 1. Die Kinder von k tragen $2k$ und $2k+1$; $d(k)=\lfloor \log_2 k \rfloor$ ist die Tiefe. Fuer einen exakt dyadischen Punkt schreiben wir seine baryzentrischen Koordinaten als $(1-(B+C)/D, B/D, C/D)$ mit $0 \le B, C$ und $B+C \le D$. Links wird gewaehlt, wenn $B \ge C$, sonst rechts. Beim Abstieg gilt

$$(B,C) \longmapsto \begin{cases} (D-B-C,\ B-C), & B \ge C, \\ (C-B,\ D-B-C), & B < C. \end{cases}$$

Der Nenner D bleibt konstant. Die Codebits muessen mit jeder dieser Entscheidungen uebereinstimmen. Die Zuordnung auf Teilungskanten bleibt damit eindeutig. Ist $x = n_x 2^{e_x}$ und $y = n_y 2^{e_y}$ normalisiert, wird $q = \max(0, -e_x, -e_y)$, $D = 2^{q+1}$, $B = n_x 2^{e_x+q}$ und $C = n_y 2^{e_y+q}$ gesetzt. Fuer $e_x, e_y \ge -62$ passen B, C, D in uint64. Die Eingangsgrenze $B \le D,\ C \le D - B$ vermeidet auch bei ungueltigen Daten einen Additionsueberlauf. Kleinere Exponenten, insbesondere subnormale Werte, verwenden weiterhin Boost cpp_int. Kein Punkt wird gerundet und keine Validierung ausgelassen.

3.2 Duplikate, Knoten und Schwerpunkte

Nach der parallelen Werte-/Geometriepruefung werden Positionen und IDs in zusammenhaengenden Arrays sortiert. Gleiche benachbarte Eintraege sind Duplikate. Das ersetzt die vorherigen baum-/hashbasierten Einzeleinfuegungen; Komplexitaet der Sortierung ist $O(n \log n)$. NaNs sind vorher ausgeschlossen, sodass die Sortierordnung wohldefiniert ist. Vorfahren werden weiterhin seriell gesammelt. Nach dem Aufbau der Indexmap ist diese unveraenderlich; Kindverweise koennen parallel nachgeschlagen werden. Die numerische Codesortierung ordnet auch nach Tiefe. Pro Tiefe sind die Eltern voneinander unabhaengig; eine Barriere trennt aufeinanderfolgende Ebenen.

$$M_k = M_{2k} + M_{2k+1}, \qquad \boldsymbol{c}_k = \frac{M_{2k}}{M_k}\boldsymbol{c}_{2k} + \frac{M_{2k+1}}{M_k}\boldsymbol{c}_{2k+1}.$$

Fehlende masselose Kinder werden ausgelassen. Pro Knoten bleibt die Operationsfolge links, rechts unveraendert. Deshalb wird keine neue Gleitkomma-Summationsreihenfolge eingefuehrt. Kleine Arbeitsbereiche unter 1024 Eintraegen bleiben seriell.

3.3 Boxgeometrie und Fehlerbehandlung

Integer-Ecken werden bis Tiefe 8 seriell weitergegeben, die dort vorhandenen Teilbaeume anschliessend parallel durchlaufen. Jeder Knoten wird einmal behandelt; es gibt keine neue Rekonstruktion ab der Wurzel fuer jeden Code. Bei konzentrierten Wolken kann diese Aufteilung unausgewogen sein. Ausnahmen werden innerhalb jedes Threads abgefangen und erst nach Teamabschluss erneut geworfen. Keine gemeinsame Hash-Tabelle wird parallel veraendert.

3.4 Empfehlung

Bei zwoelf Threads entfallen im Median 3.00 Sekunden auf Vorfahrensammlung und Indexierung: hier lohnt weitere Arbeit. Vier bis zwoelf Threads liegen nahe beieinander; eine allgemein optimale Threadzahl folgt daraus nicht. Die Skalierung bei konzentrierten Wolken und zehn Millionen Teilchen wurde nicht gemessen.

4 Versuchsprotokoll und Grenzen

Eine Million unveraenderte uniforme Teilchen aus CPP_G3, mit denselben Massen und Positionen. Je drei Wiederholungen messen Laufzeitschwankungen derselben Eingabe, nicht unterschiedliche physikalische Stichproben. Alle Millionenbeschleunigungen aller 18 Laeufe sind bitgleich zur archivierten Referenz; auch die ganzzahligen Durchlaufzaehler stimmen ueberein. Weitere sechs Vergleiche verwenden uniforme, Gauss- und Mischwolken mit je 100000 Quellen, beiden MACs und den archivierten 128 Zielen.

- Referenzcompiler GCC 13.3, Boost 1.83, C++20; `-O3 -DNDEBUG -ffp-contract=off`; kein Fast-Math und keine neue Native-/LTO-Option.
- Fuer den Millionenfall: classic, $\theta = 0.5$, $G = 1$, $\epsilon = 0$, alle Ziele.
- Reihenfolgen (0 = alter Code): 0,1,2,4,8,12; 12,8,4,2,1,0; 4,0,1,12,2,8.
- OpenMP: `OMP_DYNAMIC=FALSE`. Angeforderte und reale Teamgroessen sind in JSON gespeichert. Keine feste Kernbindung oder Taktkontrolle; hybride CPU.
- Keine gleichzeitig laufenden Builds oder anderen von uns gestarteten Rechenexperimente waehrend der Hauptmessserie. Hintergrundlast ist nicht kontrolliert.
- Release, ASan/UBSan mit und ohne OpenMP sowie begrenzte Teamgroesse geprueft. LeakSanitizer ist wegen der Umgebung deaktiviert. Kein Race-Detector und keine kuenstlich injizierten Allokationsfehler.

Prozessor: Intel(R) Core(TM) Ultra 7 255U. Die genaue CPU- und Speicherausstattung ist im Rohdatenarchiv protokolliert. Die Tests vergleichen alle Knotenfelder und Integerboxen bei mehreren Threadzahlen, beide Kraftkriterien, Tiefe 63, Grenzkoordinaten und parallele Fehlerpfade. Ein unabhaengiger Langinteger-Referenzpfad prueft die exakte Zuordnung bis Tiefe 63. Bitgleichheit ist fuer diesen Build und diese Plattform nachgewiesen, keine Zusage fuer beliebige Compiler.

4.1 Benutzung und Wiederholung

```
cfk_force --input input.json --output result.json --all
  --threads 12 --mesh-threads 12 --prepare-threads 12
```

Ohne separate Option uebernimmt die Vorbereitung den Wert von `--threads`. Die C++-API `GravityTree(std::move(p), threads)` bleibt ohne zweites Argument seriell. Ohne OpenMP erfolgt die Vorbereitung ebenfalls seriell; die Kraft-API verweigert dort weiterhin mehr als einen Thread.

```
python3 python/benchmark_prepare.py
  --output experiments/cpp_g6_neu
  --old-binary experiments/cpp_g6_force_prepare/before_force
python3 python/summarize_prepare.py experiments/cpp_g6_neu
  --report ../Berichte/2026-09-17/CPP_G6_neu
```

Die Zeilen der Beispiele sind jeweils als ein Befehl auszufuehren. Rohdaten, Programme, Quellstand, Buildoptionen und SHA-256-Manifest: `CPP/experiments/cpp_g6_force_prepare`. Sicherung davor: Sicherungen/2026-09-17_vor_CPP_Kraftbaumparallel. Dieser Bericht liefert CSV-Einzelmesswerte sowie PNG-/PDF-Diagramme. Sein separates Manifest erfasst auch die Auswertungsskripte.