CFK: Partiell paralleler Gitterbau

Eine Million Teilchen

17. September 2026

1 Ausgangslage und Aenderung

CPP_G5, 17. September 2026. Bislang waren in C++ nur die Kraftdurchgaenge parallel. Gitterbau, Blattzuordnung und Kraftbaumvorbereitung liefen seriell. Die wechselnde CPU-Auslastung war daher erwartbar.

Jetzt ist die abschliessende Blattzuordnung parallel. Erst nach dem Konformitaetsabschluss steht das fertige Gitter fest. Jeder Teilchenpfad kann darin unabhaengig von allen anderen Pfaden verfolgt werden. Die mathematischen Teilungsregeln und der Konformitaetsabschluss wurden nicht geaendert.

Eingabepruefung und Packen, zwingende Teilungen, Konformitaetsabschluss, Materialisierung der Knoten sowie Kraftbaum und Massenschwerpunkte bleiben seriell. Es handelt sich bewusst um partielle Parallelisierung, nicht um einen vollstaendig parallelen Baumaufbau.

2 Sicherheit und Reproduzierbarkeit

Die Split-Menge wird vor dem Parallelbereich vollstaendig aufgebaut und darin ausschliesslich gelesen. Jedes Teilchen besitzt eigene gepackte Koordinaten und einen eindeutigen Ausgabeplatz. Es gibt keine gemeinsamen Hash-Inserts, keine Gleitkommareduktion und keine voneinander abhaengigen Teilchenaufgaben. OpenMP verteilt zusammenhaengende Bereiche statisch.

Der normale Schleifenkoerper enthaelt keine Sperre; die Pfadlaenge ist durch die maximale Baumtiefe begrenzt. Exceptions werden pro Thread abgefangen und nach der gemeinsamen OpenMP-Abschlussbarriere weitergegeben. Damit werden zyklische Warteabhaengigkeiten im Algorithmus vermieden. Dies ist keine formale Garantie fuer Betriebssystem oder OpenMP-Laufzeit.

Die nativen Tests vergleichen mehrere Threadzahlen auf normalen und besonders feinen Koordinaten. Release sowie ASan/UBSan pruefen den seriellen und parallelen Pfad. LeakSanitizer bleibt wegen der Prozessueberwachung deaktiviert. Kein spezieller Race-Detector und keine kuenstliche Allokationsfehler-Injektion wurden ausgefuehrt.

3 Versuchsplan

Die identischen 1000000 uniformen Teilchen aus CPP_G3 werden verwendet. Compiler: bisheriger Referenzbuild mit -O3, ohne Fast-Math, ohne FMA-Kontraktion. 1, 2, 4, 8 und 12 angeforderte Aufbau-Threads, je drei separate Aufrufe. Die Reihenfolge wechselt; waehrend der Messungen wird nicht kompiliert.

Dazu kommen je drei Gesamtlaeufe mit einem beziehungsweise zwoelf Aufbau-Threads. Die Kraftrechnung verwendet in beiden Faellen zwoelf Threads und berechnet alle Teilchen: classic, theta=0.5, G=1, epsilon=0. OMP_DYNAMIC=FALSE; Takt und Hintergrundlast sind nicht kontrolliert.

Die Python-Versuchssteuerung liest und vergleicht zwischen den Aufrufen grosse Ergebnisdateien seriell. Auch diese Arbeit ist im Systemmonitor sichtbar, gehoert aber nicht zu den gemessenen C++-Prozesszeiten.

Alle 15 Gitteraufbauten mit je einer Million Teilchen stimmen in Knotencodes, zwingenden Teilungen, Blattzuordnungen und Tiefenhistogrammen exakt mit dem alten seriellen Archiv ueberein. Die sechs Gesamtlaeufe ergeben bitgleiche Kraefte und identische Baumdurchlauf-Zaehler. Drei weitere 100000er-Wolken (uniform, gaussfoermig, Mischung) bestehen ebenfalls den exakten Vergleich.

4 Skalierung des Gitterbaus

Zeitmediane in Sekunden. Gitterbau umfasst alle Aufbauphasen einschliesslich Blattzuordnung, ohne Datei-I/O. Die Zuordnungszeit enthaelt Teamstart und Synchronisation. Angefordert und tatsaechlich verwendet werden getrennt ausgewiesen.

Threads	Team	Gitter [s]	Bereich [s]	Zuordnung [s]	Faktor
1	1	17.574	16.166-18.101	11.041	1.00
2	2	10.589	10.523-11.156	4.181	1.66
4	4	8.630	8.572-8.743	1.973	2.04
8	8	7.306	7.294-8.096	0.902	2.41
12	12	7.029	7.018-7.438	0.733	2.50

5 Vollstaendige Rechnung

Gesamtzeit und Peak-RSS stammen von GNU time und enthalten Ein-/Ausgabe des C++-Prozesses, nicht den separaten Auswertungsprozess. Einzelphasenmediane sind nicht additiv zum Gesamtmedian.

Aufbau-Threads	Gitter [s]	Kraftbaum [s]	Kraefte [s]	Gesamt [s]	MiB
1	17.920	11.635	7.479	38.740	747.6
12	7.087	10.840	7.469	27.010	747.7

6 Einordnung

Der Gittermedian sinkt mit zwoelf Threads von 17.574 auf 7.029 Sekunden. Die reine Blattzuordnung sinkt von 11.041 auf 0.733 Sekunden.

Die vollstaendige Rechnung sinkt im Median von 38.740 auf 27.010 Sekunden, entsprechend Faktor 1.43 beziehungsweise 30.3 Prozent weniger Zeit.

Den kleinsten Aufbau-Median hatte hier die Einstellung mit 12 Threads. Die Unterschiede muessen mit den beobachteten Bereichen verglichen werden; drei Wiederholungen sind eine technische Diagnose, keine allgemeine Skalierungsgarantie. Gemeinsame Caches und Speicherbandbreite begrenzen den Nutzen.

Die Threads werden nicht an feste Kerne gebunden. Auf der hybriden CPU koennen Kernplatzierung, Takt und Cacheverhalten die Quotienten beeinflussen. Ein eventuell ueber der Threadzahl liegender Quotient der Zuordnungszeiten ist deshalb kein Nachweis allgemeiner ueberlinearer Skalierung.

Deadlock-Freiheit allein erzeugt keinen Speedup. Fuer einen parallelisierbaren Zeitanteil p ist das ideale Amdahl-Modell $S(T) = \frac{1}{(1-p)+p/T}$; Verwaltungs- und Speicherkosten kommen real hinzu. Die weiterhin seriellen Phasen erklaeren, weshalb der Gesamtgewinn viel kleiner als die Threadzahl ist.

Als naechster Schritt bietet sich die nun relativ wichtigere Kraftbaumvorbereitung an. Deren Validierung, Hash-Aufbau und Aggregation sollten vor einer weiteren Parallelisierung separat vermessen werden.

7 Benutzung und Archiv

In C++: build_mesh(x,y,max_depth,threads); Standard bleibt ein Thread. In der CLI gilt --threads jetzt fuer Kraftrechnung und Blattzuordnung. Mit --mesh-threads kann nur die Zuordnung separat eingestellt werden. --mesh-threads 1 --threads 12 stellt den bisherigen seriellen Aufbau wieder her.

Unter 1024 Teilchen oder ohne OpenMP wird weiterhin seriell zugeordnet. ownership_threads protokolliert die tatsaechliche Teamgroesse. Es werden keine zusaetzlichen Threadpools oder Prozesse erzeugt.

123 Dateien wurden anhand des Manifests geprueft. Rohdaten, Quellstand, Programm und Buildoptionen liegen in CPP/experiments/cpp_g5_parallel_mesh. Die Sicherung davor liegt unter Sicherungen/2026-09-17_vor_CPP_Parallelbau.

Wiederholung aus CFK/CPP: python3 python/benchmark_parallel_mesh.py --output experiments/cpp_g5_neu. Auswertung: python3 python/summarize_parallel_mesh.py experiments/cpp_g5_neu.