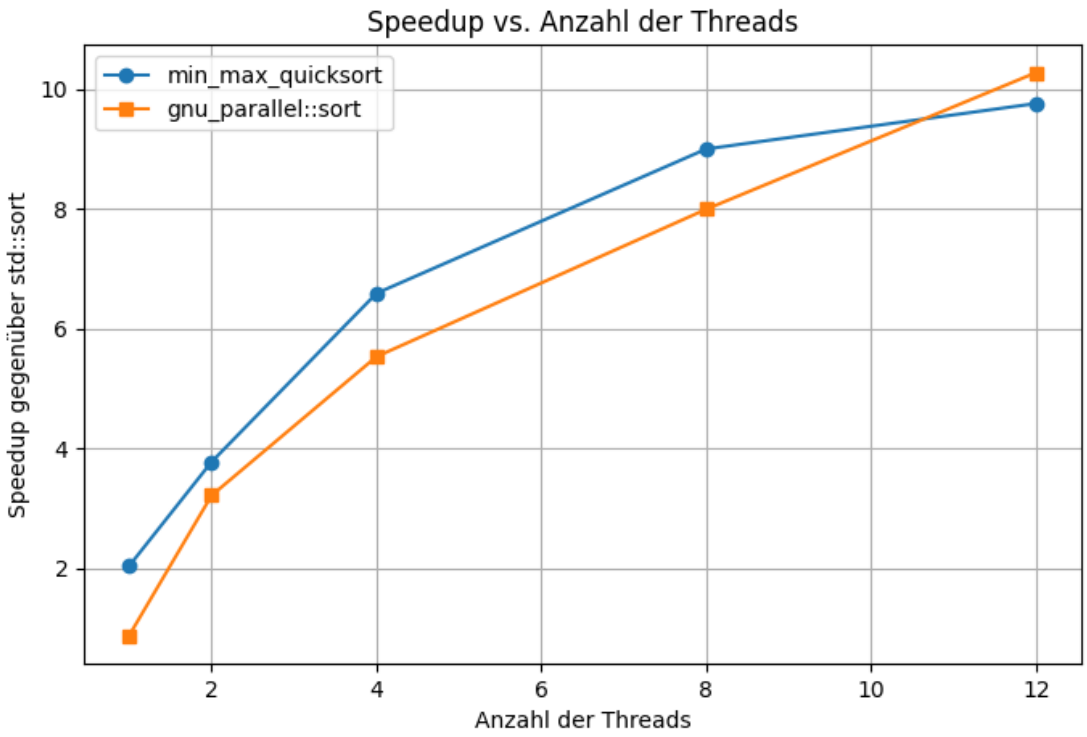
Vorlesung 4

Aufgabe 1: Benchmark

Benchmarking environment:

Betriebssystemname:	Microsoft Windows 11 Pro
Gesamter physischer Speicher:	16.313 MB
Systemtyp:	x64-based PC
Prozessor(en):	1 Prozessor(en) installiert.
	[01]: AMD64 Family 25 Model 33 Stepping 0 AuthenticAMD ~3701 MHz AMD Ryzen 5 5600X 6-Core Processor
NumberOfCores	6
NumberOfLogicalProcessors	12
Compiler	g++.exe (GCC) 13.2.0

Speedup VS Anzahl von Threads



*threads = [1, 2, 4, 8, 12]
speedup_min_max = [2.04111, 3.77806, 6.58864, 8.99693, 9.75585]
speedup_gnu_parallel = [0.881167, 3.21758, 5.53244, 7.99182, 10.2711]*

Beobachtungen:
1. Mit mehr Threads steigt der Speedup bei beiden Methoden

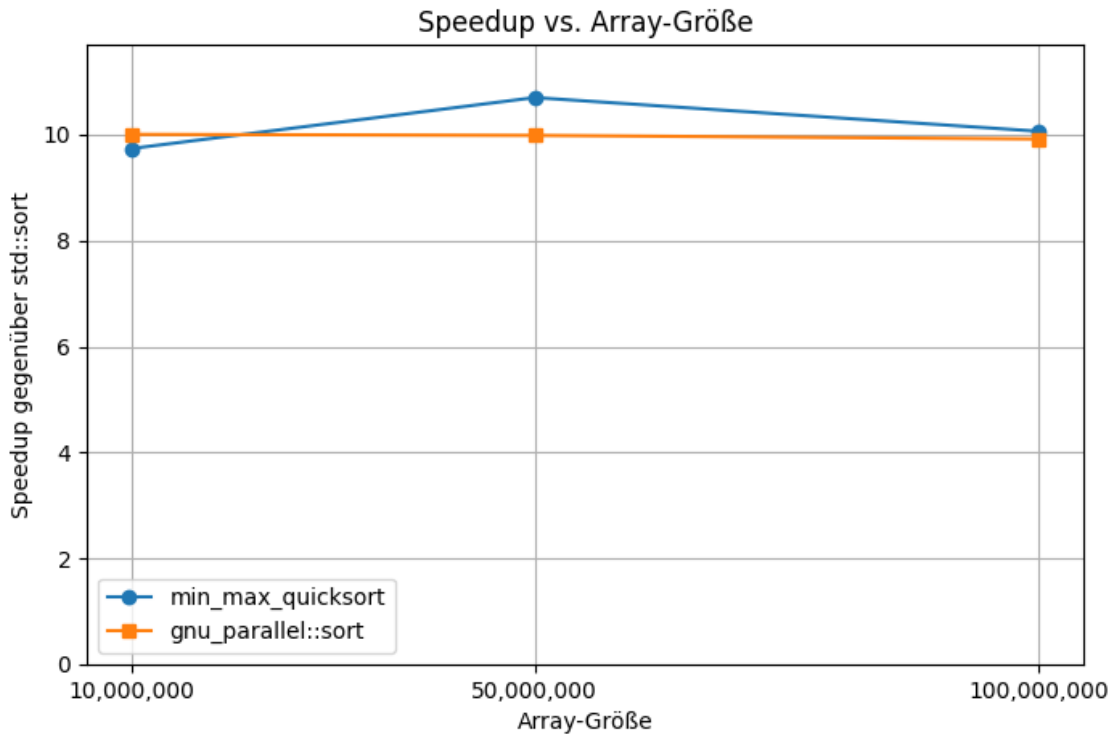
2. Unterschiedliches Speedup bei einem Thread (keine Parallelisierung) – Min-Max 2.04 und GNU Parallel nur 0.88

Vielleicht erzeugt GNU Parallel trotzdem Parallelität und verteilt die Daten auf mehrere Kerne, obwohl letztendlich nur 1 Kern arbeitet (wodurch Overhead entsteht)

3. Speedup bei GNU Parallel ist bei 12 Threads höher als Min-Max

Vielleicht liegt es daran, dass wenn ein Thread in Min-Max früher fertig ist, macht er nichts weiter, aber die Threads bei GNU Parallel nehmen sich Arbeit von anderen Threads – wodurch sie gleichmäßiger ausgelastet werden

Speedup VS Größe von Array



*size = [10.000.000, 50.000.000, 100.000.000]
speedup_min_max = [9.74, 10.7008, 10.0657]
speedup_gnu_parallel = [10.0068, 9.98742, 9.91853]

Beobachtungen:

1. Min-Max Speedup – steigt bei 50M und fällt leicht danach

Alle Arrays sind zu groß für L3, also müssen sie über RAM geholt werden. Ich weiß nicht, wieso der höchste Speedup bei 50.000.000-Array-Größe auftritt. Vielleicht weil es bei 10.000.000 zu wenig Arbeit für alle Threads gibt und bei 100.000.000 die Speicherzugriffe ineffizienter werden.

2. GNU Parallel Speedup ist im Vergleich zu Min-Max stabiler

Möglicher Grund dafür ist die Lstverteilung, bei Min-Max – statisch und bei GNU Parallel dynamisch => GNU Parallel arbeitet effizienter
Noch dazu verwendet Min-Max rekursive Threads, am Funktionsende müssen die Threads durch taskwait synchronisiert werden – da die Lastenverteilung statisch ist und da wenn ein Thread früher fertig wird, er nicht die Arbeit von anderen übernimmt – ungenutzte Ressourcen und Wartezeiten. Und GNU Parallel ist nicht an feste Synchronisationen an bestimmten Stellen angewiesen.

Aufgabe 2

„Read What every systems programmer should know about concurrency. Discuss two things you find particularly interesting.“

1. fetch_add und compare_and_swap

Ich fand interessant, dass diese 2 Konzepte unterschiedliche Strategien haben, wie man Probleme in der Parallelprogrammierung lösen kann. Und dass man nicht pauschal sagen kann, welche von beiden besser ist.

Bei fetch_add: das Wettbewerb von Threads wird quasi „akzeptiert“. Fetch_add verhindert nicht, dass mehrere Threads gleichzeitig auf eine Variable zugreifen, aber sorgt dafür dass eine Operation atomar ausgeführt wird. Die Operation wird IMMER ausgeführt, egal ob ein anderer Thread schon was verändert hat oder nicht. Fetch_add ist schnell, weil es nur eine einzelne Maschineninstruktion braucht => gut für unabhängige Updates (wie Counter), wo es egal ist, in welcher Reihenfolge sie passieren.

Bei CAS: das Wettbewerb von Threads wird eher kontrolliert, als nur akzeptiert. CAS prüft, ob sich der Wert seit dem letzten Lesen verändert hat. Wenn schon – muss der Thread später noch mal versuchen – dadurch ist CAS langsamer. Aber perfekt, wenn es sichergestellt werden muss, dass eine Variable nur dann geändert wird, wenn sie nicht unerwartet verändert wurde => gut für lockfreie Strukturen, weil es den Threads erlaubt Änderungen zu koordinieren ohne sich gegenseitig zu blockieren.

2. Memory Barriers in Weakly-Ordered Architekturen

Die Operationen werden nicht immer in der Reihenfolge ausgeführt, wie sie im Code stehen. „Feind“ Nr.1 – Compiler, „Feind“ Nr.2 – Weakly-ordered Architekturen. Ich fand interessant, welche Techniken es gibt, um trotzdem sicherzustellen, dass der Code in der korrekten Reihenfolge ausgeführt wird (trotz Compiler-Optimierungen und weakly-ordered Architekturen) – zum Beispiel, Memory Barriere.

Sie garantieren, dass die Operationen (wenn nicht gewollt) nicht vertauscht werden. Normalerweise entscheidet die CPU selbst, aber Memory Barriere sind wie eine Art mit dem Prozessor zu kommunizieren, um mitentscheiden zu dürfen, wo welche Reihenfolge wichtig ist

- std::memory_order_seq_cst – die strengste Reihenfolge – globale, sequentielle Reihenfolge aller Operationen
- std::memory_order_acquire – verhindert, dass CPU “vorausrechnet” – CPU muss warten bis geladene Variable korrekt gelesen wird
- std::memory_order_release – garantiert, dass ALLE vorhergehenden Operationen abgeschlossen sind, bevor die aktuelle Schreiboperation ausgeführt wird
- std::memory_order_relaxed – atomare Ausführung, aber ohne bestimmter Reihenfolge