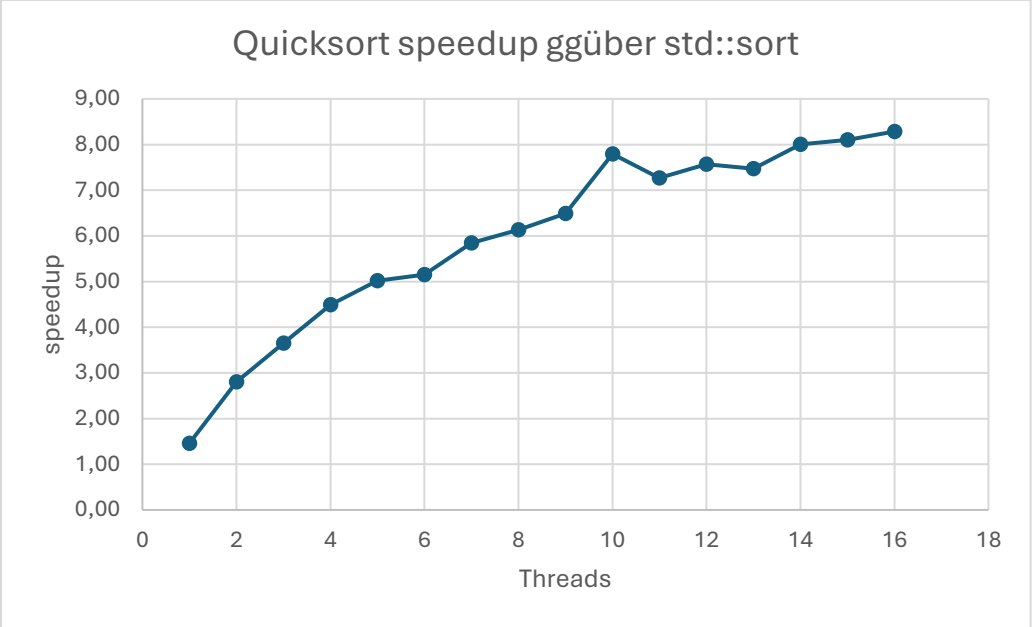
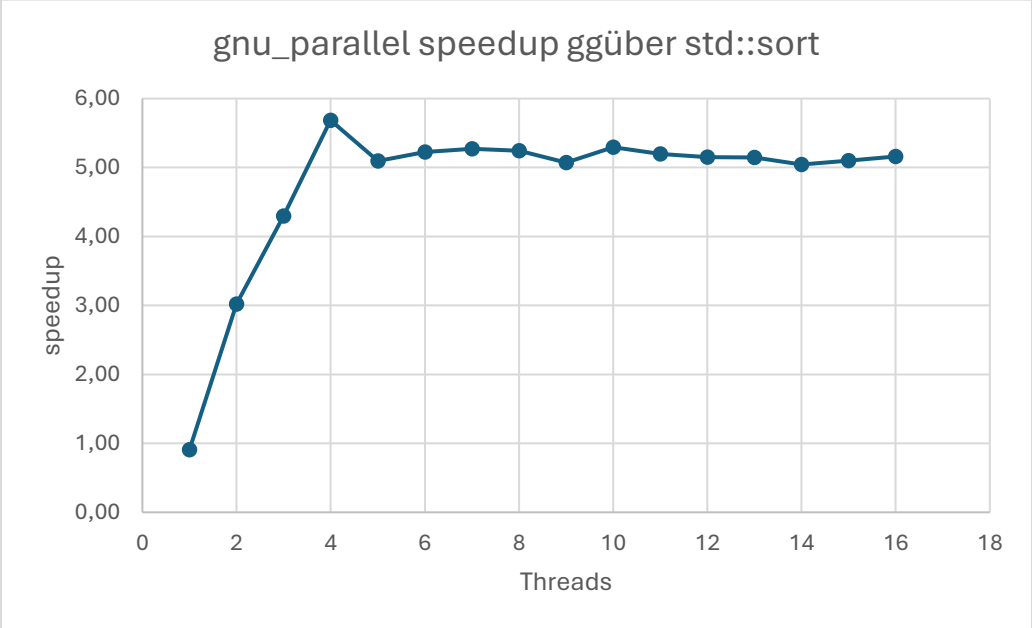
Vorlesung 4 – Exam Assignments

1) Benchmark min_max_quicksort against std::sort and __gnu_parallel::sort

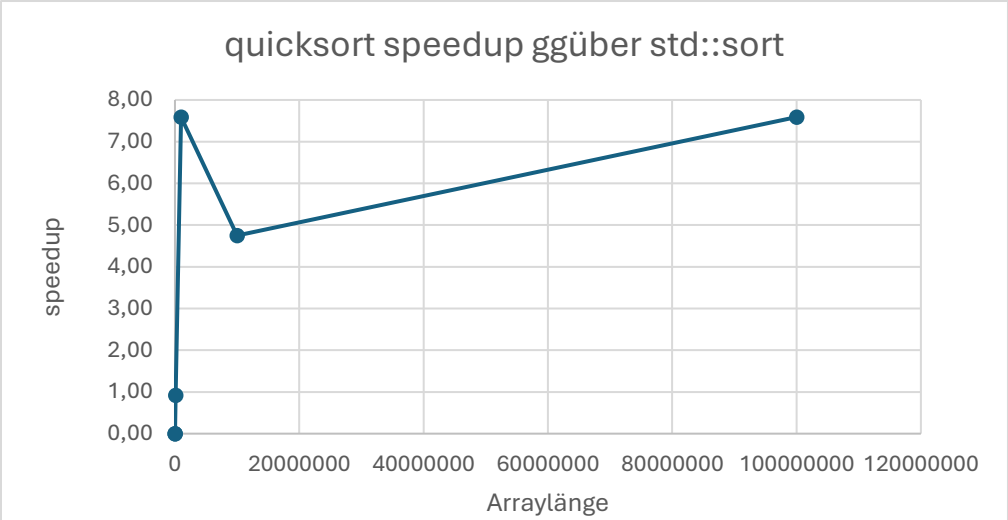
Messwerte in der Excel-Datei!

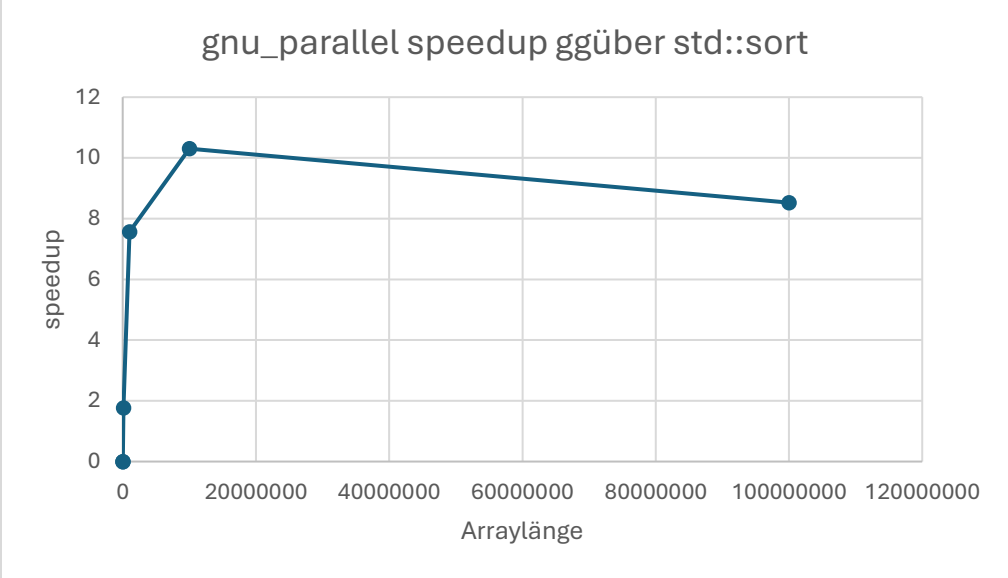


- Der Speedup nimmt mit ansteigender Thread-Zahl zu
- Es sieht nach einem linearen Anstieg aus



- Der Speedup nimmt mit Steigender Thread-Zahl schnell zu, bis er ab 4 Threads ein Plateau erreicht und nicht weiter steigt
- Hier scheinen Parallelisierungsgrenzen des Sortierverfahrens Hardwaregrenzen wie Speicherzugriffslatenzen, Cache-Kohärenz und Bandbreite erreicht





- Die Messwerte waren leider bei beiden Messungen unvollständig
- Es lässt sich ein starker Anstieg zu Beginn mit anschließendem Erreichen eines Plateaus vermuten
- Für kleine Arrays funktionieren beide Sortierverfahren eventuell schlechter als std::sort aufgrund des Overhead durch die Thread-Erstellung
- Hardwarelimitierungen dürften hier besonders eine Rolle spielen

2) „What every systems programmer should know about Concurrency“

Atomare Variablen und Operationen

- Auf Mulit-Core-Prozessoren kann es passieren, dass bei Zugriffen von mehreren Threads auf die gleiche Variable ungewollte Ergebnisse entstehen, da unterschiedliche Rechengeschwindigkeiten, Compiler und Hardwarearchitektur die Befehlsreihenfolge durcheinanderbringen können – sogenannte race conditions
- Eine Möglichkeit, um dies zu verhindern und trotzdem Parallelität zu nutzen, sind atomare Variablen
- Atomare Variablen sind wie Treffpunkte für die Threads und stellen sicher, dass Befehle, die vor einem atomaren Variablenzugriff ausgeführt werden sollen auch wirklich davor ausgeführt werden und die danach ausgeführt werden sollen auch wirklich danach ausgeführt werden

- Auch read-modify-write-Operationen können als ein atomarer Schritt definiert werden, um race conditions zu vermeiden

Cache Effekte und false sharing

- Speicher zwischen RAM und CPU wird in Cache-Lines übertragen
- Wenn ein Prozessor-Kern auf einer Cache-Line schreibt und ein anderer Kern diese lesen will, muss die gesamte Cache-Line vom Cache des ersten Kerns in den Cache des zweiten übertragen werden, damit der Speicher kohärent bleibt
- Wenn dies häufig passiert und die kritischen Abschnitte nicht sehr groß sind, kann dieses Hin-und-her-springen zwischen den Caches länger dauern als die kritischen Abschnitte selbst – es kommt so zu starken Performanceeinbußen
- Besonders schlimm ist diese Verlangsamung bei unabhängigen Variablen, die zufällig auf der gleichen Cache-Line gespeichert wurden – sogenanntes false sharing
- Eine Möglichkeit, dies zu vermeiden, besteht darin, atomare Variablen mit einer Cache-Line nicht gemeinsam genutzter Daten aufzufüllen, aber dies ist offensichtlich ein großer Kompromiss zwischen Speicherplatz und Zeit