87 lines (66 sloc) 5.43 KB

Praktikum I

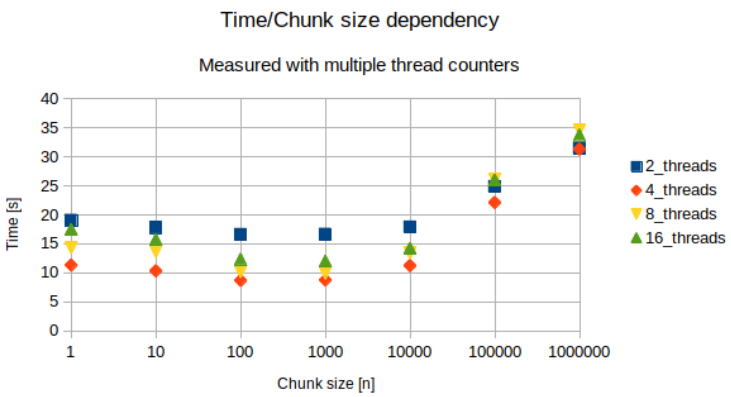
Aufgabe 1

Quellcodeverzeichnis

Teilaufgabe	Verzeichnis	Quellcode	Compile command	Execute command
A	`Exercise1/TaskA/`	`task-a.c`	`make`	`./taska <amount> <chunksize>`
B	`Exercise1/TaskB/`	`task-b.c`	`make`	`./taskb <amount> <chunksize>`
C	`Exercise1/TaskB/`	`task-b.c`	`make`	`./run-tests.sh`
D	`Exercise1/TaskD/`	`task-d.c`	`make`	`./taskd <amount> <chunksize>`

Weiterführende Erklärungen

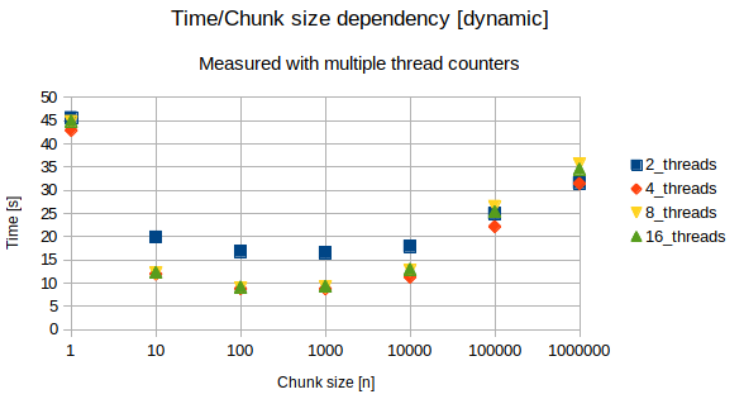
Teilaufgabe C



Zunächst einmal ist festzustellen, dass die gemessenen Laufzeiten unabhängig der Chunksizes bei threads=4 konsistent am kürzesten waren, was mit der Anzahl der genutzten CPU-Cores zusammenhängt. Bei 4 Threads/Cores wird im Optimalfall jedem Core ein Thread zugewiesen, sodass unnötiger Overhead bei etwaigem Threadwechsel entfällt. Die mit nur 2 Threads durchgeführten Berechnungen waren stets am langsamsten, da die hier nicht alle Cores parallel genutzt werden können. Die mit 8 bzw. 16 Threads erzielten Ergebnisse ordnen sich zwischen diesen beiden Extremen ein, was mit zuvor genanntem Overhead erklärbar ist.

Im Bezug auf Chunksizes überwiegt bei sehr kleinen Größen (< 100) der Overhead, während bei sehr großen Chunks (> 10.000) nicht optimal parallelisiert werden kann. Damit liegt die optimale Ausführungszeit bei Chunksizes um 100-1.000.

Teilaufgabe D



Im Großen und Ganzen treffen die Erkenntnisse aus Aufgabe C auch auf Teilaufgabe D zu, mit dem Unterschied des drastisch angestiegenen Overheads. Gerade bei sehr kleinen Chunksizes (z.B. 1) wird das besonders deutlich, da die Ausführungszeit hier sowohl im Vergleich zu größeren Chunks als auch generell statischen Chunks negativ aus dem Rahmen fällt. Darüber hinaus wird die Zeitspanne zwischen 4, 8 und 16 Threads minimiert, was wieder auf den größeren Anteil des (eher statischen) Overheads an der Ausführungszeit zurückzuführen ist.

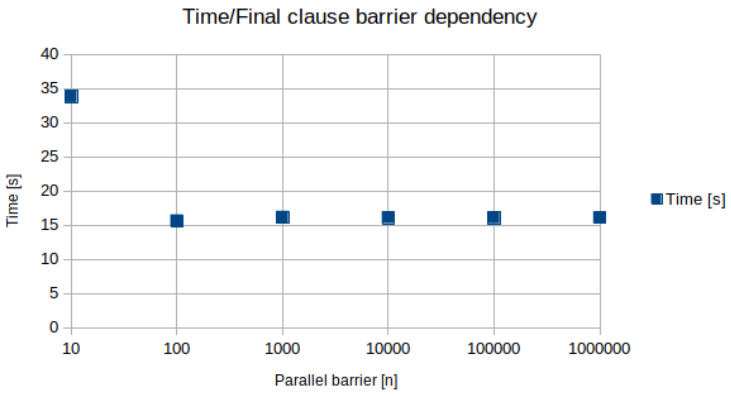
Aufgabe 2

Quellcodeverzeichnis

Teilaufgabe	Verzeichnis	Quellcode	Compile command	Execute command
A	`Exercise2/TaskA/`	`task_a.c`	`make`	`./task_a <array_length>`
B	`Exercise2/TaskB/`	`task_b.c`	`make`	`./task_b <array_length>`
C	`Exercise2/TaskC/`	`task_c.c`	`make`	`./run-tests.sh`

Weiterführende Erklärungen

Teilaufgabe C



Das Laufzeitmessungen verlaufen für alle Parallel barriers - abgesehen von n=10 - nahezu identisch. Der Ausreißer bei n=10 ist unserer Auffassung nach mit dem extrem hohen Overhead zu erklären, der anfällt, wenn Tasks selbst für triviale Berechnungen erzeugt werden. Bei n >= 100 ist sinkt der Overhead aber so weit, dass es hier de facto keinen Unterschied mehr macht, mit welcher konkreten Schrankengröße die Berechnungen durchgeführt werden. Bei geringeren Werten fallen zwar mehr Tasks an, deren Ausführung sich aufgrund der begrenzten Threads jedoch nicht weiter beschleunigt. Zusammenfassend spielt die Schranke also - sofern nicht sehr klein - eine eher untergeordnete Rolle für die Laufzeit.

Aufgabe 3

Quellcodeverzeichnis

Teilaufgabe	Verzeichnis	Quellcode	Compile command	Execute command
B	`Exercise3/`	`task3_B.c`	`make task3_B`	`./task3_B`
C	`Exercise3/`	`task3_C.c`	`make task3_C`	`./task3_C`

Weiterführende Erklärungen

Teilaufgabe A

Teil	Erklärung
sum	In Zeile 11 gibt es bei `result` eine Flow-Dependence
shift	Zeile 20 bei `a[i]` (bzw. `a[i+offeset]`) eine Anti-Dependence
hash	Bei `hash` in Zeile 36,38,42 und 44 eine Flow-Dependence
init	keine iterationsübergreifenden Datenabhängigkeiten

Aufgabe 4

Teilaufgabe A

- count : shared
- val : shared
- g : shared
- *g : private

Teilaufgabe B

- count : shared
- val : private
- cnt : private
- res : private
- *res : private
- i : shared
- j : private
- a : shared

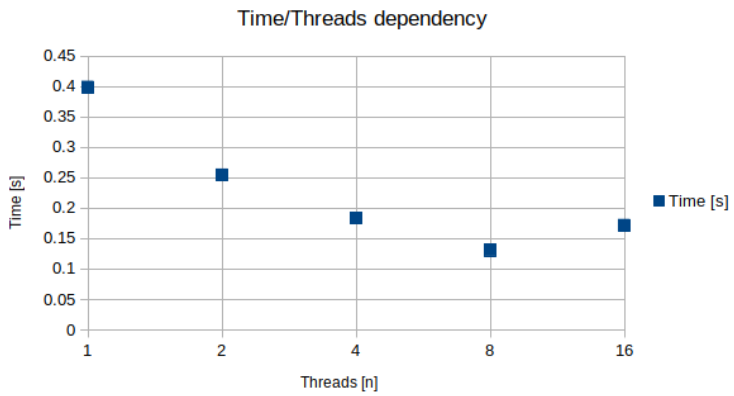
Aufgabe 5

Quellcodeverzeichnis

Teilaufgabe	Verzeichnis	Quellcode	Compile command	Execute command
A + B	`Exercise5/TaskB/`	`task-b.cpp`	`make`	`./taskb <xmin> <xmax> <ymin> <ymax> <maxIter>`
C	`Exercise5/TaskC/`	`task-c.cpp`	`make`	`./run-tests.sh`

Weiterführende Erklärungen

Teilaufgabe C


Time/Threads dependency
0.45
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
Time [s]
1
2
4
8
16
Threads [n]
Time [s]