Universität Heidelberg
Institute for Computer Engineering
(ZITI)

Master of Science Computer Engineering

Parallel Computer Architecture

Exercise 4

Group 04
Barley, Daniel
Barth, Alexander
Nisblé, Patrick

Due date 2019-11-29, 14:00

4.1 Numerische Integration revisited

4.1.1 Parallele Implementierung mittels PThreads

Aufgrund der Assoziativität der Summe kann diese in beliebiger Reihenfolge ausgeführt werden. Wir teilen deshalb jedem Thread eine Partialsumme zu. Dazu wird die Gesamtzahl and Iterationen durch die Anzahl Threads geteilt (Ceil Division, überschüßige Iterationen werden später durch Minimum ausgefiltert). Geht die Division nicht auf arbeitet der Letzte Thread weniger Elemente ab. Somit wird sichergestellt, dass alle Threads etwa gleich lange zur Fertigstellung brauchen. Da im Kern Schleife keine Speicherzugriffe stattfinden ist die Aufteilung der Summen frei wählbar. Die implementierte Aufteilung wurde der Simplizität wegen verwendet. Die einzelnen Partialsummen werden am Ende zusammengeführt. Da die Threads auf keine gemeinsamen Speicherstrukturen zugreifen gibt es während der Ausführungszeit keine Notwendigkeit zur Synchronisation, erst wenn die Partialsummen aufaddiert werden. Dies geschiet hinter der Thread join Barrier wird also erst ausgeführt wenn der letzte Thread beended wurde. Alternativ zur Reduktion aus einem Ergebnissarray hätte auch ein `atomic_int` verwendet werden können, da aber nur sehr wenige Zwischenergebnisse anfallen führt das interne locking welches damit verbunden ist zu schlechterer Performanz.

4.1.2 Experimente und Evaluation

a.

Die Auflösung des `time` commands ist zu klein um die Wallclock messen.

Tabelle 1: Ausführungszeit $t_{compute}$ (ms)

n \ Threads	1	2	4	8	16	32
2	0.23	0.257	0.321	0.495	1.496	1.866
3	0.239	0.28	0.322	0.434	1.413	1.94
4	0.294	0.318	0.338	0.531	1.507	1.953
5	0.919	0.641	0.475	0.522	1.512	1.915
6	7.179	3.731	2.039	1.859	1.322	2.149

b.

Folgende Werte wurden für π berechnet (siehe Tabelle 2). Wir erhalten die gleichen Werte für alle Anzahlen an Threads: (Innerhalb der Fehlertoleranz, da Floatingpoint Operationen nicht kommutativ)

c.

Die erreichten Speedups sind in Abbildung 2 zu sehen. Es ist deutlich zu sehen, dass bei kleinen Problemgrößen der Overhead durch threading das Program stark verlangsamt. Ab einer Problemgröße von 10^5 sind positive Speedups zu erreichen, jedoch nur bis 8 Threads mit dem maximalen Speedup von ca. 2 bei 4 threads. Für 10^6 erreichen wir einen Speedup von ca. 5.3 bei 16 Threads, danach überwiegt wieder der Overhead.

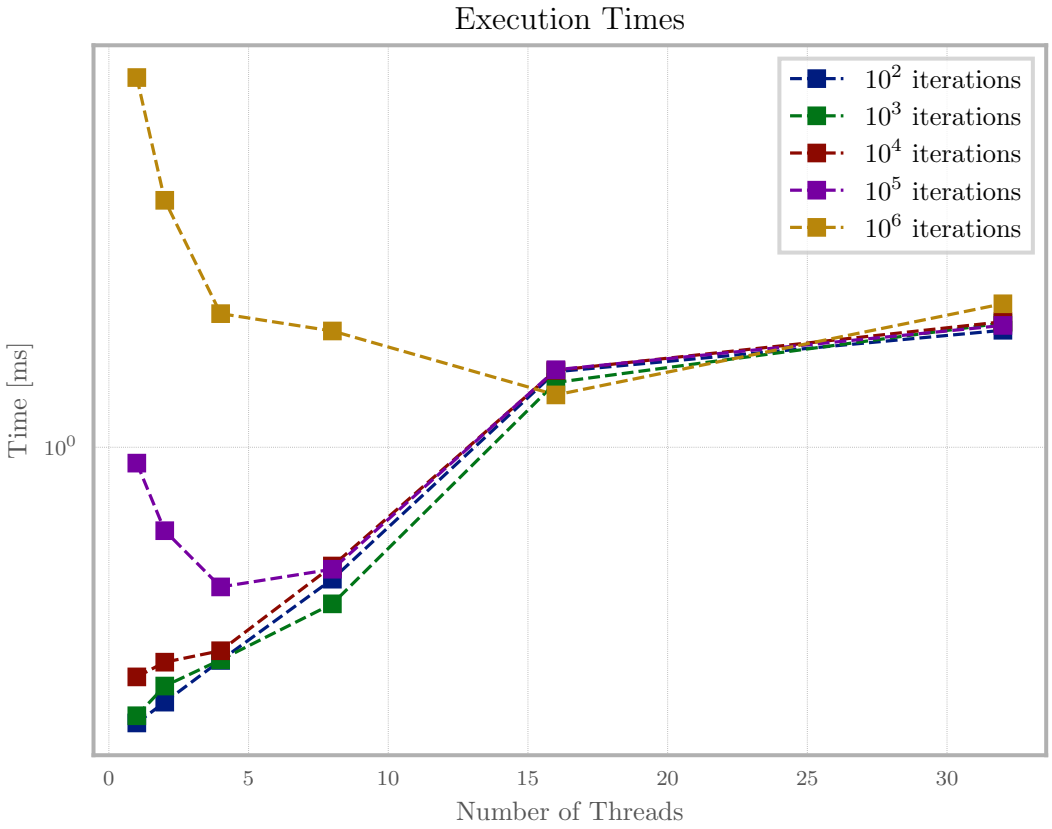

Abbildung 1: t_{compute} für verschiedene Problemgrößen, aufgrund der schlechten auflösung von `gettimeofday()` unterliegen die Wert für sehr kleine Problemgrößen starken Schwankungen

Tabelle 2: Approximierte Werte fuer π

Iterations $\log_{10}$	Values	
	π	Δ
2	3.1416009869231227	$8.333\,33 \times 10^{-6}$
3	3.1415927369231298	$8.333\,33 \times 10^{-8}$
4	3.1415926544232335	8.3344×10^{-10}
5	3.1415926535989005	$9.107\,38 \times 10^{-12}$
6	3.1415926535913021	$1.509\,02 \times 10^{-12}$

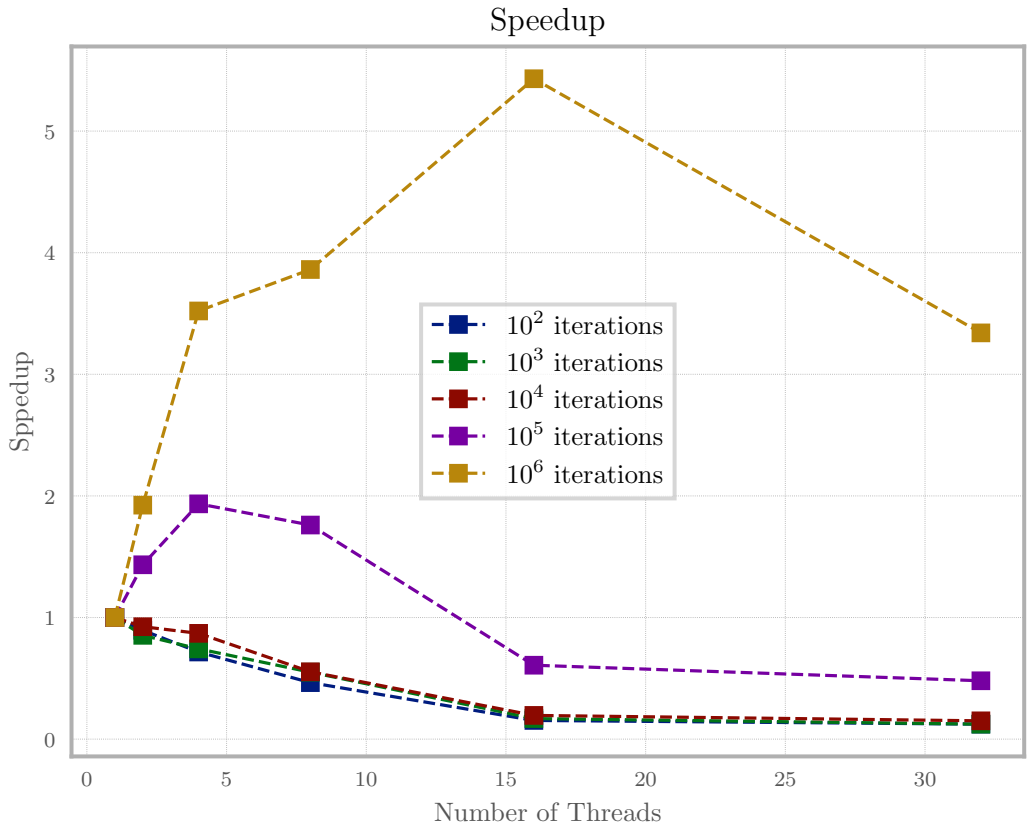

Abbildung 2: Speedups für verschiedene Problemgrößen, der Overhead der durch Threaderstellung ensteht ist besonders bei kleinen Größen zu sehen, als größter Speedup ist ein Speedup von 5 zu verzeichnen, bei 16 Threads und 10^6 Iterationen

4.2 Prozesse vs. Threads

a.

Prozesse liefern die Ressourcen, welche für die Ausführung eines Programms erforderlich sind.Ein Prozess hat einen virtuellen Adressraum, ausführbaren Code, offene Handles zu Systemobjekten, einen eindeutigen Prozessidentifier, eine Prioritätsklasse, minimaler und maximaler Arbeitsbedarf und mindestens einen Ausführungsthread. Jeder Prozess startet mit einem einzelnen Thread und kann zusätzliche Threads erstellen.

Ein Thread ist ein Objekt innerhalb eines Prozesses, das zeitlich festgelegt ausgeführt werden kann. Alle Threads eines Prozesses teilen sich den virtuellen Adressraum und Systemressourcen, besitzen Scheduling Prioritäten, lokalen Speicher, einen eindeutigen Threadidentifier und einen Strukturensatz, der den Thread Kontext speichert, bis die Ausführung des Threads festgelegt ist. Der Kontext beinhaltet den Maschinenregistersatz des Threads, den Kernel-Stack, einen Umgebungsblock und einen Benutzer-Stack im Adressraum des übergeordneten Prozesses.

b.

Die Kommunikation zwischen Threads ist programmiertechnisch einfacher als die zwischen mehrerer Prozesse. Kontextwechsel zwischen Threads sind schneller als Prozesswechsel. Das Betriebssystem kann Threads schneller stoppen und einen anderen starten, als mit zwei Prozessen.

4.3 Klassifikation nach Flynn

a.

Die Zuordnung zur Klasse der MISD-Systeme ist schwierig, da mit einem Datensatz mehrere Funktionseinheiten unterschiedliche Operationen durchführen. Genau genommen unterscheiden sich die Daten somit nach der Durchführung. Des Weiteren sind sie weniger verbreitet als MIMD- und SIMD-Systeme, welche geeigneter sind für übliche parallele Datentechniken.

b.

Ein Vektorrechner bearbeitet quasi-parallel mehrere Daten durch Pipelining. Dabei werden Maschinenbefehle in Teilaufgaben zerlegt. Diese Teilaufgaben werden für mehrere Befehle parallel ausgeführt.

Beim Feldrechner berechnen mehrere Recheneinheiten parallel die gleiche Operation auf verschiedenen Daten.