# Praktikum: C und OpenMP
## SPP WS20/21

TECHNISCHE UNIVERSITÄT DARMSTADT

**Gruppe 94: Christian Bergfried, Patrick Lutz, Lisa Zeitler**
**12. Januar 2021**

## Aufgabe 2

a)

i: private

j: shared

g1: private

g2: shared

b)

p: private

g1: shared

g2: shared

## Aufgabe 3

Die Laufzeit von `dotproduct` ist in Tabelle 1 zu sehen. Die sequenzielle Ausführung ist ungefähr so schnell wie mit einem einzelnen Thread. Mit steigender Thread-Anzahl sinkt auch die benötigte Laufzeit, bis sie bei 16 Threads aufgrund des Overheads wieder steigt.

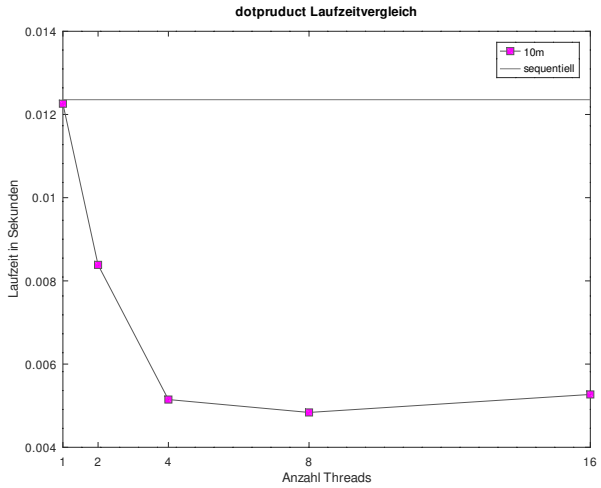


Abbildung 1: Visualisierung der Daten aus Tabelle 1.

Tabelle 1: Laufzeit von `dotproduct` für Vektoren der Größe $10 \cdot 10^6$ in Sekunden.

| Anzahl Threads | Laufzeit |
|---|---|
| Sequenziell | 0.0123552500 |
| 1 | 0.0122590400 |
| 2 | 0.0083849640 |
| 4 | 0.0051476630 |
| 8 | 0.0048374750 |
| 16 | 0.0052699860 |

## Aufgabe 4

Die Laufzeit von `heated-plate-parallel` ist in Tabelle 2 zu sehen. Die längere Laufzeit für 32 Threads ist damit zu begründen, dass der Lichtenberg-Cluster pro Node (die wir benutzen dürfen) nur 16 Kerne besitzt und somit die 32 Threads erreicht werden indem 2 Threads auf jeweils einem Kern rechnen. Dies führt zu wartenden Threads und somit der längeren Laufzeit.

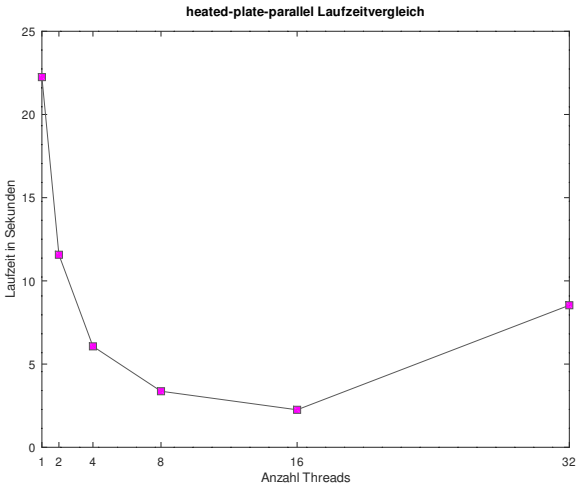


Abbildung 2: Visualisierung der Daten aus Tabelle 2.

Tabelle 2: Laufzeit von `heated-plate-parallel` für unterschieliche Thread Anzahlen.

| Anzahl Threads | Laufzeit |
|---|---|
| 1 | 22.2451 |
| 2 | 11.5705 |
| 4 | 6.0651 |
| 8 | 3.3683 |
| 16 | 2.2527 |
| 32 | 8.6349 |

## Aufgabe 5

```
int a = 0;
#pragma omp parallel private(a)
{
    a++;
    printf("%d\n", a);
}
```

Die Ausgabe des oben stehenden Codeausschnitts besteht aus n-1 mal die “1”, wobei “n” die Anzahl der Threads ist, sowie ein mal den Wert “n+1”. Das liegt daran, dass durch `#pragma omp parallel private(a)` jeder Thread seine eigenes `a` erstellt bekommt. Ein Aufruf von zum Beispiel

```
printf("I am thread %d and a is at %d",omp_get_thread_num(), &a);
```

zeigt, dass alle `a`s an einer anderen Stelle liegen, die auch zum originalen `a` vor der parallelen Umgebung unterschiedlich ist. Bei Eintritt in die parallele Umgebung wird beim Masterthread der Wert von `a` auf ”n”gesetzt und bei den anderen Threads auf “0”. Jedenfalls sind das die Zahlen, die unsere modernen Computer interpretieren, denn laut der offiziellen OpenMP Dokumentation, Seite 192ff., ist `a` in der parallelen Umgebung nun undefiniert [1] und gibt keine sinnvollen Werte zurück. Daher hat die Zeile `int a = 0;` auch keinen Einfluss auf das `a` in den Threads. Um `int a = 0;` zu übergeben, muss `firstprivate` verwendet werden.

## Aufgabe 6

Das Programm zur Multiplikation von Matrizen ist in `matMul.c` zu finden. Die Makefile wurde ebenfalls angepasst.
b)
Die Variablen werden wiefolgt klassifiziert:

N,M,P (matrix size): shared,
A,B,C (matrix): shared,
i,j,k (counter): private,
threads: private.

Die Ergebnisse der Aufgabenteile c) und d) sind in Tabelle 3 bzw. der dazugehörigen Abbildung zu sehen. Aufgrund einer nicht

dynamischen Allokierung, analog zu Aufgabenteil 4, können nur begrenzt große Matrizen erstellt werden, weshalb eine Größe von $M = 550$, $N = 500$ und $P = 530$ gewählt wurde.

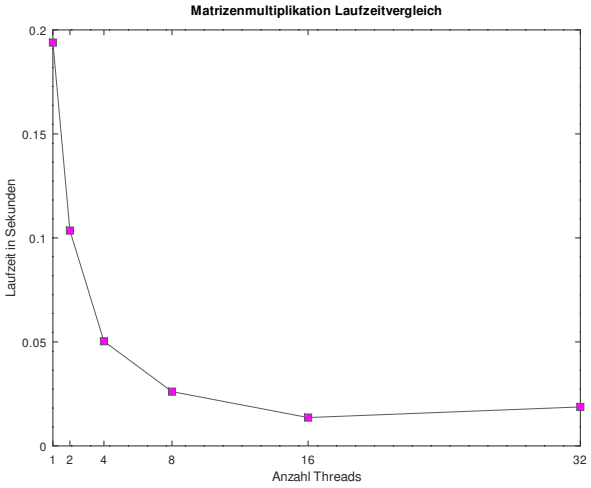


Abbildung 3: Visualisierung der Daten aus Tabelle 3.

Tabelle 3: Laufzeit der Matrixmultipliaktion für Matrizen der Größe $M, N, P = 550, 500, 530$.

| Anzahl Threads | Laufzeit |
|---|---|
| Sequenziell | 0.197666 |
| 1 | 0.193926 |
| 2 | 0.103533 |
| 4 | 0.050293 |
| 8 | 0.026104 |
| 16 | 0.013637 |
| 32 | 0.018740 |

## Literatur


[1] *OpenMP Application Programming Interface*. 2015. URL: `https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf`.