Parallele und Verteilte Systeme: WS20/21 – Übung 1 – Gruppe C

Lisa Piekarski, Klara Wichmann, Jakob Béla Ruckel

2020-11-19

README

A Makefile has been provided. From this directory, simply run `make` to build the binaries and run `make test` to run automated tests for `matmult`.

Teil 1

Aufgabe 1

Siehe `threadID.cpp`.

Aufgabe 2

```
[user@machine pvs-01]$ make threadID
g++ -fopenmp -Wall -o threadID threadID.cpp
[user@machine pvs-01]$ ./threadID
Hello from thread 0
Number of threads: 1
This task took 0.000030 seconds
```

Aufgabe 3

Example Tests:

- 10000 threads: `This task took 0.863448 seconds`
- 20000 threads: `This task took 1.898515 seconds`
- 25000 threads: `libgomp: Thread creation failed ...`
- 80000 threads: `Segmentation fault (core dumped)`

Explanation:

Increasing the `num_threads` parameter simultaneously increases the execution time. The threads are printed in random order. If the number of threads is increased to 25000 and more, the program does not create threads because the resource is “temporarily unavailable”. If the number of threads is increased to about 80000, it causes a segmentation fault.

Teil 2

Aufgabe 1

Wir haben uns dazu entschieden, hier zu parallelisieren:

```
#pragma omp parallel for collapse(2)
  for (int i = 0; i < d1; i++)
    for (int j = 0; j < d3; j++)
      for (int k = 0; k < d2; k++)
        C[i][j] += A[i][k] * B[k][j];
```

An dieser Stelle wird die meiste Rechenzeit aufgewendet, da es sich hier um drei verschachtelte `for`-Schleifen je über die Dimensionen handelt, d.h. bei $n \times n$-Matrizen läuft die Addition in der letzten Zeile ganze n^3 mal.

Dass dies der sinnvollste Ansatz ist, zeigen auch die weiteren Experimente, siehe dazu Aufgabe 4.

Weiterhin wurden die Deklaration der Schleifenvariablen `i`, `j` und `k` in die Schleifen verlagert. Das hat hier nicht primär mit Code-Stil zu tun, denn sonst werden ein und dieselben Schleifenvariablen von verschiedenen Threads gegenseitig überschrieben. Meist stürzt das Programm dann mit Segfault ab:

```
[user@machine pvs-01]$ diff -u matmult.cpp.orig matmult.cpp
--- matmult.cpp.orig    2020-11-19 00:15:49.947726884 +0100
+++ matmult.cpp 2020-11-19 00:17:24.027729628 +0100
@@ -68,11 +68,12 @@

 void matmult_parallel(float **A, float **B, float **C, int d1, int d2,
     int d3) {
   printf("Perform parallel matrix multiplication...\n");
+  int i, j, k;

 #pragma omp parallel for collapse(2)
-  for (int i = 0; i < d1; i++)
-    for (int j = 0; j < d3; j++)
-      for (int k = 0; k < d2; k++) {
+  for (i = 0; i < d1; i++)
+    for (j = 0; j < d3; j++)
+      for (k = 0; k < d2; k++) {
         C[i][j] += A[i][k] * B[k][j];
       }
 }
[user@machine pvs-01]$ make matmult
g++ -fopenmp -Wall -o matmult matmult.cpp
[user@machine pvs-01]$ ./matmult 10 10 10 test
Matrix sizes C[10][10] = A[10][10] x B[10][10]
Testing matrix equality function...
Perform serial matrix multiplication...
Serial multiplication took 0.000006 seconds.
Perform parallel matrix multiplication...
Segmentation fault (core dumped)
```

Aufgabe 2

Dazu wurde die Matrixmultiplikation in zwei Funktionen aufgeteilt, wobei die vorgegebene serielle Variante als Referenz dient. Wird das Programm mit Parameter `test` ausgeführt (z.B. `./matmult 12 34 56 test`), werden beide Varianten ausgeführt. Eine Vergleichsfunktion `mat_equal` vergleicht dann die Ergebnisse.

Bei Fehlern schlagen entsprechende `assert()`ations fehl. Die Vergleichsfunktion wird ebenfalls getestet.

Alternativ kann auch mit `diff` getestet werden, dass der Output beider Varianten identisch ist, bis auf eine Zeile Debug-Output:

```
[user@machine pvs-01]$ ./matmult 123 456 789 serial > serial.output
[user@machine pvs-01]$ ./matmult 123 456 789 parallel > parallel.output
[user@machine pvs-01]$ ls -hl *output
-rw-r--r-- 1 jakob jakob 4.5M Nov 19 01:43 parallel.output
-rw-r--r-- 1 jakob jakob 4.5M Nov 19 01:43 serial.output
[user@machine pvs-01]$ diff serial.output parallel.output
2c2
< Perform serial matrix multiplication...
---
> Perform parallel matrix multiplication...
```

Aufgabe 3

Wir haben den Code auf drei Maschinen je dreimal mit folgendem Aufruf getestet und die Laufzeiten protokolliert:

`./matmult 1500 1500 1500 test`

CPU / Kerne / Threads	Dauer Seriell	Dauer Parallel	Speedup
Ryzen 1700 / 8 / 16	21.07 sec	2.36 sec	8.93
	20.55 sec	2.40 sec	8.56
	20.58 sec	2.42 sec	8.50
i5-3570K / 4 / 4	36.64 sec	13.17 sec	2.78
	38.15 sec	15.84 sec	2.41
	36.69 sec	15.40 sec	2.38
i5-3320M / 2 / 4	41.90 sec	16.84 sec	2.49
	42.07 sec	16.91 sec	2.49
	41.97 sec	17.02 sec	2.47

Aufgabe 4

Erster Ansatz `collapse(3)` statt `collapse(2)`

Bei drei verschachtelten Schleifen bietet sich natürlich auf den ersten Blick folgendes `pragma` an:

```
#pragma omp parallel for collapse(3)
```

Allerdings wird in der Lösung die innerste Schleife nicht mit in das `collapse` aufgenommen. Bei `collapse(3)` kann eine race condition auftreten, da zwei oder mehr verschiedene Threads dies gleichen Indizes `i` und `j` haben können und dann gleichzeitig mit += in die Matrix `C` schreiben.

Tatsächlich tritt der Fehler sogar in der Praxis auf:

```
[user@machine pvs-01]$ sed -i "s/collapse(2)/collapse(3)/" matmult.cpp
[user@machine pvs-01]$ make matmult
g++ -fopenmp -Wall -o matmult matmult.cpp
[user@machine pvs-01]$ ./matmult 2000 2000 2000 test
Matrix sizes C[2000][2000] = A[2000][2000] x B[2000][2000]
Testing matrix equality function...
Perform serial matrix multiplication...
Serial multiplication took 66.321047 seconds.
Perform parallel matrix multiplication...
Parallel multiplication took 10.804740 seconds.
matmult: matmult.cpp:155: int main(int, char**): Assertion `mat_equal(C
    , C_parallel, d1, d3)' failed.
Aborted (core dumped)
```

Dass nur zwei Loops Teil des `collapse` sind, hat kaum Auswirkungen auf die Schnelligkeit, denn solange `i*j` größer gleich der Anzahl verfügbarer Threads ist, hat jeder Thread “genug zu tun”.

Einfaches `omp parallel`

Folgendes Experiment:

```
#pragma omp parallel
  for (int i = 0; i < d1; i++)
    for (int j = 0; j < d3; j++)
      for (int k = 0; k < d2; k++)
        C[i][j] += A[i][k] * B[k][j];
```

sorgt zwar auch für parallele Ausführung, allerdings führt hier jeder der *n* Threads die Matrixmultiplikation durch und addiert die Ergebnisse auf `C`, wodurch die Ergebnisse um den Faktor *n* größer sind – und natürlich ist kein Speedup möglich.

Dass dies tatsächlich passiert, zeigt die folgende Beobachtung auf einem System mit 4 Threads:

```
[user@machine pvs-01]$ ./matmult 2 2 2 serial
Matrix sizes C[2][2] = A[2][2] x B[2][2]
Perform serial matrix multiplication...
Matrix A:
   3.0      6.0
   7.0      5.0
Matrix B:
   3.0      5.0
   6.0      2.0
Matrix C:
  45.0     27.0
  51.0     45.0

Done.
[user@machine pvs-01]$ ./matmult 2 2 2 parallel
Matrix sizes C[2][2] = A[2][2] x B[2][2]
Perform parallel matrix multiplication...
Matrix A:
   3.0      6.0
   7.0      5.0
Matrix B:
   3.0      5.0
   6.0      2.0
Matrix C:
 135.0    108.0
 204.0    180.0

Done.
```