Laboratory for Parallel Programming
Prof. Dr. Felix Wolf

TECHNISCHE UNIVERSITÄT DARMSTADT

Daniel Lorenz, lorenz@cs.tu-darmstadt.de

Systemnahe und parallele Programmierung (WS 17/18)

Praktikum: C und OpenMP

Die Lösungen müssen bis zum 10. Dezember 2017 in Moodle submittiert werden. Anschließend müssen sie ihre Lösung einem Tutor vorführen. Das Praktikum wird benotet. Im Folgenden einige allgemeine Bemerkungen, die für alle Aufgaben auf diesem Blatt gelten:

- Wenn eine Teilaufgabe das Erstellen eines Programms erfordert, muss auch immer ein `Makefile` erstellt werden, dass dieses Programm baut. Ob je ein `Makefile` pro Teilaufgabe oder nur Eines mit mehreren Regeln für jede Teilaufgabe erstellt werden, bleibt Ihnen überlassen.
- Oft bauen folgende Aufgaben auf vorherigen Aufgaben auf und erfordern nur eine Modifikation eines früheren Programms. In diesen Fällen soll auch die frühere Version des Programm von der vorherigen Aufgabe erhalten werden.
- Dynamisch allokierter Speicher muss frei gegeben werden bevor das Programm endet.
- Es folgt eine Liste von Funktionen der C Standardbibliothek, die bei der Lösung hilfreich sein könnten. Bitte informieren Sie sich im Internet über die genaue Funktion und Verwendung dieser Funktionen.
  - `srand()`
  - `rand()`
  - `atol()`
- Die Programme müssen auf dem Lichtenbergcluster kompilierbar und ausführbar sein. Zeitmessungen müssen auf dem Lichtenbergcluster ausgeführt werden. Es kann ein Beispieljobscript von Moodle heruntergeladen werden. Es enthält auch Hinweise, wie sie es für ihre Bedürfnisse anpassen können.
- Die Zeit kann mit Hilfe der Funktion `omp_get_wtime()` bestimmt werden.

Aufgabe 1

(17 Punkte) In der ersten Aufgabe soll ein Programm zur Berechnung von Primzahlen erstellt werden. Dazu wird ein Array `A` mit N Bytes Länge allokiert und die Zahlen ausgestrichen, die keine Primzahlen sind. Das Byte mit Index i speichert dabei ob die Zahl i ausgestrichen wurde oder nicht. `A[i] == 0` bedeutet, dass `i` (noch) nicht ausgestrichen wurde, `A[i] != 0` bedeutet, dass `i` keine Primzahl ist.

a) (4 Punkte) In der ersten Teilaufgabe soll eine serielles Programm zur Berechnung der Primzahlen erstellt werden. Die Länge des Arrays, und damit das Maximum, bis zu dem die Primzahlen berechnet werden, soll durch ein Programmargument beim Programmaufruf übergeben werden. Nach der Berechnung der Primzahlen sollen diese auf dem Bildschirm ausgegeben werden.

b) (3 Punkte) In dieser Aufgabe sollen sie die Berechnung der Primzahlen aus der ersten Teilaufgabe mit OpenMP parallelisieren. Verwenden Sie dazu das OpenMP `for`-Konstrukt und einen statischen Schedule. Die Blockgröße soll als zweites Programmargument (nach der Länge des Arrays) übergeben werden. Messen sie die Laufzeit der parallelen OpenMP Region und geben sie diese auf dem Bildschirm aus.

c) (5 Punkte) Messen sie sie Laufzeit der Parallelen OpenMP Region aus der vorherigen Aufgabe für für die Berechnung von Primzahlen bis zu einem Maximum von 100 000 000 mit 2, 4, 8 und 16 Threads und Blockgrößen von 1, 10, 100, 1000, 10000, 100000 und 1000000. Erklären sie die Messergebnisse.

d) (5 Punkte) Erstellen sie eine weitere Variante ihres Programmes, dass statt einem statischen Schedule einen dynamischen verwendet. Führen sie dieselben Messungen mit dem dynamischen Schedule durch und erklären sie die Ergebnisse.

Aufgabe 2

(13 Punkte) In der zweiten Aufgabe soll ein Mergesort-Sortieralgorithmus implementiert werden, der ein Array von Integer-Zahlen sortiert.

- Das zu sortierende Array wird in zwei gleiche Hälften geteilt. Jede der Hälften wird rekursiv mit dem Mergesort-Algorithmus sortiert.
- Die Rekursion endet, wenn das Array nur noch aus einem Element besteht. Dieses ist trivialerweise sortiert.
- Nachdem beide Arrayhälften sortiert sind, werden sie zusammengeführt.

Das Zusammenführen funktioniert folgendermaßen:

- Es wird ein Hilfsarray benötigt, in das die sortierte Zahlenfolge geschrieben wird.
- Für jede Hälfte muss sich der Algorithmus das Element merken, dass als Nächstes in das sortierte Array eingefügt wird. Zu Beginn ist dies das jeweils erste Element jeder Hälfte.
- Die beiden als nächsten einzufügenden Element werden verglichen. Der kleinere Element wird in das Hilfsarray eingefügt.
- Dies wird so lange wiederholt, bis alle Elemente aus beiden Hälften in das Hilfsarray eingefügt wurden.

Abschließend muss die sortierte Zahlenfolge aus dem Hilfsarray wieder in das zu sortierende Array kopiert werden.

a) (5 Punkte) Als erster Schritt soll wieder eine serielle Version des Mergesort erstellt werden. Der Sortieralgorithmus soll in der Funktion `merge_sort( int *array, int* help, unsigned length )` implementiert werden. Die Länge des Arrays soll dem Programm als Programmargument übergeben werden. Ein Array der entsprechenden Länge muss danach allokiert und mit Zufallszahlen gefüllt werden. Nach dem Sortieren soll überprüft werden, dass das Array sortiert ist, indem jeweils zwei aufeinander folgende Zahlen verglichen werden.

b) (3 Punkte) Der Sortieralgorithmus aus der vorherigen Aufgabe soll nun mit OpenMP Tasks parallelisiert werden. Dazu soll der rekursive Aufruf der Funktion `merge_sort` in einem eigenen Task geschehen. Achten Sie darauf, dass die Tasks von mehreren Threads bearbeitet werden und dass der Sortieralgorithmus nicht mehrfach initiiert wird. Messen sie die Zeit die der Sortieralgorithmus benötigt und geben sie diesen auf dem Bildschirm aus.

c) (5 Punkte) Verwenden sie eine if-clause, um die Erzeugung neuer Tasks zu verhindern, wenn das zu sortierende Teilarray kürzer als eine Zahl N ist. Sortieren sie 100 Millionen Elemente mit 4 Threads und $N \in \{10, 100, 1000, 10000, 100000, 1000000\}$ laufen und stellen sie das Ergebnis graphisch dar. Erklären sie das Ergebnis.

Aufgabe 3

(9 Punkte) In dieser Aufgabe sollen das folgende Programm analysieren und parallelisieren. Den Quellcode des Programmes können sie auch von Moodle herunterladen.

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  #define LENGTH 1000000
5
6  int sum( int* a, unsigned length )
7  {
8      int result = 0;
9      for( unsigned i=0; i<length; i++ )
10     {
11         result += a[i];
12     }
13     return result;
14 }
15
16 void shift( int* a, unsigned length, unsigned offset )
17 {
18     for( unsigned i=0; i<length-offset; i++ )
19     {
20         a[i] = a[i+offset];
21     }
22     for( unsigned i=length-offset; i<length; i++)
23     {
24         a[i] = 0;
25     }
26 }
27
28 int hash( int* a, unsigned length )
29 {
30     int hash = 1;
31     for( unsigned i=0; i<length; i++ )
32     {
33         switch( hash % 3 )
34   {
35   case 0:
36     hash += a[i];
37     break;
38   case 1:
39     hash *= a[i];
40     break;
41   default:
42     hash -= a[i];
43   }
44   hash %= 1000;
45     }
46     return hash;
47 }
48
49 void init( int* a, unsigned length, int base, int offset )
50 {
51     for( unsigned i=0; i<length; i++ )
```

```
52      {
53          a[i] = base + i * offset;
54      }
55  }
56
57  int main()
58  {
59      int* a = malloc( LENGTH * sizeof( int ) );
60      int* b = malloc( LENGTH * sizeof( int ) );
61      int* c = malloc( LENGTH * sizeof( int ) );
62      int* d = malloc( LENGTH * sizeof( int ) );
63
64      init( a, LENGTH, 1, 1 );
65      init( b, LENGTH, 2, 2 );
66      init( c, LENGTH, 1, 1 );
67      init( d, LENGTH, 0, -1 );
68
69      int result = sum( a, LENGTH );
70      shift( a, LENGTH, 1000 );
71      result += hash( a, LENGTH );
72      shift( b, LENGTH, 2 );
73      result += sum( b,LENGTH );
74      result += hash( b, LENGTH );
75      result += hash( c, LENGTH );
76      shift( d, LENGTH, 623 );
77      result += hash( d, LENGTH );
78
79      free( a );
80      free( b );
81      free( c );
82      free( d );
83
84      printf( "Result: %d\n", result );
85      return 0;
86  }
```

a) (4 Punkte) Zunächst sollen sie die Parallelisierbarkeit der folgenden Funktionen untersuchen.

- sum
- shift
- hash
- init

In welcher diese Funktionen existieren iterationsübergreifende Datenabhängigkeiten? Wenn iterationsübergreifende Datenabhängigkeit existieren, geben sie bitte an, welche Variable die Abhängigkeit erzeugt und um welchen Typ von Abhängigkeit es sich handelt.

b) (3 Punkte) Parallelisieren sie bitte die Schleifen von drei Funktionen mit OpenMP. Beachten sie dabei die Korrektheit des Programms.

c) (2 Punkte) Anstatt die Schleifen zu parallelisieren soll in dieser Teilaufgabe, die `main`-Funktion parallelisiert werden. Dazu teilen sie die Arbeit in 4 Blöcke auf, so dass es außer beim Zugriff auf die Variable `result` keine Datenabhängigkeiten gibt. Parallelisieren sie diese 4 Blöcke mit OpenMP Tasks oder Sections. Stellen sie sicher, dass der Zugriff auf die gemeinsam genutzte Variable `result` jeweils exklusiv ist.

Aufgabe 4

(6 Punkte)

Gegeben sei das folgende Programm

```
1  int count = 0;
2  double val = 0.0;
3
4  double* foo( int c, double* g, double h )
5  {
6    if ( h < c ) return g;
7    else return &val;
8  }
9
10 void bar(double* a, double *z, int n)
11 {
12    static int cnt = 0;
13    int i, j;
14
15    #pragma omp parallel for firstprivate (cnt, val)
16    for( j=0; j<n; j++)
17    {
18        int* res = foo( count, &a[j], z[j] );
19        for (i=0; i<j; i++)
20        {
21      cnt++;
22        }
23        count = count + *res + cnt;
24    }
25 }
```

a) (2 Punkte) Bitte bestimmen Sie welche der folgenden Variablen in `foo()` private oder shared sind:

- count
- val
- g
- *g

b) (4 Punkte) Bitte bestimmen Sie welche der folgenden Variablen in `bar()` private oder shared sind:

- count
- val
- cnt
- res
- *res
- i
- j
- a

Aufgabe 5

(10 Punkte) In dieser Aufgabe soll ein Mandelbrotfeld berechnet werden. Für jeden Punkt $C = (x, yi)$ einer Fläche, wird dazu folgende rekursive Iteration ausgeführt:
$z_0 = (0, 0i)$
$z_{n+1} = z_n * z_n + C$
bis $norm(z) >= 16$ ist oder bis eine vordefinierte maximale Anzahl an Iterationen erreicht wurde. Für jeden Punkt wird die Anzahl der erreichten Iterationen in einem Feld gespeichert. Wenn man in Abhängigkeit von der Anzahl der Iterationen jedem Punkt eine Farbe zuweist erhält man ein Fraktal.
Zur Lösung auf dem Computer wird dass Problem diskretisiert und eine feste Anzahl an Punkten (800x800) berechnet.

a) (3 Punkte) Zunächst soll eine Klasse erstellt werden, die eine Komplexe Zahl implementiert. Diese Klasse benötigt 3 Funktionen:

- `add` Addiert eine andere komplexe Zahl zu dieser Zahl.
- `mult` Multipliziert eine andere komplexe Zahl zu dieser Zahl.
- `norm` Berechnet den Abstand des Punktes vom Ursprung.

b) (3 Punkte) Erstellen Sie ein Programm, das ein Mandelbrotfeld von 800x800 Pixeln berechnet. x_{min}, x_{max}, y_{min}, y_{max} und die maximale Anzahl an Iterationen sollen als Programmargumente übergeben werden. Anschließend soll das Feld als Graphikdatei gespeichert werden. Zum Speichern können Sie die Funktion `ppmwrite` verwenden, die sie von Moodle herunterladen können. Sie erstellt Bilder im ppm-Format, die z.B. mit GIMP angesehen werden können.

c) (4 Punkte) Parallelisieren sie die Berechnung des Mandelbrotfeldes und messen sie die Ausführungszeit der parallelen Region mit 1, 2, 4, 8 und 16 Threads. Stellen sie die Laufzeit graphisch dar.