Laboratory for Parallel Programming
Prof. Dr. Felix Wolf

TECHNISCHE UNIVERSITÄT DARMSTADT

Mohammad Norouzi, Sebastian Rinke

Systemnahe und parallele Programmierung (WS 18/19)

Praktikum: C und OpenMP

Die Lösungen müssen bis zum 11. Dezember 2018, 13:30 Uhr, 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:

- Es gibt ein vorgegebenes Makefile welches alle Programme kompiliert die Sie entwickeln sollen.
- Dynamisch allokierter Speicher muss freigegeben 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()`
  - `atoi()`
- Die Programme müssen auf dem Lichtenberg Cluster kompilierbar und ausführbar sein. Alle Zeitmessungen müssen auf dem Lichtenberg Cluster ausgeführt werden. Es kann ein Beispieljobscript von Moodle heruntergeladen werden. Es enthält auch Hinweise, wie Sie es für Ihre Bedürfnisse anpassen können.
- Vorgegebener C/C++ Code in den Dateivorlagen darf nicht geändert werden und muss wie gegeben verwendet werden.
- Die Zeit kann mit Hilfe der Funktion `omp_get_wtime()` bestimmt werden.
- Alle Lösungen die keinen Programmcode erfordern, bitte zusammen in einer PDF Datei einreichen.
- Laden Sie alle Dateien in einem tar Archiv in Moodle hoch.

Aufgabe 1

(3 Punkte) Schreiben Sie ein Programm in der Datei `hello-openmp.c` das zunächst die maximale Anzahl Threads ausgibt, die in einem *parallel construct* genutzt werden können. Setzen Sie danach diese maximale Anzahl auf einen anderen Wert. Schließlich soll ein Team von Threads erstellt werden, von dem jeder Thread `Hello` gefolgt von seinem Identifikator ausgibt.

Die OpenMP Referenz könnte dabei hilfreich sein:
`https://www.openmp.org/wp-content/uploads/OpenMP-4.5-1115-CPP-web.pdf`

Aufgabe 2

(7 Punkte) Gegeben sei das folgende Programm:

```
1  int g1, g2;
2
3  int foo(int p)
4  {
5      return p + g1 + g2;
6  }
7
8  int main()
9  {
10     int i, j;
11
12 #pragma omp parallel for private(g1)
13     for (i=0; i<10; i++)
14     {
15         j += g1 + g2 + i;
16         j += foo( g2 );
17     }
18     return 0;
19 }
```

a) Geben Sie an ob die folgenden Variablen *private* oder *shared* sind im Konstrukt `omp parallel for`:

1. i:
2. j:
3. g1:
4. g2:

b) Geben Sie an ob die folgenden Variablen *private* oder *shared* sind in der Funktion `foo`.

1. p:
2. g1:
3. g2:

Aufgabe 3

(11 Punkte) Das Skalarprodukt zweier Vektoren $\vec{x}$ und $\vec{y}$ der Länge n berechnet sich aus

$$\vec{x} \cdot \vec{y} = \sum_{i=1}^{n} x_i \cdot y_i \tag{1}$$

In dieser Aufgabe entwickeln Sie eine sequenzielle und eine parallele Version zur Berechnung des Skalarprodukts. Nutzen Sie zur Lösung die Dateivorlage `dotproduct.c`.

a) (4 Punkte) Die Funktion `test01` soll das Skalarprodukt sequenziell berechnen.

b) (4 Punkte) Die Funktion `test02` soll das Skalarprodukt parallel mit OpenMP unter Nutzung des *loop constructs* und der *reduction clause* berechnen.

c) (3 Punkte) Messen Sie die sequenzielle Ausführungszeit (`test01`) und die Zeit der parallelen Berechnung (`test02`) mit $1, 2, 4, 8$ und 16 Threads, für Vektoren der Länge 10.000.000. Stellen Sie die Ergebnisse grafisch dar.

Aufgabe 4

(19 Punkte) Quicksort ist ein Sortieralgorithmus basierend auf dem Teile-und-Herrsche Prinzip. Die Schritte sind wie folgt:

- Wähle ein Element (Pivotelement p)
- Bewege alle Elemente die kleiner als p sind nach links vor p und alle Elemente die größer oder gleich p sind nach rechts hinter p. Nach dieser Umordnung (Partitionierung) ist das Pivotelement bereits an der richtigen finalen Position im Feld.
- Wende die zwei vorherigen Schritte nun rekursiv auf das linke und das rechte Teilfeld an.

Die Rekursion endet für Felder mit keinem oder einem Element.

a) (6 Punkte) Implementieren Sie Quicksort in einem sequenziellen C Programm das die Elemente aufsteigend sortiert. Nutzen Sie dazu die Dateivorlage `quickSortSequential.c`. Die Anzahl der zu sortierenden Array Elemente soll als Befehlszeilenparameter beim Start des Programms übergeben werden. Zu Beginn sollen alle Elemente mit Zufallszahlen initialisiert werden.

b) (1 Punkt) Prüfen Sie am Ende des Programms ob das Array korrekt sortiert wurde. Jedes Element sollte kleiner oder gleich dem nächsten Element sein.

c) (8 Punkte) Parallelisieren Sie Ihr sequenzielles Quicksort Verfahren nun mit OpenMP in der Datei `quickSortParallel.c`. Stellen Sie sicher, dass keine neuen OpenMP Tasks für Arrays mit weniger als 100 Elementen kreiert werden.

d) (3 Punkte) Messen Sie die Laufzeit von Ihrem sequenziellen und parallelen Quicksort für Arrays der Größe 100.000, 1.000.000, und 10.000.000. Stellen Sie die Ergebnisse grafisch dar. Nutzen Sie 16 Threads für die parallele Ausführung.

e) (1 Punkt) Angenommen Sie möchten auch das Initialisieren des Arrays mit Zufallszahlen parallelisieren. Sie möchten dazu die Funktion rand() der C-Standardbibliothek verwenden. Welche Eigenschaft müsste diese Funktion erfüllen?

Aufgabe 5

(10 Punkte) Gegeben ist ein sequenzielles C++ Programm in der Datei `heated-plate-parallel.cpp`, dass die Wärmeleitungsgleichung für ein 2-dimensionales Gebiet löst.

a) (7 Punkte) Parallelisieren Sie das vorgegebene Programm mit OpenMP *loop constructs* in der Datei `heated-plate-parallel.cpp`. Es genügt bei verschachtelten Schleifen nur die Äußere zu parallelisieren.

b) (3 Punkte) Messen Sie die Laufzeit von Ihrem parallelen Algorithmus für 1, 2, 4, 8, 16 und 32 Threads. Stellen Sie die Messergebnisse grafisch dar.

Aufgabe 6

(5 Punkte) Was gibt `printf()` im folgenden Codeausschnitt aus? Begründen Sie Ihre Antwort.

```
1 int a = 0;
2 #pragma omp parallel private(a)
3 {
4     a++;
5     printf("%d", a);
6 }
```