Alexander Hück, alexander.hueck@tu-darmstadt.de, Sebastian Kreutzer, Yannic Fischler

Systemnahe und parallele Programmierung (WS 21/22)

Praktikum: C/C++ und OpenMP

Die Lösungen müssen bis zum 14. Dezember 2021, 15:00 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 jeweils ein vorgegebenes Makefile welches das Programm kompiliert, das 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.
  - `atoi()`
  - `calloc()`
- Aus der C++-Standardbibliothek wird der Containtertyp `std::vector` benutzt[1].
- Die Programme müssen auf dem Lichtenberg Cluster kompilierbar und ausführbar sein. Alle Zeitmessungen müssen auf dem Lichtenberg Cluster ausgeführt werden. Siehe Anleitung zur Nutzung des Clusters.
- Vorgegebener C/C++ Code in den Dateivorlagen darf nicht geändert werden und muss wie gegeben verwendet werden.
- Zeitmessungen sind bereits integriert.
- Alle Lösungen, die keinen Programmcode erfordern, bitte zusammen in einer PDF Datei einreichen.
- Laden Sie alle Dateien in einem zip-Archiv in Moodle hoch.

[1] `https://en.cppreference.com/w/cpp/container/vector`

Aufgabe 1

Parallel daxpy (20 Punkte) Bei der `daxpy`-Operation berechnet man aus den **`double`**-Vektoren $\vec{y}$ und $\vec{x}$ der Länge n sowie dem Skalar a:

$$\vec{y} = \vec{y} + \vec{x} \cdot a \tag{1}$$

In dieser Aufgabe entwickeln Sie eine sequenzielle und eine parallele Version zur Berechnung von `daxpy`. Nutzen Sie zur Lösung die Dateivorlage `daxpy.c`. Das Programm wird mit einem Aufrufparameter $t \in \mathbb{N}$ gestartet, der angibt, wie viele Threads in parallelen Programmteilen genutzt werden.

a) **(4 Punkte)** Die Funktion `daxpy_ser` soll `daxpy` sequenziell berechnen.

b) **(2 Punkte)** Die Funktion `main` soll zu Beginn die Anzahl der genutzten OpenMP-Threads auf den Wert des durch die Eingabeparameter definierten Wert `nt` setzen.

c) **(4 Punkte)** Um den Erfolg der vorigen Aufgabe zu verifizieren, soll `main` anschließend auslesen, mit wie vielen Threads die OpenMP-Umgebung arbeitet und diesen Wert in einer einzigen Konsolenzeilenausgabe an den Nutzer zurückgeben.

d) **(3 Punkte)** `daxpy_par` `daxpy` parallel berechnen. Benutzen Sie beim Spawnen der parallelen Region die **`default`**`(none)`-und geben Sie die Speicherklauseln aller Variablen explizit an.

e) **(4 Punkte)** Messen Sie die sequenzielle Ausführungszeit `daxpy_ser` und die Zeit der parallelen Berechnung `daxpy_par` mit $1, 2, 4, 8$ und 16 Threads für die vorgegebene Vektorlänge 500.000.000. Stellen Sie die Ergebnisse grafisch oder tabellarisch dar.

f) **(3 Punkte)** Setzen Sie das Scheduling der parallelisierten **`for`**-Schleife auf `(`**`static`**`,3)`. Wie verhalten sich nun die Ausführungszeiten? Begründen Sie die Veränderung.

Aufgabe 2

(30 Punkte) Bei einer symmetrischen Matrix liegen die Einträge symmetrisch um die Diagonale der Matrix. Somit haben z.B. in Abbildung die Einträge a_{12} und a_{21} den gleichen Wert.

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \end{pmatrix}$$

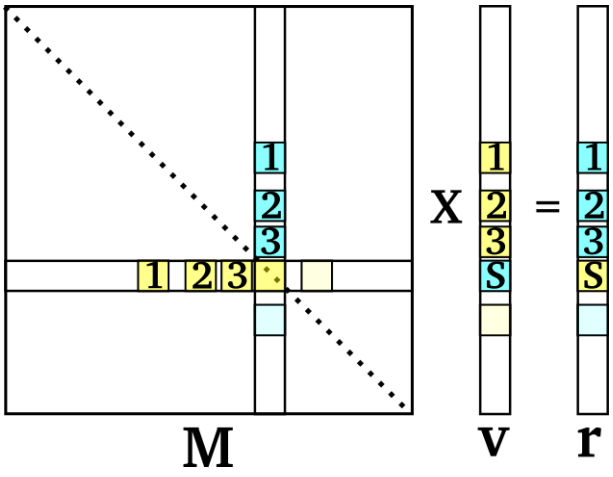
Figure 1: An der Diagonale gespiegelte Wertepaare.

Um Speicher zu sparen reicht es somit nur die blauen und schwarzen a_{ij} als so genannte untere Dreiecksmatrix zu speichern.
In dieser Aufgabe werden Sie ein Programm schreiben, welches das Produkt einer symmetrischen $n \times n$ Matrix M mit einem n-Vektor $\vec{v}$ berechnet:

$$r = M \cdot \vec{v}$$

Das Ergebnis der Multiplikation ist der n-Vektor $\vec{r}$.
Bei der Multiplikation lässt sich ausnutzen, dass sich ein bereits geladenes Element a_{ij} ebenfalls als a_{ji} verwenden lässt, was Abbildung veranschaulicht. Hat man z.B. gerade die gelbe 1 aus M geladen, so

lassen sich die Berechnungen für die blaue 1 ebenfalls ausführen. Somit gilt: Berechnet man $r_i = r_i + M_{i,j} \cdot v_i$ $(i \neq j)$ lässt sich auch $r_j = M_{j,i} \cdot v_j$ $(M_{j,i} = M_{i,j})$ ohne weitere Ladeoperation auf M berechnen.



a) **(4 Punkte)** Zu Beginn der `main`-Methode wird die Matrix `m` als zweidimensionales, dynamisches Array alloziert und jeder Eintrag mit 0 initialisiert. Lassen Sie das Programm unter dem Kommentar `//Fill with Values` folgendermaßen füllen: An der Position $[i][j]$ im unteren Dreieck steht der Wert $(i/2+20)(j/2-20)\%25$, im korrespondierenden Eintrag im oberen Dreieck der gleiche. Auf der Diagonalen $[i][i]$ steht der Wert $i+1$. Die Funktion `checkIfFullMatrixIsCorrect` prüft die Korrektheit Ihrer Initialisierung.

b) **(3 Punkte)** Implementieren Sie die Funktion `defaultMultiply`, welche die Symmetrie von M ignoriert, eine normale Matrix-Vektor-Multiplikation zeilenweise berechnet und das Ergebnis in `result` zurück gibt.

c) **(5 Punkte)** Parallelisieren Sie die Berechnung in `defaultMultiply`. Geben Sie die Speicherklauseln aller Variablen explizit an. Vergleichen Sie tabellarisch die Laufzeit bei 1, 2, 4, 8, 16 und 32 Threads.

d) **(3 Punkte)** Verwenden Sie im entsprechenden pragma die `collapse` Klausel für das Schleifennest. Können Sie damit signifikante Änderungen der Ausführungszeiten erreichen?

e) **(5 Punkte)** In der `main`-Methode soll unter dem Kommentar `Allocate a symmetric matrix` die Matrix `m2` als untere Dreiecksmatrix alloziert werden. Hierbei soll wiederum ein zweidimensionales Array verwendet werden, diesmal mit angepasster Länge der Zeilen. Füllen Sie anschließend die Matrix mit den gleichen Werten wie in der ersten Teilaufgabe.

f) **(3 Punkte)** Implementieren Sie die Multiplikation mit der symmetrischen Matrix `m2` in `symmetricMultiply` .

g) **(3 Punkte)** Parallelisieren Sie die symmetrische Multiplikation über die äußere Schleife. Hinweis: Bei der Ausnutzung der Symmetrie kann es zu Data Races in $\vec{r}$ kommen.

h) **(4 Punkte)** Messen Sie die Laufzeit bei 1, 2, 4, 8, 16 und 32 Threads und tragen sie die Ergebnisse in die bestehende Tabelle ein. Begründen Sie welche der beiden Varianten die bessere ist. Was sind Vor- und Nachteile der beiden Varianten.

Aufgabe 3

(16 Punkte) Quicksort[2] ist ein rekursiver Sortieralgorithmus, der im Durchschnitt $O(n \cdot \log n)$ Vergleiche durchführt um ein Array der Länge n zu sortieren.
Aus dem Array wird ein Eintrag als Pivotelement gewählt und das Array anschließend so aufgeteilt, dass, der linke Teil alle Elemente kleiner dem Pivotelement enthält, der rechte Teil die größeren. Nun ruft man für jedes Teil-Array Quicksort rekursiv auf. Die Rekursion endet, wenn die Teil Arrays Länge 0 erreichen. Aufgrund der Anwendung dieses "Teile und herrsche" Prinzips, lässt Quicksort sich mit dem OpenMP Task Konstrukt parallelisieren.
Eine Beispiel Implementierung ist in der Datei `quicksort.cpp` gegeben.

a) (8 Punkte) Parallelisieren Sie die Routine **`void`** `quickSort_par( ... )` mittels Tasking pragmas. Messen Sie die Laufzeit der Version ohne Pragmas **`void`** `quickSort_ser` und der parallelisierten Version mit 1, 2, 4 und 8 Threads. Wie entwickelt sich die Laufzeit?

Hinweis: Sie müssen für diese Aufgabe nur Code in den Funktionen `main` (unter dem Kommentar `Call the parallel quickSort implementation`) und `quickSort_par` hinzufügen.

Hinweis 2: Die `main`-Routine lässt die parallele Implementierung automatisch mit verschiedener Threadanzahl laufen und gibt die benötigte Laufzeit dafür an.

b) (8 Punkte) Wie erklären Sie das Laufzeitverhalten? Fügen Sie weiteren Code hinzu, der die Geschwindigkeit deutlich verbessert. Tragen Sie die Laufzeiten Ihrer verbesserten Version in die Tabelle aus Teil a) ein.

[2] https://de.wikipedia.org/wiki/Quicksort

Aufgabe 4

(9 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: