Grundlagen der Informatik III

Wintersemester 2010/2011

Lucas Rothamel, Wolfgang Heenes, Patrik Schmittat

TECHNISCHE UNIVERSITÄT DARMSTADT

3. Praktikum

Ausgabe: 01.12.2010; Abgabe: 22.12.2010, 23:59 Uhr

Abgabe der Programme: Die Abgabe der Programme erfolgt über den SVN-Server https://ics.ra.informatik.tu-darmstadt.de/svn/. Die Beantwortung der Theoriefragen erfolgt in einem eigenen Dokument (PDF).

Hinweis: Im Rahmen des Praktikums kann auch der *clientssh3.rbg.informatik.tu-darmstadt.de* der RBG benutzt werden.

Aufgabe 1: Matrixmultiplikation in Assembler

Durch die Verwendung von Matrizen wird die Berechnung von linearen Abbildungen und das Lösen von linearen Gleichungssystemen wesentlich vereinfacht. Gerade im Bereich der Computergraphik finden Matrizen einen großen Anwendungsbereich. In diesem Praktikum werden Sie sich daher näher mit der effizienten Implementation der Matrixmultiplikation beschäftigen.

Die Matrixmultiplikation für $A = (a_{ij})_{i=1..l,j=1..m}$, $B = (b_{ij})_{i=1..m,j=1..n}$ ist mathematisch definiert als:

$$A \cdot B := (c_{ij})_{i=1..l,j=1..n} \text{ , wobei } c_{ij} = \sum_{k=1}^{m} a_{ik} \cdot b_{kj}$$

a) Führen Sie zur Illustration von Hand die Matrixmultiplikation $A \cdot B$ für die Matrizen

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \text{ und } B = \begin{pmatrix} -1 & 9 & -2 \\ 3 & -11 & 5 \\ 7 & 4 & -2 \end{pmatrix}$$

durch. Alle Zwischenschritte müssen gut erkennbar sein. Bitte laden Sie diese zum Schluss als PDF hoch. Gerne können Sie Ihre Lösung gut lesbar einscannen/fotografieren und im PDF einbetten. Im C-Pool sowie in der Fachbereichsbibliothek stehen Ihnen im Bedarfsfall Kopiergeräte mit Scan-Funktion zur Verfügung.

b) Schreiben Sie ein IA32-Assemblerprogramm, welches die Matrixmultiplikation zweier quadratischer Matrizen (NxN) ausführt. N ist dabei eine natürliche Zahl. Nutzen Sie zur Definition einen .data Block mit den Matrizenwerten in den Datenfeldern **matA** und **matB** sowie *matResult* vom Typ .long, und speichern Sie sich in einer weiteren Variable **matsize** die Breite bzw. Höhe der Matrix, um darüber iterieren zu können. Weitere Variablen, mit Ausnahme von Strings, sollten Sie im .data Bereich nicht benötigen. Nutzen Sie soweit sinnvoll die **Scaled Index Adressierung**.

c) Testen Sie Ihr IA32-Assemblerprogramm mit den Werten der obigen Matrizen A und B.

Aufgabe 2: Matrixmultiplikation in C

Nun soll die Implementierung der Matrixmultiplikation in C erfolgen. Dabei sollen zwei beliebige NxN Matrizen multipliziert werden. Die zum Import von Matrizen notwendigen Funktionen sind bereits vorgeben. Sie finden die Vorlage in der Datei *prak3.c*.[1]

Hinweis: Kommentieren Sie Ihren Code ausführlich.

Mit dem mitgelieferten Matrixgenerator (*matrixgenerator.c*) können Sie Eingabematrizen generieren. Der Matrixgenerator schreibt die Matrizen in die Konsole, sodass Sie z. B. mit folgendem Befehl eine Matrix der Größe 3 x 3 erzeugen können und in der Datei *3.txt* speichern können: `./matrixgenerator 3 > 3.txt`. Die Matrizeneinträge sind zufällig erzeugte Ganzzahlen. Verwenden Sie den optionalen Parameter *-mode double*, um Matrizen mit Gleitkommazahlen zu generieren. **Hinweis**: Der erste Wert in der Ausgabe gibt die Zeilen- und Spaltenanzahl der Matrizen an.

a) Schreiben Sie eine Funktion *printMatrix()*, welche die Ergebnismatrix in einem der Matrix-Eingabedatei identischen Format ausgibt. Das Ergebnis Ihrer Berechnung können Sie so auf einfache Weise weiter verarbeiten. Falls eine solche Ausgabe durch ihr Programm durchgeführt wird, können Sie mit folgendem Befehl die Ausgabe in eine Datei umleiten: `./prak3 > file.txt`. Testen Sie Ihre Implementierung mit den obigen Matrizen A und B.

b) Das Programm soll über die Kommandozeile eine Textdatei übergeben bekommen, in der die zu multiplizierenden Matrizen enthalten sind. Als zweiter Parameter wird die *Berechnungsmethode* übergeben, z. B. “c_row”, “c_column” oder “c_double” (vgl. Aufgabenteil c). Bis auf die letzte Variante arbeiten die Berechnungsmethoden mit Ganzzahlen vom Typ .long; die letzte Variante arbeitet mit Gleitkommazahlen doppelter Genauigkeit. Nutzen Sie die Funktion *validateArguments()* um die übergebenen Parameter auf ihre Korrektheit zu überprüfen.

c) Schreiben Sie nun in C zwei Funktionen zur Matrixmultiplikation: 1. *multiplyCbyColumn()*, welche die Matrixelemente spaltenweise adressiert, und 2. *multiplyCbyRow()*, welche die Matrixelemente zeilenweise adressiert. Es empfiehlt sich die Addition und Multiplikation der Werte in eine eigene Funktion auszulagern.

In der Vorlesung wurden Möglichkeiten der Parallelisierung vorgestellt. Als ein einführendes Beispiel wird die parallelisierte Berechnung des folgenden Integrals betrachtet (vgl. Übung 4).

Die Berechnung von π kann auf ein Flächenintegral zurückgeführt werden.

$$\int_0^1 \frac{4}{1+x^2} dx$$

Das C Programm ist im Folgenden zu sehen.

```
1  #include <stdio.h>
2  #include <math.h>
3
4  static long num_steps = 100000;
5  double step;
6
7  main()
8  {
9     int i;
10    double x, pi, sum = 0.0;
11
12    step = 1.0 / (double) num_steps;
13    for (i = 1; i <= num_steps; i++)
14    {
15       x = (i - 0.5) * step;
16       sum = sum + 4.0 / (1.0 + x * x);
17    }
18
19    pi = step * sum;
20    printf("PI %lf\n", pi);
21 }
```

[1] Selbstverständlich besteht kein Zwang, diese Vorlage zu benutzen.

Mittels einiger OpenMP[2] Befehle können Schleifen ausgerollt und parallel bearbeitet werden. Dies funktioniert ohne Änderungen am eigentlichen Berechnungscode.

Eine parallelisierte Implementierung ist im Folgenden zu sehen.

```
1  #include<stdio.h>
2  #include<math.h>
3  #include<omp.h>
4
5  int main (void) {
6   int i;
7   long num_steps = 100000000;
8   double x,pi,step,sum=0.0;
9
10  step = 1.0/(double) num_steps;
11
12  #pragma omp parallel for private(x) reduction(+:sum)
13  for (i=0; i< num_steps; i++) {
14   x = (i-0.5)*step;
15   sum += 4.0/(1.0+x*x);
16   }
17
18   pi = step * sum;
19   printf("PI %lf\n",pi);
20   return 0;
21 }
```

Die Übersetzung erfolgt mittels **gcc -fopenmp program.c**. Die Anzahl der Threads kann im Programm angegeben werden. Alternativ kann auch eine Umgebungsvariable gesetzt werden. Bei zwei Threads sollte sich die Programmlaufzeit halbieren (Kontrolle z. B. über time).

d) Optimieren Sie die Funktionen *multiplyCbyColumn()* und *multiplyCbyRow()* mittels OpenMP. Weisen Sie Ihren Compiler an, die Schleifen zur Matrixmultiplikation in mehrere parallele Threads aufzuteilen.

e) Testen Sie, welche der beiden Traversierungsreihenfolgen schneller arbeitet. Implementieren Sie basierend auf der schnelleren Traversierungsreihenfolge eine Funktion *multiplyCDouble()*, welche auf Gleitkommazahlen doppelter Genauigkeit arbeitet und das Ergebnis in *matResultDouble* ablegt. Ergänzen Sie Ihr Programm um eine Funktion, die das Ergebnis von *multiplyCDouble()* ausgeben kann.

Aufgabe 3: Vergleich der Implementierungen

Neben der Benutzung von OpenMP stellt der GCC Compiler verschiedene Optimierungen zur Verfügung.

Vergleichen Sie die Laufzeit der in C implementierten Funktionen:

- *multiplyCbyColumn()* und *multiplyCbyRow()*
- *multiplyCbyColumn()* und *multiplyCbyRow()* mit OpenMP
- *multiplyCDouble()*

Verwenden Sie für alle Tests eine Matrix mit 2000 x 2000 Elementen. Eine entsprechende Matrixdatei können Sie sich mit dem mitgelieferten Generator erstellen. Zur Zeitmessung können Sie unter Unix den Befehl *time* verwenden, notieren Sie dann sowohl die Benutzerzeit als auch die reelle Zeit. Stellen Sie die Ergebnisse in einer Tabelle dar.

Für alle Implementierungen sollen Tests

- ohne Optimierung durch den Compiler,
- mit -O1,
- mit -O2

[2] http://openmp.org/wp/

durchgeführt werden.

Hinweis zu OpenMP: Führen Sie Durchläufe für einen Thread, zwei Threads und vier Threads durch.

Vergleichen Sie die Laufzeiten der unterschiedlichen Traversierungsstrategien (*multiplyCbyColumn()* und *multiplyCbyRow()*), sowie Unterschiede zwischen der mittels OpenMP optimierten Variante. Was fällt Ihnen auf? Womit können Sie dies begründen? Was ergibt der Vergleich mit der Funktion *multiplyCDouble()*? Geben Sie den Speedup der OpenMP Implementierungen an.

Reduzieren Sie den Befehlssatz Ihres Compilers auf den i386 Befehlssatz und bestimmen Sie die Laufzeit ohne Compiler-Optimierungen, sowie mit -O1 und OpenMP-Optimierung. Die Flags zur Reduzierung des Befehlssatzes lauten: `-m32 -mtune=i386`. Vergleichen und begründen Sie Ihre Resultate.

Ermitteln Sie den Prozessor in Ihrem Rechner (bzw. RBG) und recherchieren Sie seine FLOPS[3]-Zahl. Überschlagen Sie die Anzahl der Gleitkommaarithmetik-Operationen der optimierten C-Implementierung und berechnen Sie die erreichten FLOPS. Wie vergleichen sich die beiden Resultate?

Folgende Dateien werden von Ihnen als Abgabe via SVN erwartet:

- matrixmultiplikation.pdf mit Ihrer Lösung der 1. Teilaufgabe der 1. Aufgabe.
- matrixmultiplikation.asm mit Ihrer Lösung der Programmieraufgabe der 1. Aufgabe.
- prak3.c mit Ihrer Lösung der 2. Aufgabe.
- auswertung.pdf mit Ihrer Auswertung.

[3] floating point operations per second