Übungen zur Vorlesung

Paralleles Höchstleistungsrechnen

Dr. S. Lang, D. Popović

Abgabe: 27. Oktober 2009 in der Übung

Übung 1 Instruction Level Parallelism (10 Punkte)

Instruction Level Parallelism (ILP) ist ein Maß, wieviele Operationen eines Programms parallel bearbeitet werden können. In dieser Aufgabe untersuchen wir an einem Beispielprogramm, wie verschiedene Daten- und Kontrollfluß-Abhängigkeiten den ILP beeinflussen können. Wir betrachten folgende Abhängigkeiten:

- *Data true dependence, DTD*:
  Instruktionen hängen von Resultaten vorheriger Instruktionen ab,
- *Data antidependence, DA*:
  Instruktionen schreiben in Adressen, die von einer anderen Instruktion gelesen werden,
- *Data output dependence, DOD*:
  Instruktionen schreiben in Adressen, die von einer anderen Instruktion beschrieben werden,
- *Control dependence, CD*:
  Die Ausführung einer Instruktion hängt von Kontrollbedingungen (`if, ...`) ab.

Wir untersuchen die Abhängigkeiten für eine unvollständige Implementierung einer Hash-Tabelle. Mit Hash-Tabellen können Fragen wie „*Existiert ein Element in einer gegebenen Menge?*“ beantwortet werden. Dazu speichert die Hash-Tabelle etwa verkettete, geordnete Listen von Elementen. Die Listen werden über Schlüssel, die sogenannte *Hash-Funktion*, zugänglich. Um nun zu testen, ob ein Element in der Menge vorhanden ist, wird dessen Schlüssel, der *Hash-Wert*, errechnet, und nur in der passenden Liste (generell nennt man die speichernde Datenstruktur *Bucket*) gesucht. Unsere Hash-Tabelle kann 1024 Buckets mit je einer verketten Liste von Elementen speichern. In folgendem Listing wird gezeigt, wie die Hash-Tabelle initialisiert wird:

```
1   /*************************************************************/
2   /* an incomplete implementation of a hash table              */
3   /*************************************************************/
4
5   /* struct for elements to be inserted */
6   typedef struct Element {
7     int value;
8     struct Element *next;
9   }
10
11  Element myElements[N_ELEMENTS]; /* array initialized with items */
12  Element *bucket[1024];          /* to be inserted, pointers in  */
13                                  /* buckets initialized to NULL  */
14
15  for (i=0; i<N_ELEMENTS; i++)
16  {
17    Element *ptrCurr, **ptrUpdate;
18    int hashIndex;
19
20    /* find location where the new element is to be inserted */
21    hashIndex = myElements[i].value & 1023;
22    ptrUpdate = &bucket[hashIndex];
23    ptrCurr   = bucket[hashIndex];
24
```

```
25      /* find place in chain to insert the new element */
26      while ( ptrCurr && ptrCurr->value =< myElements[i].value)
27      {
28        ptrUpdate = &ptrCurr->next;
29        ptrCurr   = ptrCurr->next;
30      }
31
32      /* update pointers to insert the new element into chain */
33      myElements[i].next = *ptrUpdate;
34      *ptrUpdate = &myElements[i];
35    }
```

Dabei können die Elemente ein Integer speichern. Die einzufügenden Elemente (`N_ELEMENTS` Stück) sind im Feld `myElements` gespeichert, die Hash-Tabelle `bucket` kann 1024 Zeiger auf verkettete Element-Listen speichern, alle Zeiger zeigen zu Beginn ins Nichts.

Nun wird über die vorhandenen Elemente iteriert und in Zeile 21 der Hash-Wert des Elements berechnet. Die Hashing-Funktion ist recht einfach, sie besteht aus den zehn letzten Bits des Wertes eines Elements, denn der Operator `&` verknüpft den Wert des Elements mit dem Bitmuster 11 1111 1111 der Dezimalzahl 1023 mit logischem `AND`. Anschliessend wird über die verkettete Liste iteriert. Mit dem Zeiger `ptrCurr` wird die Liste traversiert, die Variable `ptrUpdate` merkt sich den Pointer, der angepasst werden muss, um ein Element an eine Stelle in der Liste einzufügen. Dazu wird in Zeile 23 `ptrCurr` auf das erste Element des entsprechenden buckets gesetzt, und in der anschliessenden while-Schleife (Z. 25-30) solange iteriert, bis die passende Einfüge-Stelle gefunden ist. Danach wird in den Zeilen 33, 34 das neue Element eingefügt, indem der `next`-Zeiger des einzufügenden Elements auf die Adresse des nachfolgenden Elements, die in `ptrUpdate` steht, gesetzt wird. In Zeile 34 wird dann der Zeiger aufs Nachfolge-Element auf das aktuelle Element gesetzt (`ptrCurr` ist ein Doppelzeiger, d.h. der Operator `*` dereferenziert einmal und zeigt somit auf einen Zeiger).

Wir interpretieren nun jede Zeile als eine Maschinen-Instruktion und untersuchen die Abhängigkeiten. Für sie erhalten wir einen dynamischen Abhängigkeitsgraphen ähnlich zu jenem aus Abbildung 0.1 (es sind nicht alle Abhängigkeiten eingezeichnet, und es fehlen die Instruktionen aus Z. 28, 29). Jeder Knoten des Graphen repräsentiert eine Maschineninstruktion, die innerhalb eines Zyklus bearbeitet wird. Jede Horizontale enthält also Instruktionen (Zeilen), die parallel bearbeitet werden können. Pfeile zwischen den Knoten beschreiben Abhängigkeiten. So besteht z.B. zwischen den Zeilen 15 und 21 eine DTD, da `i` in Zeile 15 berechnet und in Zeile 21 verwendet wird.

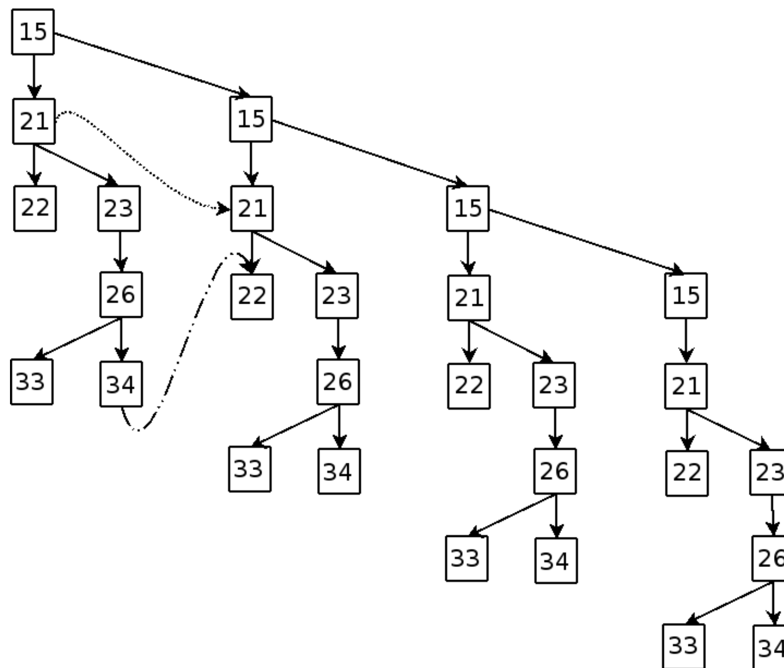

Abbildung 0.1: Dynamische Abhängigkeiten der Hash-Tabelle

Teilaufgabe (a) **(5 Punkte)**

(a) Was für eine Abhängigkeit besteht zwischen den Zeilen 21 und 22/23?

Für verschiedene Elemente könnte derselbe Hash-Wert berechnet werden, was zu Abhängigkeiten zwischen den Schleifendurchläufen führt.

(b) Was für eine Abhängigkeit besteht nun zwischen den Zeilen 21 und der gleichen Zeile 21 für zwei verschiedene Elemente mit gleichem Hash-Wert?
(c) Was für eine Abhängigkeit besteht zwischen den Zeilen 34 und 22?

Teilaufgabe (b) **(5 Punkte)**

Wir gehen nun von einem stark vereinfachten Szenario aus, in dem die Hash-Tabelle initial leer ist und die Zahlen $0 \dots 1023$ einzufügen sind, d. h. pro Bucket gibt es nur ein Element, Wir betrachten die Abhängigkeiten bei Einfügen der ersten vier Elemente.

(a) Welche ist die einzige Abhängigkeit, die nun noch Zwischen den Zeilen besteht (betrachten Sie die Variable `i` in Zeile 15).
(b) Schreiben Sie die `for`-Schleife so um, dass die Abhängigkeit zwischen den Schleifendurchläufen vermindert wird (Tipp: In einer Schleife Instruktionen für `i` und `i+1` implementieren).
(c) Laut Graph besteht eine Iteration der äusseren Schleife aus 7 Instruktionen, insgesamt bei 1024 Iterationen also 7168 Instruktionen. Diese Instruktionen sind in 4 Zyklen abgearbeitet, die Schleife braucht also $4 + 1024 = 1028$ Zyklen, bis sie ganz abgearbeitet ist. Dies ergibt einen ILP (Available Instruction Level Parallelism) von $7128/1028 = 6.97$. Welchen Einfluss hat Ihre Optimierung aus (b) auf den ILP?

Übung 2 Messen von MFLOPS (10 Punkte)

In dieser Übung wollen wir für verschiedene numerische Anwendungen ausprobieren, wieviele Rechenoperationen pro Sekunde heute möglich sind. Dazu wollen wir folgende mathematische Operationen implementieren:

1. *Matrixmultiplikation.*
   Es seien zwei Matrizen $A, B \in \mathbb{R}^{n \times n}$ gegeben. Dann ist das Matrixprodukt $C = AB$ wiederum eine Matrix $C \in \mathbb{R}^{n \times n}$, deren Einträge c_{ij} definiert sind durch:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}.$$

2. *Gauß-Seidel 2d.*
   Es sei ein Gebiet in $d = 2$ Dimensionen definiert über

$$\Omega_n^d = \Big\{ (i_0, \dots, i_{d-1}) \in \mathbb{Z}^d \ \mid \forall\ 0 \le k < d : 0 \le i_k < n \Big\}.$$

   In $2d$ wäre dies zum Beispiel ein Netz mit n^2 Punkten. Wir wollen ein Gitter mit äquidistanten Punkten wählen, d. h. $\Omega = [0, n-1]^2$. Auf diesem Gitter sei eine Gitterfunktionen $u^m : \Omega_n^2 \to \mathbb{R}$ gegeben. Für diese definiert die Iterationsvorschrift

$$u^{m+1}(i,j) = \frac{1}{4} \Big\{ u^{m+1}(i-1,j) + u^{m+1}(i,j-1) + u^m(i,j+1) + u^m(i+1,j) \Big\} \quad (i,j) \in [0, n-1]^2$$

   die sogenannte Gauss-Seidel-Iteration.

Teilaufgabe (a) (5 Punkte)

Implementieren Sie die Matrixmultiplikation in der Programmiersprache C/C++ und verwenden Sie für die Matrizen eine beliebige Datenstruktur ihrer Wahl (z. B. mehrdimensionale Felder). Bestimmen Sie die Anzahl der Fließkommaoperationen und ermitteln Sie die Geschwindigkeit des Programmes in „Millionen Fließkommaoperationen pro Sekunde".

Zur Zeitmessung können Sie die Funktionen, welche in `timer.h` zur Verfügung gestellt werden, verwenden. Den Header finden Sie auf der Vorlesungs-Homepage. Hinweise zur Verwendung finden Sie auf der letzten Seite. Führen Sie die Zeitmessung bei kleiner Problemgröße n jede Operation mehrmals hintereinander in einer Schleife ausführen, um Zeitmessfehler zu verringern. Initialisieren Sie die die Felder mit sinnvollen Daten (nicht 0.0), z. B. $u(i,j) = i + j$. Übersetzen Sie das Programm mit maximaler Optimierungsstufe. Für den `GNU C/C++` Compiler ist etwa `-O3 -funroll-loops` empfehlenswert.

Stellen Sie die Ergebnisse in graphischer Form, d. h. MFLOPS über Problemgröße dar. Diskutieren Sie die Form der Kurven, insbesondere warum die MFLOP-Rate wann absinkt.

Teilaufgabe (b) (5 Punkte)

Wiederholen Sie die Untersuchungen aus Teilaufgabe (b) für das Gauss-Seidel-Verfahren.

Zusatzpunkte (4 Punkte)

2 ZP Verbessern Sie die Cachenutzung durch Kacheln und ermitteln Sie die Beschleunigung.

2 ZP Verwenden Sie bei der Gauß-Seidel-Iteration andere Durchlaufreihenfolgen, z. B. das Schachbrettmuster: Bearbeite zuerst alle Indizes mit $i + j$ gerade, dann die mit $i + j$ ungerade.

Hinweise zur Zeitmessung

Die verschiedenen Zeiten

Bei der Zeitmessung am Computer ergibt sich das Problem, dass die Zeit, die ein Programm benötigt, von der Auslastung des Systems abhängt. Sind viele Prozesse tätig, bekommt der einzelne nur wenig Zeit und läuft dementsprechend lange.

Was konstant bleibt, ist die sogenannt Prozessorzeit, die angibt, wieviele Prozessorsekunden das Programm verbraucht hat. Die Uhr tickt also weiter, solange das Pogramm läuft, wenn das Betriebssystem das Programm warten läßt, steht die Uhr.

timer.h

Um Ihnen die Arbeit zu erleichtern, haben wir für Sie eine Headerdatei `timer.h` mit Hilfsbefehlen, um die verbrauchte Prozessorzeit auszulesen, vorbereitet. Sie stellt Ihnen drei Befehle zur Verfügung:

- `void reset_timer(struct timeval* timer)`: Zähler zurücksetzen/initialisieren.
- `double get_timer(struct timeval timer)`: verbrauchte Sekunden auslesen.
- `void print_timer(struct timeval timer)`: verbrauchte Sekunden ausgeben.

Beispiel

```
#include "timer.h"       // Headerfile zur Zeimessung

int main()
{
  struct timeval timer; // Variable zur Zeitmessung
  reset_timer(&timer);  // Zaehler zuruecksetzen/initialisieren
  ...                   // Was tun und Zeit verbrauchen
  print_timer(timer);   // Zaehler ausgeben
}
```