Übungen zur Vorlesung

Paralleles Höchstleistungsrechnen

Dr. S. Lang, J. Pods

Abgabe: 6. November 2012, 12 Uhr, per E-mail an `jurgis.pods@iwr.uni-heidelberg.de`

Übung 1 Effiziente Cache-Nutzung (5 Punkte)

Im folgenden Programmsegment sei u ein Feld der Dimension $n \times n$:

```
1 for (int k=0; k<1000; ++k)
2   for (int i=1; i<n-1; ++i)
3     for (int j=1; j<n-1; ++j)
4       u[i][j] = 0.25 * (u[i-1][j] + u[i+1][j] + u[i][j-1] + u[i][j+1]);
```

Das Feld u sei sinnvoll initialisert, im inneren des Feldes werden also die gemittelten Werte aus den Nachbarn eingetragen. Beachten Sie, daß alte (noch nicht besuchte) und neue (schon besuchte) Einträge in die Mittelung eingehen. Im verwendeten Rechner habe eine Cacheline die Größe $l = 32$ Byte, und eine Fließkomma-Zahl belege 8 Byte. In eine Cache-Line passen also 4 Zahlen.

1. Kann man durch Umordnen der Schleifen eine bessere Cache-Nutzung erreichen?
2. Schreiben Sie den Algorithmus so um, dass eine Kachelung der Daten verwendet wird, und diskutieren Sie die Auswirkungen auf die Cache-Nutzung.

Übung 2 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, 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 z.B. verkettete, geordnete Listen von Elementen. Die Listen werden über Schlüssel, berechnet durch die *Hash-Funktion*, zugänglich. Um zu testen, ob ein Element in der Menge vorhanden ist, wird dessen Schlüssel, der *Hash-Wert*, errechnet, und nur in der passenden Liste (die speichernde Datenstruktur nennt man *Bucket*) gesucht. Unsere Hash-Tabelle kann 1024 Buckets mit je einer verketten Liste von Elementen speichern. Das folgende Listing zeigt, 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 of items */
12  Element *bucket[1024];          /* 1024 buckets, pointers in each */
13                                  /* bucket are 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  }
```

Die Elemente speichern jeweils ein Integer, die `N_ELEMENTS` einzufügenden Elemente 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 durch eine einfache Hashing-Funktion berechnet: Der Hash-Wert 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 idem in 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.

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

(a1) 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.

(a2) Was für eine Abhängigkeit besteht nun zwischen den Zeilen 21 und der gleichen Zeile 21 für zwei verschiedene Elemente mit gleichem Hash-Wert?

(a3) 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 $0 \ldots 1023$ einzufügen sind, d. h. pro Bucket gibt es nur ein Element, Wir betrachten

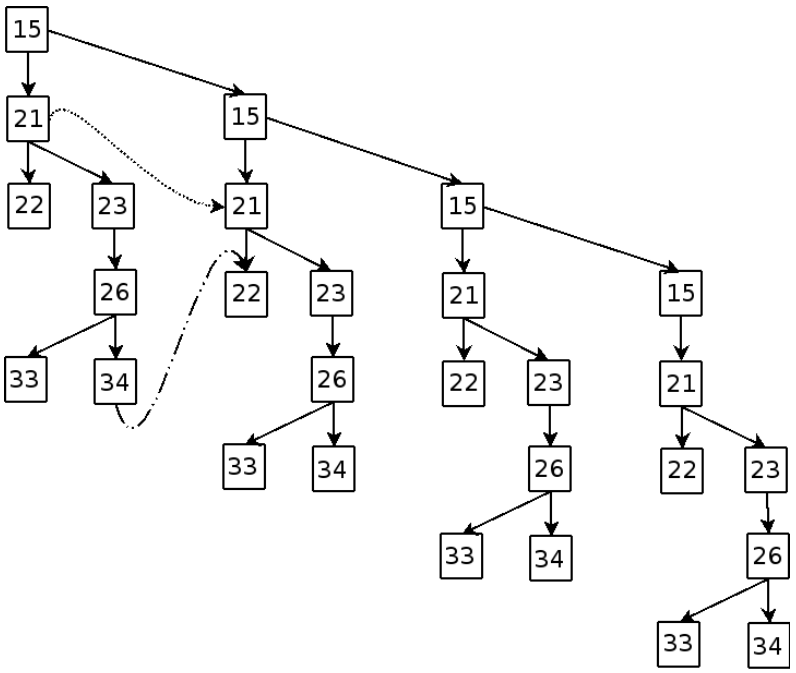

Abbildung 0.1: Dynamische Abhängigkeiten der Hash-Tabelle. Jeder Knoten entspricht einer Programm-Zeile, die Pfeile geben verschiedene Abhängigkeiten an. Gezeigt wird hier der ideale Fall (Teilaufgabe b), in dem die `while`-Schleife nicht betreten wird (Instruktionen 28, 29 fehlen daher). Die gestrichelten Pfeile zeigen mögliche Abhängigkeiten bei Teilaufgabe (a).

die Abhängigkeiten bei Einfügen der ersten vier Elemente.

(b1) Welche ist die einzige Abhängigkeit, die nun noch Zwischen den Zeilen besteht (betrachten Sie die Variable `i` in Zeile 15).

(b2) 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).

(b3) 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?