Seminerausarbeitung
zur Lehrveranstaltung 185.272
“Grundlagen methodischen Arbeitens“
im WS 2006/07

Automatic Loop Interchange

bearbeiten von

Ahmet Hulusi AKAN
Matrikelnummer: 0325157 Studienkennzahl: 534

Technische Universität Wien
Fakultät für Informatik
Institut für Computersprachen
Arbeitsbereich Programmiersprachen und Übersetzer

Lehrveranstaltungsleiter: A.o. Univ. Prof. Dr. Anton Ertl

Eingereicht am 05.12.2007

Erklärung

Hiermit erkläre ich, dass die vorliegende Arbeit selbständig, unter Angabe aller Zitate und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe, und ich diese Arbeit zuvor keiner anderen Stelle oder Institution als Studiums- oder Prüfungsleistung vorgelegt habe.

Wien den / 05.01.2007

Kurzzusammenfassung

Dieser Ausarbeitung liegt der Artikel über die *Automatischer Schleifenaustausch* Randy Allen und Ken Kennedy zugrunde. Die Hauptaussagen dieses Artikels sind Methoden, um Schleifen auszutauschen, hängen von ihrem Aufbau und insbesondere der Abhängigkeiten zwischen den Anweisungen ab. Wenn es möglich ist, Schleifen zu vertauschen, kann dies für bestimmte Plattformen Performancegewinne bringen.

Zwei Schleifenköpfe dürfen in einer vorzüglichen Schleifenverschachtelung verändert werden, wenn die daraus resultierende Veränderung in der Ausführungsreichenfolge der Anweisungen im gemeinsamen Rumpf nicht der Richtung der urwüchsigen Datenabhängigkeiten entgegensteht.

Einleitung

In dieser Arbeit geben wir einen kurzen Überblick über grundlegende Definition und Theoreme, Schleifenvertauschung und Skalarexpansionen. Dies geschieht anhand eines im Rahmen der „84 Symposium on Compiler Construction“ veröffentlichen Dokument von John R. Allen und Ken Kennedy „*Automatischer Schleifenaustausch*“. Diese Arbeit diskutiert bestimmter Anwendungen von dieser Theorie der Schleifeaustausch.

Ausgangpunkt für das von R. Allen und K. Kennedy in [All1] vorgeschlagene Analyseverfahren zur Bestimmung wertegleicher Ausdrücke in Programmen war die Beobachtung, dass es eine einfache und effektive Methode gibt die Zulässigkeit einer Schleifenvertauschung und Parallelisierung zu überprüfen.

Eine automatische Parallelisierung ist im Allgemeinen nicht tauglich um die Leistungsfähigkeit einer parallelen Architektur ganz oder umfangreich auszunutzen.

In der 1970er Jahren wurde mit den *Vektorrechnern*[1] ein erster Schritt in Richtung Parallelverarbeitung getan. Sie arbeiten nach dem *SIMD* (Single Instruktion, Multiple Data) – Prinzip. Da diese Rechner Vektoroperationen (*datenparallele Operationen*) sehr schnell ausführen können, sind sie skalaren Rechnern diesbezüglich signifikant überlegen. Trotzdem sind sie technologischen Beschränkungen unterworfen: Sie sind nicht *skalierbar*, d.h. die Anzahl der Pipeline-Stufen kann nicht beliebig gesteigert werden; sie bieten oft nicht genug Flexibilität, und sie können nicht effektiv eingesetzt werden für unregelmäßige oder nicht – datenparallele Programme.

---

[1] Ein Vektorrechner enthält eine oder mehrere arithmetische Verarbeitungseinheiten, die in mehrere Aufeinanderfolgende Teilschritte (Stufen) aufgeteilt wurden, sodass in allen Stufen gleichzeitig, allerdings auf verschiedenen Daten, gerechnet werden kann („*pipelining*“). Dadurch ist ein Vektorrechner für die Verarbeitung von Sequenzen gleichartiger Operationen (Vektoroperationen) auf Operandenfeldern (Vektoren) geeignet. Für skalare Rechnungen ergibt sich kein Geschwindigkeitsgewinn. Beispiele für Vektorrechner sind Cray 1, Fujitsu VP-100 oder die Vektorknoten der Thinking Maschines CM-5[Keß 1].

Abhängigkeit

Man klassifiziert Abhängigkeiten in Datenabhängigkeiten und Steuerungsabhängigkeiten. Im Folgenden werden nur Datenabhängigkeiten betrachtet.

Eine Datenabhängigkeit für zwei Anweisungen S1 und S2 liegt vor, wenn und nur wenn beide die Anweisungen gilt, dass sie gleiche Speicherzelle benutzen und wenigsten ein Zugriff schreibender Natur ist. Man klassifiziert die Abhängigkeiten nach Art des Speicherzugriffs in echte Abhängigkeiten, Antiabhängigkeiten und Ausgabeabhängigkeiten.

Die Arten von Abhängigkeiten

- echte Abhängigkeit RAW (Read after Write): eine echte Abhängigkeit (true dependence) von S1 nach S2, zeigt mit S1 δ S2.

```
S1      …=X
S2      X=…
```

notation S1 δ S2

- Antiabhängigkeit WAR (Write after Read): eine anti Abhängigkeit (anti dependence) von S1 nach S2, zeigt mit δ^{-1}.

```
S1      X=…
S2      …=X
```

notation S1 δ^{-1} S2

- Ausgabeabhängigkeit WAW (Write after Write) : (output dependence)

```
S1      X=
S2      X=
```

notation: S1 δ^{o} S2

Erweiterung des Abhängigkeitsbegriffs

- Bei folgendem Beispiel ist die Aussage „S hängt von sich selbst ab“ nicht ausreichend, da die genaue Form der Abhängigkeit nicht erkennbar ist.

```
DO      I = 1, N
        A(I + 1)= A(I)+B(I)
ENDDO
```

- Für folgendes Beispiel gilt ebenfalls „S hängt von sich selbst ab“.

```
DO      I = 1, N
        A(I + 2) = A(I) + B(I)
ENDDO
```

Der Iterationsvektor gibt an, welche Anweisung S innerhalb des Schleifenrumpfes gemeint ist. So ist *S* mit dem Iteration *I (2,1)* die Anweisung, welche bei der zweiten Iteration der

äußeren Schleife und der ersten Iteration der zweiten Schleife ausgeführt wird. Alle Iterationsvektoren bilden einen Iterationsraum.

Distanzvektor - Richtungsvektor

Wenn es eine Abhängigkeit zwischen S1 bei der Iteration i und S2 bei der Iteration j gibt, dann ist der Distanzvektor *d[i ,j]* definiert als eine Längenvektor, so dass

$$d[i,j]_k = j_k - i_k$$

Aus dieser Definition folgt, dass bei einer Schachtelung der Schleifen, so dass $i < j$, folgt $d[i,j]>0$

Der Richtungsvektor ist wie folgt definiert:

$$D(i,j)_k = \begin{cases} "<" & \text{, wenn} \quad d(i,j)_k > 0 \\ "=" & \text{, wenn} \quad d(i,j)_k = 0 \\ ">" & \text{, wenn} \quad d(i,j)_k < 0 \end{cases}$$

Die Zeichen" $<$ " und " $>$ " kann man sich wie Pfeile vorstellen. Die „Pfeilspitze“ zeigt auf die Iteration, die zuerst ausgeführt werden muss.

Schleifenvertauschung

Die Schleifenvertauschung ist eine Transformation, die bei der Vektorisierung von geschachtelten Schleifen angewandt wird. Es handelt sich hierbei schlisslich um eine Umordnung von Anweisungen. Diese Schleife enthält eine echte Abhängigkeit.

```
DO I = 1, N
        DO J = 1 , M
S                A( I , J +1 ) = A( I , J ) + B
        ENDDO
ENDDO
```

```
DO J = 1 , M
        DO I = 1 , N
S                A( I , J +1 ) = A( I , J ) + B
        ENDDO
ENDDO
```

```
DO I = 1 , M
S       A( 1 :N, J +1 ) = A( 1 :N, J ) + B
ENDDO
```

Die Schleife liest sich Vektorisierung, weil die Abhängigkeit nach Schleifenvertauschung von der äußeren Schleife getragen wird, und die innere Schleife abhängigkeitsfrei ist.
Wenn man nun S(I,J) als Instanz der Anweisung S auffasst, ergibt sich folgende Darstellung

```
DO J = 1 , M
        DO I = 1 , N
                S
        ENDDO
ENDDO
```

Man kann sehen, dass S(1,2) in diesem Beispiel nach S(2,1) ausgeführt wird. Bei einer Schleifenvertauschung würde S(1,2) jedoch vor S(2,1) ausgeführt. So betrachtet, kann man sich die Schleifenvertauschung tatsächlich als Umordnung von Anweisungen vorstellen. Deshalb kann man mit Hilfe der Datenabhängigkeitsbetrachtung zeigen, ob und wann eine Schleifenvertauschung erlaubt ist.

Sicherheit der Schleifenvertauschung

Bei dem folgendem Beispiel :

```
DO J = 1 , M
        DO I = 1 , N
                A( I , J +1 ) = A( I +1, J ) + B
        ENDDO
ENDDO
```

Ist die Scheifenvertauschung nicht durchführbar, weil sie Abhängigkeiten verletzen würde.

In diesem Beispiel der Wert *A(2,2)* bei der Iteration *(2,1)* zugewiesen. Bei einer Vertauschung würde *A(2,2)* bei der Iteration *(1,2)*, also würde bei der Zuweisung der falsche Wert zur Berechnung verwendet.

Es gibt zwei Klassen von Abhängigkeiten, die für die Sicherheit der Schleifen wichtig sind.

Vertauschungsverhindernde Abhängigkeit ist gegeben, wenn eine die Vertauschung der Schleifen die Endpunkte einer Abhängigkeit vertauscht.

Vertauschungssensitive Abhängigkeit liegt dann vor, wenn die Abhängigkeit von derselben Schleife getragen wird. Das heißt, dass *die vertauschungssensitive Abhängigkeit* sich mit ihrer Trägerschleife auf den neuen Level bewegt.

Eine geschachtelte Schleife mit dem Richtungsvektor *(<,>)* ist zum Beispiel vertauschungsverhindernd, weil der Richtungsvektor nach der Vertauschung die folgende Gestalt hat: *(>,<)*. Man sieht deutlich, dass die Endpunkte der Abhängigkeit vertauscht würden. Ist *D(i,j)* ein Richtungsvektor für eine geschachtelte Schleife, so erhält man den Richtungsvektor nach der Permutation der Schleifen, indem man die Elemente von *D(i,j)* auf die gleiche Weise permutiert.

Eine Richtungsmatrix für eine verschachtelte Schleife ist eine Matrix, in welcher jede Zeile ein Richtungsvektor enthält. Die Vektoren repräsentieren Abhängigkeiten zwischen Anweisungen innerhalb der Schachtelung. Identische Richtungsvektoren werden in einer einzigen Zeile zusammengefasst.

Eine Permutation der Schleifen in einer Verschachtelung ist legal, wenn und nur wenn nach der Permutation der Matrixspalten kein „>“ als das am weitesten links stehende nicht „=“ Symbol auftritt.

Beispiel

Für folgenden Quelltext

```
DO I = 1 , N
        DO J = 1 , M
                DO K=1 , L
                        A( I +1, J -1,K) = A( I , J ,K) + A( I , J+1,K+1)
                ENDDO
        ENDDO
ENDDO
```

Ergibt sich die Richtungsmatrix

$$\begin{bmatrix} < & < & = \\ < & = & > \end{bmatrix}$$

Wird nun die äußerste Schleife in die innerste Position „verschoben“, ergibt sich durch Anwendung der gleichen Permutation die folgende Matrix

$$\begin{bmatrix} < & = & < \\ = & > & < \end{bmatrix}$$

Daraus folgt, dass diese Transformation illegal ist.

Wahl der besten Schleifenanordnung

Diese Frage kann nur unter Kenntnis der Zielplattform beantwortet werden.
Für das folgende Beispiel

```
DO I = 1 , N
        DO J = 1 , M
                DO K = J , L
                        A( I +1, J +1,K) = A( I , J ,K) + B
                ENDDO
        ENDDO
ENDDO
```

Ist z.B. folgende Vektorisierung möglich

```
DO I = 1 , N
        A( I +1,2:M+1,1: L ) = A( I , 1 :M, 1 : L ) + B
ENDDO
```

Für *SIMD* Maschinen mit vielen synchron arbeitenden Einheiten ist eine solche Optimierung vielleicht sinnvoll. Vektormaschinen können aber oft nur einen Schleifendurchlauf parallelisieren, so dass diese Optimierung für diese Maschinen nicht geeignet ist.

Schleifenvertauschung und Vektorisierung

Eine Schleife die keine Abhängigkeiten in sich trägt, kann folglich auch keine Abhängigkeit tragen, welche ihre Vertauschung verhindern würde. Diese Tatsache impliziert, dass das

Verschieben der Schleife nach innen immer legal ist. Dieser Prozess kann ohne Probleme fortgesetzt werden, bis diese Schleife auf der untersten Verschachtelungstiefe ist.

Durch diesen Prozess erhalten wir also eine abhängigkeitsfreie Schleife auf der untersten Verschachtelungstiefe. Dies ist für die Vektorisierung von großem Vorteil, da eine abhängigkeitsfreie Schleife immer vektorisierbar ist. Deshalb wird bei der Schleifenvertauschung eine solche Strategie verfolgt.

Algorithmus : select_loop_and_interchange

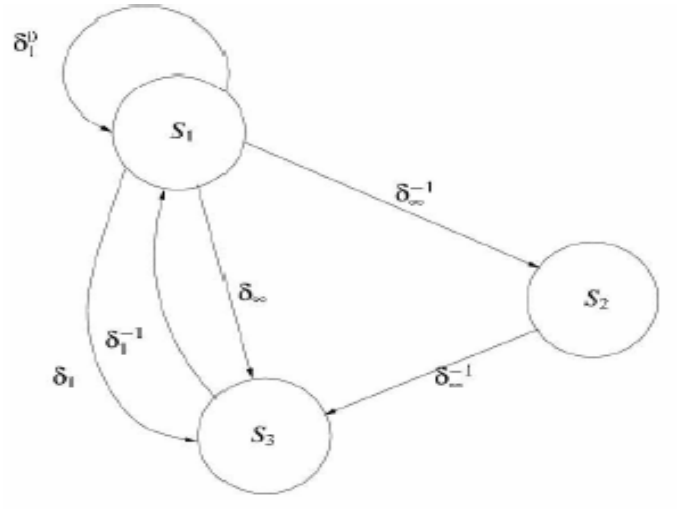
```
procedure select_loop_and_interchange(R,k)
    // k is the current nesting level in region R
    // R is strongly connected when only edges at level k and deeper are considered.
    let N be the deepes loop nesting level;
    let p be the level of outmost carried dependence;
    if p = k then begin
        not_found = true;
        while (not_found and p ≤ N) do
            if the level-p loop can be safely shifted outward level k and there exists a dependence d
            carried by the loop such that the direction vector for d has "=" in every position but p
            then not_found = false;
            else p := p + 1;
        end
        if p > N then p = k;
    end
    if p > k then shift loops at level k, k+1, ..., p-1 inside the level-p loop;
```

Skalarenexpansion

Das folgende Beispiel zeigt ein typisches Stück Code für das Umkopieren von zwei Vektoren

```
        DO I = 1 , N
S1          T = A( I )
S2          A( I ) = B( I )
S3          B( I ) = T
        ENDDO
```

Der zugehörige Abhängigkeitsgraph



Beispiel

Das Programmfragment kann nicht ohne weiteres vektorisiert werden. Ersetzt man jedoch, die Skalare *T* durch eine temporäre Arrayvariable *T$* so ist eine Vektorisierung möglich

```
  DO I = 1 , N
S1          T$ ( I ) = A( I )
S2          A( I )  = B( I )
S3          B( I )  = T$ ( I )
  ENDDO
T=T$ (N)
```

Aus diesem erweiterten Quelltext lassen sich folgende Anweisungen generieren

```
S1          T$ ( 1 : N) = A( 1: N)
S2          A( 1 : N)  = B( 1: N)
S3          B( 1 : N)  = T$ ( 1: N)
T  =  T$ (N)
```

Dieses Verfahren ist mit hohen Kosten in Form von zusätzlichem Speicherbedarf verbunden.

Die Skalarenexpansion kann immer durchgeführt werden. Die Expansion ist aber nicht immer sinnvoll, weil sie nicht immer zu einem erhöhten Parallelitätsgrad führt. Eine Expansion ist nur dann sinnvoll, wenn die Skalare für eine gemeinsam genutzte Speicherzelle steht, und nicht für ein gemeinsam benutztes Datum. Es ist zu beachten, dass die Skalare ein umschlissende Definition (cover definition) hat. Wenn dies der Fall ist, könnte die Variable, z.B. nicht initialisiert sein.

Zusammenfassung und Ausblick

Mit dem in [All2] vorgestellten neuartigen Verfahren zur Bestimmung wertegleicher Ausdrücke in Programmen haben Allen und Kennedy vollautomatische Schleifenvertausch maßgeblich und beeinflusst und vorangetrieben. Aus diesen Betrachtungen folgt, dass es eine

einfache und effektive Methode gibt die Zulässigkeit einer Schleifenvertauschung zu überprüfen.

Man konstruiere die Richtungsvektoren für die Abhängigkeiten in den Schleifen und trage sie in eine Richtungsmatrix ein. Dann führt man die gewünschten Permutationen auf der Matrix durch und kann nun einfach die Gültigkeit der Umformung erkennen.

Auch wenn in den letzten Jahren mehrere interessante Ansätze zum Problem der automatischen Datenaufteilung erscheinen sind, ist doch die halbautomatische Parallelisierung noch immer der aktuelle Stand der Technik. Vollautomatische Parallelisierung erscheint noch als Utopie, und manche glauben sogar, dass sie es auch für immer bleiben werde.

Literatur- und Quellenverzeichnis

[All1] J. R. Allen, und K. Kennedy :Automatic Loop Interchange, Proceedings of the ACM SIGPLAN ´84 Symposium un Compiler Construction, *SIGPLAN Notice Vol. 10, No. 6 June 1984*

[All2] J. R. Allen, und K. Kennedy . *Optimizing Compilers for Modern Architecturs*. ISBN : 1-55860-286-0, Morgan Kaufmann , 26 September 2001

[Keß1] W. Ch. Kessler. *Automatische Parallelisierung* : Universität Trier, FB 4 – Informatik , SS 1995