Automatische Parallelisierung

Seminar: Sprachen für Parallelprogrammierung

Julian Oppermann

Die Entwicklung von effizienten und korrekten parallelen Programmen ist schwierig. Automatische Parallelisierung bedeutet, dass die für die Nutzung paralleler Rechner erforderlichen Programmtransformationen im Compiler durchgeführt werden. Dabei wird die Reihenfolge von Schleifeniterationen unter Berücksichtigung der bestehenden Datenabhängigkeiten umgeordnet. Die Ermittlung präziser Abhängigkeitsinformationen ist das zentrale Problem der automatischen Parallelisierung. Voraussetzung für die Abhängigkeitsanalyse ist eine statische Vorhersage der Speicherzugriffe. Für die Analyse affiner Zugriffe auf statische Arrays existieren praxistaugliche Verfahren. Dynamische Datenstrukturen auf dem Heap, so wie sie objektorientierte Programme oft verwenden, sind problematisch für die Analyse und ein aktuelles Forschungsthema.

1 Einleitung

Spätestens seit die Prozessorhersteller die Taktfrequenzen und damit die sequentielle Rechenleistung ihrer Prozessoren nicht weiter steigern konnten und Mehrkernprozessoren den Einzug in den Massenmarkt erhalten haben, ist die Verbesserung der parallelen Ausführung von Programmen ein drängendes Thema.

Ein Ansatz ist, die Parallelität explizit vom Programmierer ausweisen zu lassen. Dazu existieren experimentelle Sprachen mit Sprachkonstrukten für die parallele Ausführung und Programmierschnittstellen wie OpenMP und MPI, mit denen bestehende Programmiersprachen erweitern lassen. Auf ganzer Linie durchsetzen konnte sich noch kein Konzept, und somit ist die Suche nach einer »besten« Lösung zur parallelen Softwareentwicklung ein offenes Forschungsthema.

Die Erstellung von effizienten, parallelen Programmen ist zeitaufwändig und fehleranfällig und unterscheidet sich grundsätzlich von der klassischen, sequenzielle Denkweise. Somit ist es schwierig, neue Software parallel zu entwickeln - noch schwieriger ist es aber, bestehende Quelltexte nachträglich zu parallelisieren, weil diese Veränderung ein tiefes Verständnis des Gesamtsystems verlangt.

Die *automatische Parallelisierung*, oder kurz Autoparallelisierung, sequentieller Programme ist daher ein interessanter Ansatz, weil sie die Komplexität des Prozesses in den Compiler auslagert. Neue und bestehende Software könnten so gleichermaßen profitieren.

Überblick Zunächst besprechen wir in Abschnitt 2, welche Anforderungen an eine parallele Programmausführung gestellt werden. In Abschnitt 3 betrachten wir Autoparallelisierung für imperative Programmiersprachen, so wie sie in den 80er- und 90er-Jahren erforscht wurde. Der Fokus liegt dabei auf der Ermittlung der Parallelität, weniger auf den anschließenden Transformationen und der Codeerzeugung. Abschnitt 4 behandelt ausgewählte, aktuelle Ansätze. In Abschnitt 5 ziehen wir ein Fazit zur Tauglichkeit der Autoparallelisierung.

2 Grundlagen der parallelen Programmausführung

Bei der parallelen Programmausführung entsteht durch Kommunikation und Synchronisation ein Mehraufwand gegenüber der sequentiellen Ausführung. Für eine effiziente parallele Programmausführung muss die *Granularität* der Parallelität, also das Verhältnis von Recheninstruktionen zu Kommunikationsanweisungen groß sein.

Üblicherweise werden folgende Granularitäten unterschieden:

- *Parallelität auf Befehlsebene* (engl. Instruction Level Parallelism). Ein Prozessor kann n Instruktionen gleichzeitig ausführen.
- *Datenparallelität* (engl. Data Parallelism). Ein Programm wird parallel von n Prozessorkernen ausgeführt.
- *Parallelität auf Taskebene* (engl. Task Parallelism). n Programme laufen zur gleichen Zeit auf einem Rechnersystem.

Das Bereitstellen von Parallelität auf Befehlsebene verbessert nicht die Auslastung eines Mehrprozessorsystems und spielt daher hier keine Rolle.
Datenparallelität und Parallelität auf Taskebene lassen sich durch die Aufteilung des Prgrammes auf mehrere Prozessorkerne ausnutzen. Parallelität auf Taskebene ist grobgranularer als Datenparallelität, allerdings enthalten Anwendungen im Allgemeinen nur konstant viele unterschiedliche Aufgaben [1], so dass eine so parallelisierte Anwendung nicht mit der Anzahl der verfügbaren Prozessoren skaliert. Die Menge der verfügbaren Datenparallelität ist hingegen üblicherweise abhängig von der Größe des Eingabedatensatzes. Im Folgenden konzentrieren wir uns auf die Suche nach Datenparallelität.

3 Autoparallelisierung für imperative Programmiersprachen

Numerische Anwendungen enthalten häufig Berechnungen, die aus mehrfach geschachtelten Schleifen bestehen, in denen auf Daten in statischen Arrays zugegriffen wird. Solche Schleifen sind eine ergiebige Quelle von Datenparallelität [1]. Autoparallelisierung von Schleifen wurde in den 80er- und 90er Jahren intensiv an den damals weitverbreiteten Sprachen Fortran und C erforscht. Bacon, Graham und Sharp haben ein umfassenden Überblick veröffentlicht [2].

Im Folgenden stellen wir Ansätze zum Auffinden geeigneter Datenparallelität in dieser Domäne vor. Die grundlegende Idee ist es, Schleifeniterationen zu suchen, die gleichzeitig ausgeführt werden dürfen.

3.1 Abhängigkeitsanalyse

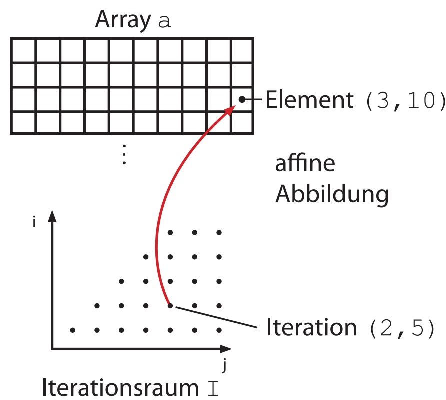
Der Autoparallelisierung muss wie anderen Transformationen auch eine Analysephase vorausgehen, die sicherstellt, dass sich das beobachtbare Verhalten des Programms nicht verändert.

Zwischen zwei Zugriffen auf die gleiche Speicherstelle, von denen mindestens eine eine Schreiboperation ist, besteht eine *Datenabhängigkeit*. Ist der erste Zugriff eine Schreiboperation und der zweite eine Leseoperation, handelt es sich um eine *echte Datenabhängigkeit*. Der umgekehrte Fall wird als *Gegenabhängigkeit* bezeichnet. Ein *Ausgabeabhängigkeit* liegt vor, wenn beide Zugriffe Schreiboperationen sind. Zwei Leseoperationen verursachen keine Abhängigkeit.

Die Autoparallelisierung kann Schleifeniterationen nur umordnen, wenn zwischen den Speicherzugriffen in ihrem Rumpf keine Datenabhängigkeiten bestehen, oder die bestehenden Datenabhängigkeiten beachtet werden. Dies bedeutet konkret, dass die Abfolge der Lade- und Speicheroperationen auf eine Speicherstelle im Vergleich mit der sequentiellen Ausführung nicht verändert werden darf. Datenabhängigkeiten, die zwischen verschiedenen Iterationen bestehen, werden als *schleifengetragene Abhängigkeiten*[1] bezeichnet.

Die Ermittlung präziser Abhängigkeitsinformationen ist ein zentrales Problem der Autoparallelisierung. Die hier beschriebenen Grundlagen dieser Technik stammen aus [2, 4].

(a) Abbildung Iterationsraum auf Datenraum



(b) Beispielcode

```
1 for (i=1; i<=5; i++)
2   for (j=i; j<=7; j++)
3     a[i+1,2*j] = 0;
```

Beispiel 1: Affine Arrayzugriffe

3.1.1 Affine Arrayzugriffe

Nun stellt sich folgende Frage: Wann haben zwei Speicherzugriffe das gleiche Ziel? Zugriffe auf skalare Variablen lassen sich einfach analysieren. Unter der Annahme, dass keine Zeiger oder Referenzen verwendet werden, bezeichnet eine Variable genau eine Speicherstelle. Diese Betrachtungsweise ist für Zugriffe innerhalb eines Arrays korrekt, aber zu unpräzise für die Autoparallelisierung. Jeder Arrayzugriff in einer Schleife würde dazu führen, dass Datenabhängigkeiten zwischen allen Schleifeniterationen angenommen werden müssen, die die Umordnung der Iterationen verhindern.

In Beispiel 1 ist eine geschachtelte Schleife mit einem Schreibzugriff auf ein zweidimensionales Array dargestellt. Die betroffene Speicherstelle ist von den Werten der *Arrayindizes* abhängig, die wiederum als Funktion über den Schleifenvariablen ausgedrückt ist[2]. Jede Schleifeniteration kann durch einen *Indexvektor*, der aus den Werten der Schleifenvariablen besteht, identifiziert werden. Die Menge der gültigen (bezüglich der Schleifengrenzen) Indexvektoren bildet den *Iterationsraum*. Es gibt also eine bijektive Abbildung vom Iterationsraum zu den Arrayindizes und somit auch zu den Datenelementen innerhalb des Arrays. Diese Abbildung wird *Arrayindexfunktion* genannt. Im obigen Beispiel ist die Arrayindexfunktion vektorwertig und lautet $f(i,j) = (f_1(i), f_2(j))$ mit $f_1(i) = i + 1$ und $f_2(j) = 2 \cdot j$.

Die Suche nach schleifengetragenen Abhängigkeiten lässt sich nun folgendermaßen formulieren: Seien `a[`$f(\vec{i})$`]` und `a[`$g(\vec{i})$`]` Arrayzugriffe mit den Arrayindexfunktionen f und g über den Schleifenvariablen, und sei mindestens einer der Zugriffe ein Schreibzugriff.

[1] engl. loop-carried dependences
[2] Wäre das nicht der Fall, wäre das Ziel des Arrayzugriffs schleifeninvariant und könnte wie eine skalare Variable behandelt werden.

Gibt es gültige Indexvektoren $\vec{x}, \vec{y}$ mit $\vec{x} \neq \vec{y}$, so dass $f(\vec{x}) = g(\vec{y})$?

Wenn f und g beliebige Funktionen sind, ist dieses Problem nicht entscheidbar (vgl. Satz v. Rice). Eine weit verbreitete Einschränkung ist, nur *affine*[3] *Arrayzugriffe* (d.h. Arrayzugriffe mit affinen Arrayindexfunktionen) zuzulassen. Beispiele für affine Arrayzugriffe sind `a[`i`]`, `a[`$i-1$`]`, `b[`$2 \cdot i + 2$`, `$3 \cdot j - 7$`]`, wohingegen es sich bei dem indirekten Zugriff `a[a[`i`]]` und bei Produkten von mehr als einer Schleifenvariable, z.B. `a[`$i \cdot j$`]`, nicht um affine Zugriffe handelt.

3.1.2 Test auf Unabhängigkeit

Mit der Einschränkung auf affine Arrayzugriffe können die schleifengetragenen Abhängigkeiten durch das Lösen von linearen Gleichungssystemen gefunden werden. Existieren keine Lösungen innerhalb des Iterationsraums, so gibt es keine schleifengetragenen Abhängigkeiten.

Das Bestimmen von ganzzahligen Lösungen eines linearen Gleichungssystems ist ein NP-schweres Problem. Autoparallelisierende Compiler wenden daher zuerst Heuristiken an. Können diese das Problem nicht entscheiden, kann man auf exakte Verfahren zurückgreifen, die praxisbezogene Einschränkungen nutzen. Ein Überblick ist in [2] zu finden.

ggT-Test Ein Ansatz ist, die Existenz ganzzahliger Lösungen generell auszuschließen. Dann ist klar, dass es auch im Iterationsraum keine ganzzahligen Lösungen gibt. Die Theorie ganzzahliger Gleichungen kennt folgendes Kriterium: Teilt der größte gemeinsame Teiler der Koeffizienten des linearen Teils der Gleichungen *nicht* den konstanten Term, so existieren keine ganzzahligen Lösungen. Beispiel 2 zeigt ein einfaches Beispiel zum ggT-Test.

`a[`$2i$`] = ...; ... = a[`$2i+1$`]`
Frage: Existieren x, y, so dass $2x = 2y + 1$?

$$\Longrightarrow \underbrace{2x - 2y}_{\text{linearer Teil}} = \underbrace{1}_{\text{konstanter Term}}$$

$$\Longrightarrow \text{ggT}(2,2) = 2 \nmid 1$$

Nein! Die Arrayzugriffe sind unabhängig.

Beispiel 2: Beispiel zum ggT-Test

Banerjee-Wolfe-Test Lässt sich mit dem ggT-Test nicht die Unabhängigkeit zeigen, kann man stattdessen prüfen, ob es überhaupt Lösungen innerhalb des Iterationsraumes gibt. Ist der minimale Wert des linearen Teils größer als der konstante Term, oder ist der maximale Wert des linearen Teils kleiner als der konstante Term, so existieren keine reellen und damit auch keine ganzzahligen Lösungen im Iterationsraum.

3.1.3 Abhängigkeitsvektoren

Vollständige Unabhängigkeit von Arrayzugriffen ist selten. Wenn Datenabhängigkeiten bestehen, muss der Compiler wissen, wie sie verlaufen. Ein Konzept, um Datenabhängigkeiten zu beschreiben, sind Abhängigkeitsvektoren[4]. Seien $\vec{x}, \vec{y}$ die Lösungen eines Gleichungssystem, welches eine Abhängigkeit beschreibt. Der Abhängigkeitsvektor $\vec{z}$ beschreibt das Verhältnis zwischen den einzelnen Komponenten von $\vec{x}$ und $\vec{y}$.

$$\vec{z} = (z_1, .., z_d), z_k = \begin{cases} = & \text{Abh. zur selben Iteration der k-ten Schleife} \\ < & \text{Abh. zu einer späteren Iteration der k-ten Schleife} \\ > & \text{Abh. zu einer früheren Iteration der k-ten Schleife} \end{cases}$$

[3] Affine Abbildungen lassen sich in der Form $\Phi(x_1, ..., x_n) = \sum_{i=1..n} \lambda_i x_i \; + c$ mit λ_i, c konstant, darstellen.
[4] engl. dependence vectors

(a) Modifizierter Beispielcode

```
1 for  (i=1; i<=5; i++)
2   for  (j=i; j<=7; j++)
3     a[i+1,2*j] = a[i,j]+c;
4       ^^^^^^^     ^^^
5          |          |
6          x          y
```

(b) Abgeleitete Gleichungen

$$x_i + 1 = y_i$$

$$2x_j = y_j$$

$$1 \leq x_i, y_i \leq 5$$

$$x_i \leq x_j \leq 7 \wedge y_i \leq y_j \leq 7$$

(c) Lösungen

$$[\vec{x}, \vec{y}] = \underbrace{[(1,1),(2,2)]}_{(<,<)}, \underbrace{[(1,2),(2,4)]}_{(<,<)}, ...$$

Beispiel 3: Abhängigkeitsvektoren

Beispiel 3 zeigt die Anwendung der bisher angesprochenen Konzepte. In der Schleife befindet sich ein Schreibzugriff auf das Array a mit den Arrayindexfunktionen $f_1(i) = i+1, f_2(j) = 2 \cdot j$ und ein Lesezugriff mit den Indexfunktionen $g_1(i) = i, g_2(j) = j$. Daraus leiten sich die Gleichungen 1 und 2 ab. Die verbleiben Gleichungen sind die Begrenzungen für den Iterationsraum und leiten sich aus den Schleifengrenzen ab. Das Gleichungssystem hat offensichtlich Lösungen, aus denen der Abhängigkeitsvektor $(<,<)$ abgelesen werden kann.

Ein Compiler bestimmt die Abhängigkeitsvektoren nicht aus gegebenen Lösungen, sondern testet für jeden möglichen Vektor, ob eine entsprechende Abhängigkeit besteht. Dazu nimmt er an, die Abhängigkeit bestünde, und versucht mit den oben genannten Mitteln, die Unabhängigkeit zu zeigen. Würde der Compiler auf $(=,<)$ testen wollen, ändert sich die dritte Gleichung in Beispiel 3 zu $1 \leq x_i = y_i \leq 5$. Hierfür lässt sich die Unabhängigkeit zeigen, da es keine ganze Zahl gibt, die $x + 1 = x$ erfüllt. Burke und Cytron [4] haben einen hierarchischen Abhängigkeitstest entwickelt, durch den man nicht alle 3^d möglichen Abhängigkeitsvektoren einer d-fach geschachtelten Schleife prüfen muss, um alle bestehenden Abhängigkeiten zu finden.

Mithilfe der nun gewonnenen Abhängigkeitsinformationen kann man eine Vielzahl von Transformationen durchführen (siehe [2]). Es ist aber unklar, welche Transformationen in welcher Reihenfolge angewendet werden sollen, um die Parallelität im Programm auszunutzen. Stattdessen betrachten wir ein neueres Verfahren, welches diese Einzeltransformationen subsumiert.

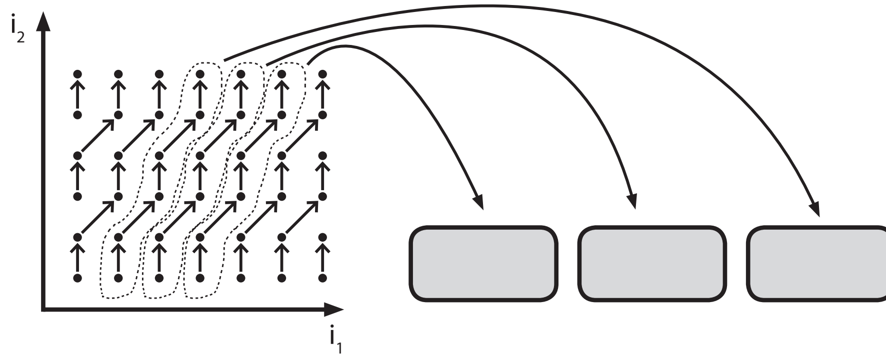
3.2 Affine Partitionierung

Lim und Lam [5] stellen einen Algorithmus vor, der sämtliche in einer geschachtelten Schleife vorkommende Parallelität ohne Synchronisation oder mit $O(1), O(n), ...$[5] Synchronisationsanweisungen findet. Die gefundene Parallelität ist so grobkörnig wie möglich.

Die Idee ist, eine Partitionierung des Iterationsraum zu berechnen. Zwischen Iterationsraum und Speicherstellen besteht bereits eine affine Abbildung, nämlich durch die Forderung, dass Arrayindizes und Schleifengrenzen affine Ausdrücke der umgebenden Schleifenvariablen sind.
Gesucht ist nun eine affine Abbildung, die jedem Indexvektor seine Partitionsnummer zuordnet. Die Datenabhängigkeiten werden durch Beschränkungen der Partitionen sichergestellt. Die konkrete Partitionierung kann man mit Standardverfahren der linearen Algebra berechnen.[5, 1]

Raumpartitionen Die Bedingung für die Raumpartitionen lautet: Besteht zwischen zwei Anweisungen eine Abhängigkeit, werden sie in die gleiche Partition abgebildet. Das bedeutet, dass zwischen zwei Raumpartitionen keine Abhängigkeiten existieren und die zugehörigen Anweisungen *synchronisationsfrei* ausgeführt werden können[6]. Ein mögliches Parallelisierungsschema ist,

[5] n = Anzahl Schleifeniterationen. Bei Bedarf lässt sich das Verfahren auch analog weiter anwenden, um Parallelität mit $O(n^2), O(n^3)$ usw. Synchronisationsweisungen zu finden.
[6] Innerhalb der Partitionen wird die ursprüngliche, sequentielle Ausführungsreihenfolge beibehalten.



Beispiel 4: Synchronisationsfreie Parallelität durch Raumpartitionen

jedem realen Prozessor ein oder mehrere Raumpartitionen zuzuweisen. In Beispiel 4 ist ein Iterationsraum mit eingezeichneten Abhängigkeiten dargestellt. Durch die Raumpartitionierung lassen sich die vorhandenen Abhängigkeitsketten erkennen.

Für den Spezialfall, dass nur eine Raumpartition existiert, wird der Programmabhängigkeitsgraph konstruiert. Er besteht aus Knoten für die statischen Instruktionen des Programms. Zwischen zwei Knoten verläuft eine Kante, wenn dynamische Instanzen der Instruktionen existieren, die datenabhängig sind. In diesem Graphen werden nun starke Zusammenhangskomponenten (SZK) gesucht. Anschließend wird für jede SZK die Raumpartitionierung versucht. Das zugehörige Parallelisierungsschema ist, die zu den Zusammenhangskomponenten zugehörigen Raumpartitionen parallel auszuführen. Zwischen zwei Zusammenhangskomponenten wird eine Barrierensynchronisation eingefügt. Da der Programmabhängigkeitsgraph ein statisches Konstrukt ist, kann es nur konstant viele SZKs und damit auch Barrieren geben. Die so gefundene Parallelität erfordert also konstant viele Synchronisationsanweisungen.

Zeitpartitionen Die Aufteilung in Zeitpartitionen ist wie folgt: Operationen, die sich in Zeitpartition i befinden, müssen aufgrund der Datenabhängigkeiten vor denen in der Zeitpartition $i+1$ ausgeführt werden. Dabei wird die Partitionierung so gewählt, dass so viele Abhängigkeiten wie möglichen implizit durch die Partitionierung erfüllt werden. Dadurch werden die Partitionen maximal groß.

Jedes mögliche Parallelisierungsschema für die Zeitpartionierung muss mindestens $O(n)$ Synchronisationsanweisungen beinhalten. Möglich ist beispielsweise, die Zeitpartitionen von einer Schleife ausführen zu lassen, an deren Rumpfende eine Barriere platziert ist. Lim und Lam schlagen aber vor, eine Softwarepipeline einzusetzen, da Barrieren durch Punkt-zu-Punkt-Synchronisation ersetzt werden können und die Pipelinelösung zu einer besseren Datenlokalität führt [5].

4 Aktuelle Forschung

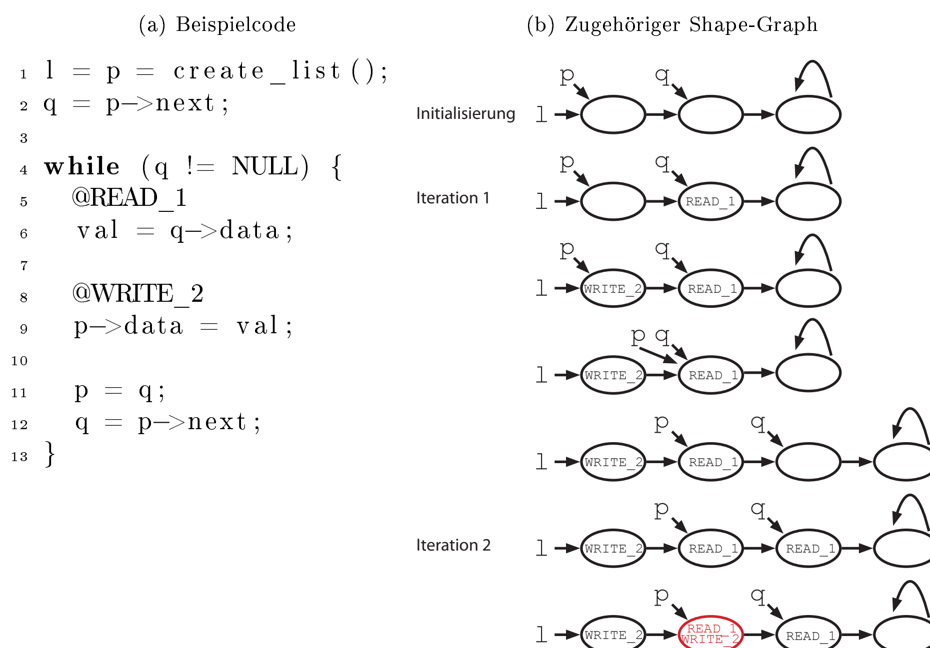
Heutzutage ist es gängige Praxis in der Softwareentwicklung

- aufwändige Objektstrukturen im Heapspeicher statt linearer Arrays, und
- über Methodengrenzen verteilte Algorithmen anstatt geschachtelter Schleifen

zu nutzen.

Die in Abschnitt 3 vorgestellten Verfahren funktionieren für diese neue Problemdomäne nicht mehr. Das zentrale Problem, die Vorhersage der Speicherzugriffe, ist nun nicht mehr so effizient und präzise wie bei affinen Arrayzugriffen möglich. Im Folgenden werden zwei Lösungsansätze aus der aktuellen Forschung betrachtet.

4.1 Abhängigkeitsanalyse mit Shape Analysis [6]



Beispiel 5: Abhängigkeitsanalyse mit Shape Analysis

Allgemein leitet man mithilfe der Shape Analysis Aussagen über den Aufbau von dynamischen Datenstrukturen im Heapspeicher her. Eine mögliche Aussage einer Datenstruktur ist beispielsweise, dass es sich um eine verkettete Liste handelt und auch nach Ausführen von einer Menge an Instruktionen diese Eigenschaft immer noch gilt.

Ein Beispiel, wie Shape Analysis zur Abhängigkeitsanalyse einer Liste eingesetzt werden kann, ist in Beispiel 5 zu sehen. Zunächst werden die Lese- und Schreibzugriffe auf die Listenelemente mit einer Beschriftung annotiert. Dann wird das Programm mittels *abstrakter Interpretation* analysiert.

Nach der Initialisierung zeigen `l` und `p` auf das erste Element, und `q` auf das zweite Element. Es ist unbekannt wie lang die Liste ist, die verbleibenden Listenelemente werden daher durch ein *Summary Node* dargestellt.

Die Zugriffe werden interpretiert und die betroffenen Listenelemente beschriftet. Am Ende der ersten Iteration zeigen `p` und `q` auf das zweite und dritte Listenelement. Beim Annotieren des zweiten Listenelements mit `WRITE_2` in der zweiten Iteration stellt die Analyse fest, dass dieses Element schon die Beschriftung `READ_1` trägt, und somit eine schleifengetragenene Abhängigkeit besteht.

4.2 Trace-basierte Autoparallelisierung [3]

Ein *Trace* ist eine Abfolge von paarweise verschiedenen Grundblöcken. Eine darauf aufbauende Autoparallelisierung ist nicht mehr auf eine Schleifenstruktur angewiesen und kann auch ohne Kenntnis der Quelltexte durchgeführt werden. Die Granularität liegt zwischen Daten- und Task-parallelität. Die Autoren konnten während ihrer Experimente mit Java-Benchmarks Parallelität für bis zu vier Prozessoren ermitteln.

5 Schluss

Wir haben gesehen, dass die automatische Parallelisierung von numerischen Anwendungen möglich ist. Das Kernproblem besteht darin, präzise Informationen über die im Programm enthaltenen Datenabhängigkeiten zu ermitteln. Dazu ist es erforderlich, konkurrierende Speicherzugriffe statisch ermitteln zu können. Reguläre Zugriffsmuster, wie sie etwa bei affinen Arrayzugriffen in geschachtelten Schleifen erfolgen, eignen sich gut für die Analyse. Die Erweiterung der Abhängigkeitsanalyse für Programme, die nicht nach diesem Muster aufgebaut sind, ist ein aktuelles Forschungsthema.

Zu Beginn haben wir die Frage gestellt, ob die automatische Parallelisierung die für den Programmierer schwierige explizite Parallelisierung ersetzen kann. Diesem Anspruch werden die vorgestellten Verfahren nicht gerecht. Eine automatische Parallelisierung kann bestenfalls nur die Parallelität, die im verwendeten Algorithmus vorhanden ist, finden und nutzen. Ein Programmierer kann durch eine ungeschickte Implementierungsentscheidung die Parallelisierung unmöglich machen. Die automatische Parallelisierung ist in zu wenigen Fällen anwendbar, als dass bei der Softwareentwicklung generell auf Überlegungen zur Parallelisierung des Programms verzichtet werden kann.

In den Domänen, in denen sie anwendbar ist, ist die automatische Parallelisierung jedoch eine interessante Optimierung, die ein Compiler durchführen sollte. Transformationen zur Vergrößerung der Parallelität verbessern gleichzeitig auch die Datenlokalität und liefern somit auch auf Einprozessorsystemen durch Cache-Effekte Laufzeitgewinne [1].

References

[1] Aho, A. V., Ed. *Compiler : Prinzipien, Techniken und Werkzeuge*, 2., aktualisierte aufl. ed. it Informatik. Pearson Studium, München [u.a.], 2008, ch. 11, pp. 929–1090.

[2] Bacon, D. F., Graham, S. L., and Sharp, O. J. Compiler transformations for high-performance computing. *ACM Comput. Surv. 26*, 4 (1994), 345–420.

[3] Bradel, B. J., and Abdelrahman, T. S. A study of potential parallelism among traces in java programs. *Science of Computer Programming 74*, 5-6 (2009), 296 – 313. Special Issue on Principles and Practices of Programming in Java (PPPJ 2007).

[4] Burke, M., and Cytron, R. Interprocedural dependence analysis and parallelization. In *SIGPLAN '86: Proceedings of the 1986 SIGPLAN symposium on Compiler construction* (New York, NY, USA, 1986), ACM, pp. 162–175.

[5] Lim, A. W., and Lam, M. S. Maximizing parallelism and minimizing synchronization with affine transforms. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 1997), ACM, pp. 201–214.

[6] Tineo, A., Corbera, F., Navarro, A., Asenjo, R., and Zapata, E. On the automatic detection of heap-induced data dependencies with interprocedural shape analysis. In *Parallel Processing Workshops, 2009. ICPPW '09. International Conference on* (22-25 2009), pp. 378 –385.