UNIVERSITÄT PASSAU

Seminar:

Parallelisierung irregulärer Probleme

Thema:

Ein Compiler für dünne lineare Algebra

Markus Keidl

Prof. C. Lengauer, Ph. D.
Lehrstuhl für Programmierung

Inhaltsverzeichnis

1 Motivation **3**

2 Dünne Matrizen **4**

3 Der Sparse-Compiler **5**
3.1 Kennzeichnung von dünnen Matrizen 6
3.2 Arbeitsprinzip 7
3.3 Transformationen 8

4 Performance-Betrachtungen **10**
4.1 Algorithmen für dünne Matrizen 10
4.1.1 Matrix mal Vektor 10
4.1.2 Matrix mal Matrix 11
4.1.3 Ergebnisse 12
4.2 Parallelität 12
4.2.1 Datenabhängigkeiten 13
4.2.2 Schleifensynchronisation 15

5 Zusammenfassung **17**

1 Motivation

Das Rechnen mit Matrizen nimmt in vielen Teilgebieten der Mathematik, der Physik und des Ingenieurwesens einen zentralen Platz ein. Matrizen werden dabei oft mit Vektoren oder anderen Matrizen multipliziert oder in spezielle Teilmatrizen zerlegt. Da dabei fast immer mit immensen Datenmengen operiert wird, ist es notwendig, die entsprechenden Berechnungen effizient bezüglich Speicherverbrauch und Laufzeit auszuführen.

In der Praxis zeigt sich, daß bei derartigen Berechnungen auftretende Matrizen oft nur dünn besetzt sind, d. h. viele Null-Einträge besitzen. Die Effizienz von solchen Berechnungen läßt sich deshalb auf zwei Arten steigern:

- Man speichert möglichst nur die Nichtnull-Werte einer Matrix ab. Dadurch wird der Verbrauch von Speicher in den meisten Fällen reduziert. Natürlich darf dadurch die Berechnung insgesamt nicht langsamer werden.

- Bei Berechnungen mit dünn besetzten Matrizen[1] darf nicht mit den Null-Einträgen gerechnet werden, wenn sie keine Auswirkungen auf das Ergebnis haben. Man beachte, daß eine simple Abfrage der Form $A(i, j) = 0$ vor einer Berechnung zu keiner Effizienzsteigerung führt, da sie für jeden Eintrag der Matrix ausgeführt werden muß.

Aufgrund dieser Problematik gibt es für dünne Matrizen spezielle Algorithmen, die auf deren Eigenschaften Rücksicht nehmen. Gewöhnlich wählt der zuständige Programmierer einen für das gegebene Problem passenden Algorithmus aus. Allerdings muß dann natürlich der Sourcecode geändert werden, wenn sich die Struktur[2] der Matrizen ändert. Sinnvoller wäre es deshalb, die Berechnungen allgemein zu halten, d. h. keine Voraussetzung über die Struktur der Matrix zu machen. Der auf diese Weise erstellte Sourcecode müßte dann von einem speziellen Restrukturierungscompiler so transformiert werden, daß für jede Berechnung der optimale Algorithmus gewählt wird. Zusätzlich sollte dieser Compiler in der Lage sein, Beispielmatrizen, die etwa in einer Datei in einem festem Format vorliegen, zu analysieren, und dann den Sourcecode passend zu der Analyse umzuformen. Damit müßte man den Sourcecode nur neu compilieren, falls sich die Struktur der dünnen Matrizen ändern sollte, und somit nichts am ursprünglichen Code verändern.

In den folgenden Abschnitten wird nun ein derartiger Compiler vorgestellt. Dazu wird im nächsten Abschnitt zuerst genauer auf dünne Matrizen und für sie geeignete Speicherschemata eingegangen. Anschließend folgt eine Betrachtung der Arbeitsweise des *Sparse Compilers* [BW96b, BW95, BBW, Bik96] , eines Compilers der genau die oben genannten Voraussetzungen erfüllt. Das vorletzte Kapitel

[1] Kurz: *dünne* Matrizen

[2] Unter der Struktur einer Matrix wird in diesem Artikel die Verteilung der Nichtnull-Einträge verstanden.

widmet sich dann genauer der Performance, die mit einem solchen Compiler und dem daraus generierten Code erreicht werden kann. Eine Zusammenfassung bildet dann den Schlußpunkt dieses Seminarartikels.

2 Dünne Matrizen

In Abschnitt 4.1 werden später Algorithmen vorgestellt, die mit dünnen Matrizen arbeiten. Um die dabei auftretenden Sourcecode-Fragmente verstehen zu können, ist es nötig, etwas genauer auf die mögliche Struktur von dünnen Matrizen und das darausfolgende, vom Sparse Compiler gewählte Speicherschema einzugehen.

Eine Matrix, die ausreichend viele Null-Einträge besitzt, wird als *dünn* bezeichnet. Ausreichend bedeutet in diesem Sinne, daß sich das Ausnutzen dieser Eigenschaft bezahlt macht: in Rechenzeit, in Speicherplatz oder in beidem, je nachdem, welche Anforderungen an das Programm gestellt werden. Alle anderen Matrizen werden *dicht* genannt.

Definition 1 Für eine dünne $m \times n$-Matrix definiert man folgende Mengen:

- Die Menge $Nonz(A) = \{(i,j) \in [1,m] \times [1,n] \mid a_{ij} \neq 0\}$ wird **Nichtnull-Struktur** genannt.
- $\mathbf{E_A}$ bezeichnet die Elemente einer dünnen Matrix, die explizit in der die Matrix repräsentierenden Datenstruktur gespeichert werden, auch wenn sie Null sein sollten.

Wie oben erwähnt, kann die Menge E_A durchaus Nullwerte enthalten, etwa wenn die verwendete Datenstruktur dadurch effizienter wird. Es gilt also:

$$Nonz(A) \subseteq E_A \subseteq [1,m] \times [1,n]$$

Weiterhin werden Elemente aus E_A als *Einträge* bezeichnet, Elemente die nicht in E_A enthalten sind, nennt man *Nicht-Einträge*.

Dünne Matrizen können eine beliebige Struktur haben. So können nur bestimmte Diagonalen besetzt sein oder aber es lassen sich rechteckige Bereiche bestimmen, die als einzige Nichtnull-Einträge besitzen.

Vor allem im Hinblick auf die in 4.1 folgenden Beispiele soll deshalb ein Speicherschema, *zeilenweise Abspeicherung* genannt, vorgestellt werden, daß keine Voraussetzungen an die Matrix stellt, außer daß sie eben dünn ist. Bei dieser Technik werden alle Elemente aus E_A zeilenweise und hintereinander in einem Array abgelegt. Insgesamt sind folgende Datenstrukturen nötig:

- Ein Array `VAL_A`, das die Einträge aus E_A aufnimmt.
- Ein Array `IND_A`, das zu jedem Eintrag aus `VAL_A` dessen Spaltenindex angibt.

- Zwei weitere Arrays `LOW_A` und `HGH_A`. Dabei gibt der an i-ter Stelle stehende Wert von `LOW_A` die Position an, an der die Einträge der i-ten Zeile in `IND_A` und `VAL_A` beginnen. Der an i-ter Stelle stehende Wert von `HGH_A` hingegen bestimmt, wo die Einträge der i-ten Zeile in `IND_A` bzw. `VAL_A` enden.

Folgendes Beispiel demonstriert dieses Speicherschema anhand einer gegebenen Matrix.

Beispiel 1 Sei folgende Matrix gegeben:

$$A = \begin{pmatrix} a_{11} & & a_{13} & a_{14} & \\ & a_{22} & & & a_{25} \\ a_{31} & & a_{33} & a_{34} & \\ a_{41} & & & a_{44} & \\ & & a_{53} & & a_{55} \end{pmatrix} \qquad (1)$$

Die Nichtnull-Einträge von A werden nun in dem Array `VAL_A` abgelegt, Abbildung 1 illustriert dies. Wie ersichtlich müssen die einzelnen Einträge einer Zeile nicht geordnet sein. `IND_A` enthält jeweils den Spaltenindex des entsprechende Eintrags in `VAL_A`. `LOW_A` verweist auf den ersten Eintrag einer Zeile, `HGH_A` auf den letzten.

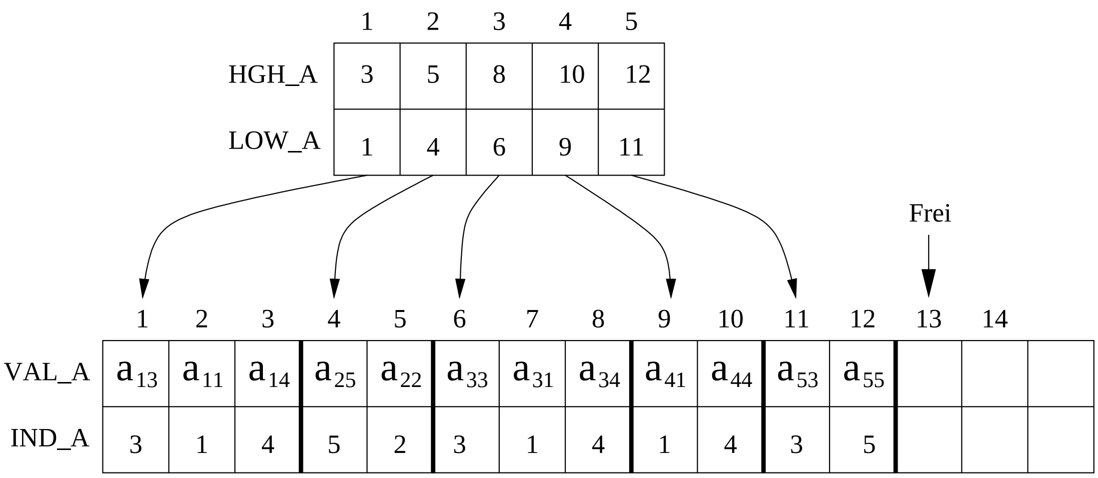


Abbildung 1: Speicherschema für eine zeilenweise abgespeicherte Matrix

3 Der Sparse-Compiler

Ein Restrukturierungscompiler, der die in Abschnitt 1 erwähnten Eigenschaften bietet, wird in [BW96b] vorgestellt[3]. Er transformiert normalen, mit einigen

[3]Im folgenden wir dieser Compiler als *Sparse Compiler* bezeichnet

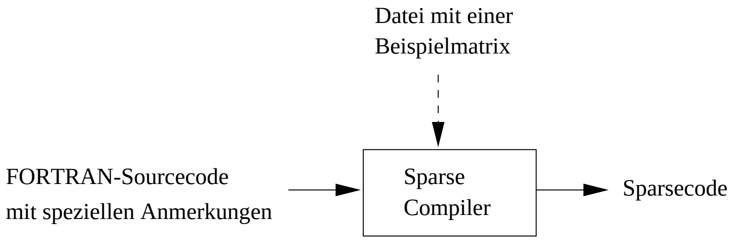


Abbildung 2: Das Prinzip des Sparse Compilers

zusätzlichen Anmerkungen versehenen FORTRAN-Sourcecode in *Sparse-Code*, d. h. für dünne Matrizen optimierten FORTRAN-Sourcecode (siehe Abbildung 2). Folgende Begriffe gilt es im folgenden klar zu unterscheiden:

- Die dünn besetzte $m \times n$-Matrix A.
- Das Array `REAL A(M,N)`, das als Datenstruktur für die Matrix A vom Programmierer gewählt wird.
- Das Speicherschema `A'`, das der Sparse Compiler statt des Arrays `A` für die Matrix A wählt.

Diese Unterscheidung ist nötig, um zum einen über mathematische Berechnungen, wie etwa $b = Ax + b$, sprechen zu und andererseits über die Implementierung im Sourcecode Aussagen treffen zu können. Der Programmierer selbst wird nur mit dem Array `A` arbeiten, erst der Sparse Compiler wählt eine effiziente Datenstruktur, um die dünne Matrix A abzuspeichern.

3.1 Kennzeichnung von dünnen Matrizen

Natürlich kann der Sparse Compiler nicht zwischen normalen Arrays und Arrays, die dünne Matrizen repräsentieren, unterscheiden. Deshalb muß man ihm diese Informationen geben. Dies geschieht durch Einfügen von *Anmerkungen* in den ursprünglichen FORTRAN-Code. Durch eine solche Anmerkung wird ein Array explizit als Repräsentation einer dünnen Matrix gekennzeichnet. In Beispiel 2 wird dargestellt, wie ein Array `A` als Repräsentation für die in Beispiel 1 vorgestellte dünne 5×5-Matrix gekennzeichnet wird:

Beispiel 2

```
REAL A(5,5)
C_SPARSE(A)
```

Nun reicht es aber im allgemeinen nicht, eine Matrix einfach als dünn zu markieren. Für die Wahl eines effizienten Speicherschemas sollten auch Informationen über die Verteilung der Einträge, d. h. der Nichtnull-Werte der Matrix vorliegen. Hier bietet der Sparse Compiler zwei Alternativen:

- Sollten die Matrizen in Dateien vorliegen, so kann der Compiler diese Dateien automatisch analysieren und dann ein dazu passendes Speicherschema wählen. Dazu wird der Dateiname bei der Anmerkung mitangegeben:

  ```
  C_SPARSE(A, FILE('demo_matrix.cs'))
  ```

- Ist die Struktur der Matrizen explizit bekannt, so kann der Programmierer mit Hilfe von Anmerkungen dem Compiler diese mitteilen:

  ```
  C_SPARSE(A, _ZERO(1 ≤ I-J ≤ -1))
  ```

  Diese Anmerkung bedeutet, daß das Array `A` eine dünne Matrix repräsentiert, bei der die drei Diagonalen Null sind, für deren Einträge die Ungleichung `1 ≤ I-J ≤ -1` gilt.

Wird wie in Beispiel 2 keine solche zusätzliche Anmerkungen angegeben, so setzt der Sparse Compiler keine spezielle Struktur voraus. In diesem Fall speichert er die als dünn gekennzeichnete Matrix in einer allgemeinen Datenstruktur ab, also etwa zeilenweise (siehe Abschnitt 2). Eine ausführliche Dokumentation des Sparse Compilers findet sich in [BBW].

3.2 Arbeitsprinzip

Nachdem der Compiler durch die im Sourcecode befindlichen Anmerkungen also weiß, welche Arrays dünne Matrizen repräsentieren, kann er sämtliche Zugriffe auf Elemente dieses Array erkennen und auswerten.

Wesentlich für die Verkürzung der Rechenzeit ist nun die Beobachtung, daß Anweisungen, bei denen einem Nicht-Eintrag der Wert Null zugewiesen wird oder bei denen die linke Seite gleich der rechten Seite ist, nicht ausgeführt werden müssen. Der zweite Fall kann etwa bei einer Anweisung des Typs `ACC = ACC + A(4,4)` auftreten. Sollte nun `A(4,4)` Null sein, so wäre die entstehende Anweisung `ACC = ACC` überflüssig.

Nachdem der Compiler also diese Zugriffe analysiert und daraus bestimmte Zugriffsmuster isoliert hat - vor allem in Schleifen ist dies wesentlich - kann er ein für diese Muster passendes Speicherschema wählen und die Anweisungen, die die entsprechenden Zugriffe enthalten, so transformieren, daß sie nun auf das gewählte Speicherschema zugreifen (Weitere Informationen lassen sich in [BW93] und [BW94] finden).

3.3 Transformationen

Der Compiler betrachtet jede Anweisung, die einen Zugriff auf ein Array beinhaltet, das eine dünne Matrix repräsentiert. Symbolisiert $A(F(\overline{I}))$ den Zugriff auf ein solches Array, wobei die affine Abbildung $F : \mathcal{Z}^d \rightarrow \mathcal{Z}^2$ die Indizes liefert, so wird die Bedingung $F(\overline{I}) \in E_A$ in einer IF-Anweisung genutzt, um zwischen Operationen auf Einträgen und Nicht-Einträgen unterscheiden zu können.

Beispiel 3

```
      DO I = 1,N
        DO J = 1,N
S1:       ACC = ACC + A(I,J)
        ENDDO
      ENDDO
```

$\overline{I}$ entspricht den umgebenden Schleifenindizes einer Anweisung. Im Falle von S_1 gilt also: $\overline{I} = (I, J)$. Betrachtet man nun die Schleifeniteration mit $I = 2$ und $J = 3$, so liefert $F(2, 3) = (2, 3)$ die Position des Eintrags in `A`, auf den in dieser Iteration in S_1 zugegriffen wird.

Im folgenden wird eine abstrakte Datenstruktur `A'` als Repräsentation der Datenstruktur verwendet, die später vom Sparse Compiler wirklich verwendet wird (etwa eine zeilenweise Abspeicherung, siehe 2). Eine weitere, bijektive Funktion $\sigma_A : E_A \rightarrow AD_A$ sorgt dann für die Umwandlung der Indizes der Array-Einträge in Adressen bezüglich `A'`. Dabei symbolisiert AD_A den Adreßraum von `A'`. Der Zugriff auf ein Element einer dünnen Matrix A kann beispielsweise folgendermaßen dargestellt werden:

Beispiel 4

$$
\mathtt{S} = \mathtt{A}(\mathtt{F}(\overline{\mathtt{I}})) \quad \rightarrow \quad
\begin{array}{l}
\mathtt{IF}\ \mathtt{F}(\overline{\mathtt{I}}) \in \mathtt{E_A}\ \mathtt{THEN} \\
\quad \mathtt{S} = \mathtt{A}'[\sigma_{\mathtt{A}}(\mathtt{F}(\overline{\mathtt{I}}))] \\
\mathtt{ELSE} \\
\quad \mathtt{S} = 0.0 \\
\mathtt{ENDIF}
\end{array}
$$

Allerdings erscheint diese Unterscheidung hierbei umständlich, deshalb wird die Funktion σ_A so definiert, daß $\sigma_A(F(\overline{I})) = \perp$, falls $F(\overline{I}) \notin E_A$ und $A'[\perp] = 0$. Andererseits kann diese Unterscheidung in anderen Situationen durchaus sinnvoll sein, vor allem wenn Verzweigungen, in denen die Dünnheit ausgenutzt wird, eliminiert werden können. Etwa:

Beispiel 5

$$
\mathtt{ACC} = \mathtt{ACC} + \mathtt{A}[\mathtt{F}(\overline{\mathtt{I}})] \quad \rightarrow \quad
\begin{array}{l}
\mathtt{IF}\ \mathtt{F}(\overline{\mathtt{I}}) \in \mathtt{E_A}\ \mathtt{THEN} \\
\quad \mathtt{ACC} = \mathtt{ACC} + \mathtt{A}'[\sigma_{\mathtt{A}}(\mathtt{F}(\overline{\mathtt{I}}))] \\
\mathtt{ENDIF}
\end{array}
$$

Die Bedingungsprüfungen und die Aufrufe der σ_A-Funktion stellen den Overhead da, der durch die Verwendung von kompakten Datenstrukturen entsteht. Eine verbreitete Technik zur Eliminierung dieses Overheads ist die Verkapselung einer dominierenden Bedingung, d. h. einer Bedingung, die bestimmt, welche Iterationen einer Schleife ausgeführt werden, in das *Execution Set* der umgebenden Schleife. Dazu muß man die Adressen aller Einträge entlang des Zugriffsmusters dieser Schleife leicht berechnen können, wodurch man eine semantisch äquivalente Schleife mit weniger Iterationen und Overhead erhält. Als Beispiel diene die Verkapselung der Bedingung $(\mathtt{I},\mathtt{J}) \in \mathtt{E_A}$ in die Grenzen der J-Schleife in folgendem Codefragment[4]:

Beispiel 6

$$
\begin{array}{lcl}
\mathtt{DO\ I} = \mathtt{1,N} & & \mathtt{DO\ I} = \mathtt{1,N} \\
\quad \mathtt{DO\ J} = \mathtt{1,N} & & \quad \mathtt{DO\ AD} \in \mathtt{PAD_A^I} \\
\quad\quad \mathtt{IF}\ (\mathtt{I},\mathtt{J}) \in \mathtt{E_A}\ \mathtt{THEN} & & \quad\quad \mathtt{J} = \pi_2 * \sigma_{\mathtt{A}}^{-1}(\mathtt{AD}) \\
\quad\quad\quad \mathtt{ACC} = \mathtt{ACC} + \mathtt{A}'[\sigma_{\mathtt{A}}(\mathtt{I},\mathtt{J})] & \rightarrow & \quad\quad \mathtt{ACC} = \mathtt{ACC} + \mathtt{A}'[\mathtt{AD}] * \mathtt{J} \\
\quad\quad \mathtt{ENDIF} & & \quad \mathtt{ENDDO} \\
\quad \mathtt{ENDDO} & & \mathtt{ENDDO} \\
\mathtt{ENDDO} & &
\end{array}
$$

Dabei enthält $\mathtt{PAD_A^I}$ die Adressen alle Einträge entlang folgendem Zugriffsmusters:

$$\{(\mathtt{I},\mathtt{J}) | \mathtt{1} \leq \mathtt{J} \leq \mathtt{N}\}$$

Zusätzlich müssen die Spaltenindizes, d. h. die Werte $\pi_2 * \sigma_A^{-1}(AD)$, wobei $\pi_i * \vec{x} = x_i$, für jeden Eintrag vorhanden sein, um den Schleifenindex wiederherstellen zu können.

Beispiel 7 Nimmt man für die dünne Matrix A aus obigem Beispiel die Datenstruktur aus Abschnitt 2, so würden die Einheiten in `VAL_A` abgespeichert, das Array `IND_A` würde die Spaltenindizes enthalten und `LOW_A` sowie `HGH_A` würden für jede Zeile auf die entsprechenden Spalteneinträge in `IND_A` verweisen. Verwendet man obigen Beispielcode mit der Matrix aus Abbildung 3, so generiert der Sparse Compiler folgenden Sparsecode:

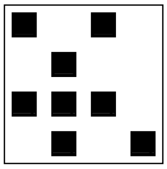
```
DO I = 1,N
  DO AD = LOW_A(I),HGH_A(I)
    J = IND_A(AD)
    ACC = ACC + VAL_A(AD) * J
  ENDDO
ENDDO
```

Dabei werden die Bedingungsprüfungen und σ_A-Aufrufe zusammengefaßt und in entsprechende Abfragen und Prozedurenaufrufe umgewandelt. Referenzen auf

[4] Im folgenden sei der Einfachheit halber $F = id_{\mathcal{Z}^2}$.

LOW_A	1	3	5	9
HGH_A	2	3	7	10

A



	1	2	3		5		7		9	10
VAL_A	a_{11}	a_{13}	a_{22}		a_{31}	a_{33}	a_{32}		a_{42}	a_{44}
IND_A	1	3	2		1	3	2		2	4

Abbildung 3: Beispiel für eine dünne Datenstruktur

Konstrukte wie $A', \pi_i * \sigma_A^{-1}$ und PAD_A^I werden durch Referenzen auf die entsprechenden Datenstrukturen ersetzt.

4 Performance-Betrachtungen

Obwohl die Verwendung eines Sparse Compilers das Entwickeln von optimiertem Code vereinfacht, muß doch auch gewährleistet werden, daß bezüglich Speicherverbrauch und Laufzeit der generierte Sparsecode effektiver als der ursprüngliche Programm ist. Zusätzlich sollte er kaum ineffizienter sein als „von Hand“ erstellter Code. Die folgenden beiden Unterabschnitte sollen Aufschluß darüber liefern, inwieweit der Sparse Compiler dieses Ziel erreicht.

4.1 Algorithmen für dünne Matrizen

Im folgenden werden nun zwei allgemeine mathematische Probleme und ihre algorithmischen Implementierungen untersucht. Dabei wird insbesondere auf die Unterschiede der normalen und der dünnen Algorithmen[5] Wert gelegt.

4.1.1 Matrix mal Vektor

Eine der grundlegensten Algorithmen für das Rechnen mit Matrizen ist die Multiplikation einer Matrix A mit einem Vektor x, d. h. der Ergebnisvektor $b = Ax$ soll berechnet werden. Ein Algorithmus, der dies leistet, wäre:

[5] Algorithmen, die an die Struktur von dünnen Matrizen angepaßt sind.

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

`N` bezeichnet die Größe der Matrix. Wird der Sparse Compiler nun auf dieses Programm angewendet, so liefert er abhängig von der Struktur der Matrix A ein äquivalentes „dünnes" Programm. Als Beispiel soll das in 2 vorgestellte Speicherschema dienen, d. h. es wird keine bestimmte Struktur der Matrix vorausgesetzt. Kennzeichnet man A als allgemeine dünne Matrix - sie wird dann also zeilenweise, entsprechend Abbildung 1 gespeichert - transformiert der Sparse Compiler das vorherige Codebeispiel folgendermaßen:

```
REAL    VAL_A(1:SIZE_A)
INTEGER IND_A(1:SIZE_A)
INTEGER LOW_A(1:N), HGH_A(1:N)
...
DO I = 1, N
  DO J_ = LOW_A(I), HGH_A(I)
     J = IND_A(J_)
     B(I) = B(I) + VAL_A(J_) * X(J)
  ENDDO
ENDDO
```

`SIZE_A` sei hierbei ausreichend groß, so daß alle Einträge aus E_A in den entsprechenden Arrays gespeichert werden können. Wie man sieht, iteriert die innere Schleife nur über die Werte einer Zeile, die nicht Null sind.

4.1.2 Matrix mal Matrix

Ebenso elementar wie die Multiplikation von Matrix und Vektor ist die Berechnung des Produktes von zwei Matrizen. Folgender Code berechnet das Ergebnis von $C = C + AB$ auf herkömmliche Weise (A, B und C seien dabei normale $n \times n$-Matrizen):

```
DO I = 1, N
  DO J = 1, N
    DO K = 1, N
      C(I,J) = C(I,J) + A(I,J) * B(I,J)
    ENDDO
  ENDDO
ENDDO
```

Bei der Verwendung von allgemeinen, zeilenweise abgespeicherten dünnen Matrizen wird folgendes Codesfragment generiert:

```
DO I = 1, N
  DO J = 1, N
    DO K_ = LOW_A(I), HGH_A(I)
      K = IND_A(K_)
      C(I,J) = C(I,J) + VAL_A(K_) * B(K,J)
    ENDDO
  ENDDO
ENDDO
```

Man sollte beachten, daß bei diesem Beispiel nur die Matrix A als dünn angenommen wurde. Über die anderen beiden Matrizen B und C werden keine Voraussetzungen gemacht, da der generierte Code sonst zu unübersichtlich werden würde.

4.1.3 Ergebnisse

Performance-Tests, die in [BW96b] genauer beschrieben werden, haben ergeben, daß die Ausnutzung der Dünnheit die Rechenzeit wesentlich reduziert, auch bei einer allgemeinen dünnen Matrix, die zeilenweise abgespeichert wurde und über deren Struktur keine weiteren Aussagen gemacht wurden. Lagen definitive Erkenntnisse über die Struktur der verwendeten Matrizen vor, so reduzierte sich die Rechenzeit nochmals. Während die Laufzeit des normalen, nicht auf dünne Matrizen ausgelegten Programms bei steigender Matrixgröße annähernd exponentiell Anstieg, entwickelten sich die Laufzeiten bei den dünnen Programmen fast linear.

4.2 Parallelität

Wichtig bei der Entwicklung und Generierung von parallelen Programmen ist es, Datenabhängigkeiten im Sourcecode zu erkennen, denn nur unabhängige Anweisungen können parallel ausgeführt werden. Leider versagen die meisten aktuellen Compilertools im Falle von Sparsecode, sie liefern oft sehr konservative Schätzungen bezüglich der Datenabhängigkeiten. Dies resultiert aus der Kompaktheit der Datenstrukturen, die bei für dünne Matrizen optimierten Algorithmen verwendet werden, um sowohl den Speicherplatz als auch die Rechenzeit zu minimieren.

Der in Abschnitt 3 vorgestellte Compiler bietet nun die Möglichkeit, das obengenannte Problem zu beheben. Ihm wird normaler Sourcecode mit einigen speziellen Anmerkungen zur Verfügung gestellt. Er transformiert diesen dann in optimierten Code. Deshalb kann er Standardverfahren zur Prüfung von Datenabhängigkeiten auf den *ursprünglichen Sourcecode* anwenden und mit Hilfe der dabei gewonnenen Informationen die mögliche Parallelität im generierten *Sparsecode* bestimmen, zumal er zusätzlich die Datenstrukturen auswählt und den normalen Code in Sparsecode transformiert. Er besitzt also bei weitem mehr

Informationen und einen größeren Überblick als ein normales Tool für Datenabhängigkeitsprüfung haben kann.

Da die meisten rechenintensiven Programme einen Großteil ihrer Rechenzeit in Schleifen verbringen, soll die Betrachtung von Parallelität im folgenden darauf beschränkt bleiben.

4.2.1 Datenabhängigkeiten

Um über die Abhängigkeiten von Daten sprechen zu können, ist es zuerst nötig, die Anweisungen, die diese Daten manipulieren, in eine eindeutige Reihenfolge zu bringen.

Definition 2 Sei S eine Anweisung in einer geschachtelten Schleife und seien $I_1, \ldots, I_d$ die umgebenden Schleifenindizes.

- $S(\overline{I})$ mit $\overline{I} = (I_1, \ldots, I_d)$ wird eine **indizierte Anweisung** genannt.
- Ersetzt man die einzelnen Schleifenindizes durch gültige Werte, so erhält man eine **Instanz** dieser Anweisung.

Die Verschachtelungstiefe 0 kennzeichnet Anweisungen, die in keiner Schleife enthalten sind und demzufolge nur eine Instanz besitzen.

Anweisungen sind entsprechend ihrer Ausfürungsreihenfolge geordnet.

Definition 3 Seien $S_l(\vec{i})$ und $S_m(\vec{j})$ Instanzen zweier Anweisungen. Dann gilt:

$$S_l(\vec{i}) <_o S_m(\vec{j}), \quad \begin{array}{l} \text{wenn } i - j \text{ lexikographisch positiv ist oder} \\ \text{wenn } i = j \text{ und } S_l \text{ im Programm lexikalisch vor } S_m \text{ liegt} \end{array}$$

Dabei wird aber angenommen, daß nur Schleifen betrachtet werden, die beide Anweisungen umgeben und diese eine positive Schrittweite aufweisen.

Um gültige Transformationen des Sourcecodes durchführen zu können, müssen die Abhängigkeiten zwischen den einzelnen Anweisungen bzw. ihren Daten erkannt werden. Dann ist es möglich, Umformungen durchzuführen, ohne die Semantik des Programms zu ändern. Im folgenden sollen die Mengen **IN** und **OUT** alle Variablen einer Anweisung enthalten, die gelesen bzw. geschrieben werden. Nun kann man verschieden Arten von Datenabhängigkeiten zwischen zwei Anweisungsinstanzen definieren:

Definition 4 Datenabhängigkeiten
Seien $S_l(\vec{i})$ und $S_m(\vec{j})$ zwei Anweisungsinstanzen, für die gilt:

$$S_l(\vec{i}) <_o S_m(\vec{j})$$

Dann folgt:

- Wenn $OUT(S_l(\vec{i})) \cap IN(S_m(\vec{j})) \neq \emptyset$, besteht zwischen $S_l(\vec{i})$ und $S_m(\vec{j})$ eine **Flußabhängigkeit**.
- Wenn $IN(S_l(\vec{i})) \cap OUT(S_m(\vec{j})) \neq \emptyset$, besteht zwischen $S_l(\vec{i})$ und $S_m(\vec{j})$ eine **Antiabhängigkeit**.
- Wenn $OUT(S_l(\vec{i})) \cap OUT(S_m(\vec{j})) \neq \emptyset$, besteht zwischen $S_l(\vec{i})$ und $S_m(\vec{j})$ eine **Outputabhängigkeit**.

Zwischen zwei Anweisungen S_l und S_m besteht eine Datenabhängigkeit, kurz $S_l \delta S_m$, wenn es mindestens eine Abhängigkeit zwischen einem Paar ihrer Instanzen gibt. Ein zusätzlicher *Distanzvektor* $\vec{v}$ als Index gibt die Richtung der Abhängigkeit an. S_l wird als **Source-Anweisung**, S_m als **Sink-Anweisung** bezeichnet. Als **Cross-Abhängigkeit** bezeichnet man ein Datenabhängigkeit, die zwischen zwei Anweisungsinstanzen existiert, die in verschiedenen Iterationen ausgeführt werden.

Durch die Transformation von normalem Code in Sparsecode werden viele Anweisungsinstanzen nicht ausgeführt (siehe Abschnitt 3.3). Deshalb vermindert sich die Zahl der Datenabhängigkeiten, da eine Abhängigkeit mit einer Source- oder Sink-Anweisungsinstanz, die nicht ausgeführt wird, nicht mehr existiert. Folgendes Beispiel demonstriert dies:

Beispiel 8 Anfangs bestehen zwei Datenabhängigkeiten: $S_1 \delta S_2$ und $S_2 \delta S_3$. Diese verlangen, daß die drei Anweisungen sequentiell ausgeführt werden:

$$\begin{aligned} S_1 :&\quad \mathtt{B(3) = EXPR} \\ S_2 :&\quad \mathtt{ACC = ACC + A(1,3) * B(3)} \\ S_3 :&\quad \mathtt{C(1) = 6.0 * ACC} \end{aligned}$$

Würde man dies nun in Sparsecode umwandeln, so würde die Anweisung S_2 nur ausgeführt werden, wenn $(1,3) \in E_A$ wäre. Sollte hingegen $(1,3) \notin E_A$ sein, dann könnte man diese Anweisung entfernen und die Abhängigkeitskette $S_1 \rightarrow S_2 \rightarrow S_3$ wäre durchbrochen, S_1 und S_3 könnten also parallel ausgeführt werden. Mit den bekannten `COBEGIN/COEND`-Anweisungen könnte man dies beispielsweise folgendermaßen beschreiben:

```
      IF (1,3) ∈ E_A THEN
S1 :    B(3) = EXPR
S2 :    ACC = ACC + A'[σ_A(1,3)] * B(3)
S3 :    C(1) = 6.0 * ACC
      ELSE
        COBEGIN
S1 :      B(3) = EXPR
S3 :      C(1) = 6.0 * ACC
        COEND
      ENDIF
```

Allerdings kann man durch diese Art der Transformation keinen großen Geschwindigkeitsvorteil erreichen. Deshalb soll im folgenden eine andere Methode vorgestellt werden, die mit solchen Abhängigkeiten effizienter umgeht.

4.2.2 Schleifensynchronisation

Damit eine Schleife parallel ausgeführt wird, muß man eine andere Anweisung als `DO` verwenden, da diese bekanntermaßen eine sequentielle Ausführung der Schleife bewirkt. Deshalb verwendet man eine `DOALL`-Schleife, wenn die einzelnen Iterationen untereinander unabhängig sind und demzufolge parallel ausgeführt werden können, und eine `DOACROSS`-Schleife, wenn Cross-Abhängigkeiten existieren.

Zur Schleifensynchronisation wird in [BW95] eine Methode verwendet, die *Random Synchronization* bezeichnet wird und ähnlich dem Semaphorenkonzept mancher Betriebssysteme arbeitet (genaueres siehe [Wol89]). Dabei verwendet man zwei Primitiven **post** und **wait**:

- **Post**, eine nicht-blockierende Anweisung, setzt ein eindeutig festgelegtes Bit in einem Bitvektor.
- **Wait** wartet solange, bis ein festgelegtes Bit gesetzt ist. Dabei können verschiedene **wait**-Instanzen auf dasselbe Bit warten.

Hinter jeder Source-Anweisung wird nun ein **post** eingefügt, wogegen **wait** vor jede Sink-Anweisung gestellt wird. Als Beispiel diene folgendes Codefragment, wobei die Abhängigkeiten $S_1 \delta_= S_2$ und $S_2 \delta_< S_1$ existieren.

Beispiel 9

$$
\begin{array}{ll}
 & \mathtt{DO\ I = 2,\ N} \\
S_1: & \quad \mathtt{C(I) = B(I-1)} \\
S_2: & \quad \mathtt{B(I) = B(I) + A(I,I) * C(I)} \\
 & \mathtt{ENDDO}
\end{array}
\quad \rightarrow \quad
\begin{array}{l}
\mathtt{DOACROSS\ I = 2,\ N} \\
\quad \mathbf{wait}(\mathtt{BSYNC, I-1}) \\
\quad \mathtt{C(I) = B(I-1)} \\
\quad \mathtt{B(I) = B(I) + A(I,I) * C(I)} \\
\quad \mathbf{post}(\mathtt{BSYNC, I}) \\
\mathtt{ENDDOACROSS}
\end{array}
$$

Tabelle 1[6] zeigt die Ausführung beider Schleifen für $N = 4$[7]. Wie man sieht, führt der zusätzliche Overhead sogar zu einer längeren Laufzeit des Programms.

Repräsentiert jetzt aber das Array `A` eine dünne Matrix A, so muß etwa die Anweisung $S_2(3)$ nicht ausgeführt werden, wenn $a_{33} = 0$ gilt. Damit wäre dann die Abhängigkeitskette $S_1(3) \rightarrow_= S_2(3) \rightarrow_< S_1(4)$ zerbrochen und die Iterationen $I = 3$ und $I = 4$ unabhängig. Allgemein bedeutet dies: Falls $(I, I) \notin E_A$, dann

[6] Die Abkürzungen pBi und wBi stehen für **post**`(BSYNC,i)` und **wait**`(BSYNC, i)`.

[7] **wait**(BSYNC,1) blockiert nicht, da $I = 1$ undefiniert ist.

Tabelle 1: Ausführungsreihenfolge ohne Sparsetransformation

	sequentiell			parallel		
I:	2	3	4	2	3	4
	$S_1(2)$			wB1	wB2	wB3
	$S_2(2)$			$S_1(2)$	.	.
		$S_1(3)$		$S_2(2)$	.	.
		$S_2(3)$		pB2	.	.
			$S_1(4)$		$S_1(3)$	.
			$S_2(4)$		$S_2(3)$	.
					pB3	.
						$S_1(4)$
						$S_2(4)$
						pB4

muß die Anweisung S_2 nicht ausgeführt werden. Dies gilt aber nicht nur für S_2, sondern auch für die nachfolgende **post**-Anweisung. Das führt zu folgendem Code:

```
      DOACROSS I = 2, N
        IF (I > 2) THEN
          IF (I − 1, I − 1) ∈ E_A
            wait(BSYNC, I − 1)
          ENDIF
        ENDIF
S1 :    C(I) = B(I − 1)
        IF (I, I) ∈ E_A THEN
S2 :      B(I) = B(I) + A'[σ_A(I, I)] ∗ C(I)
          post(BSYNC, I)
        ENDIF
      ENDDOACROSS
```

Tabelle 2 demonstriert die Ausführung dieser Schleife für den Fall, daß `N = 4` und $a_{33} \notin E_A$.

Nur durch das Setzen eines Wertes der Matrix auf Null konnte in obigem Beispiel eine Verkürzung der Laufzeit des Programms erreicht werden. Dies zeigt das Potential, das in diesem Ansatz steckt. Weiterhin ist noch eine Vielzahl von Optimierungen denkbar, eine Auswahl davon ist in [BW95] zu finden.

Tabelle 2: Ausführungsreihenfolge

	$(3,3) \in E_A$			$(3,3) \notin E_A$		
I:	2	3	4	2	3	4
	$S_1(2)$	wB2	wB3	$S_1(2)$	wB2	$S_1(4)$
	$S_2(2)$	.	.	$S_2(2)$	.	$S_2(4)$
	pB2	.	.	pB2	.	pB4
		$S_1(3)$	.		$S_1(3)$	
		$S_2(3)$	.			
		pB3	.			
			$S_1(4)$			
			$S_2(4)$			
			pB4			

5 Zusammenfassung

In den vorherigen Abschnitten wurde ein Compiler vorgestellt, der normalen, *dichten* Code in Sparsecode transformiert, sofern ihm ein Mindestmaß an Strukturinformationen über die benutzten dünnen Matrizen zur Verfügung steht. Es zeigte sich, daß sowohl in der automatischen Wahl von geeigneten Speicherschemata als auch in der möglichen Parallelisierung ein enormes Potential steckt.

Allerdings wurden bisher nur statische dünne Matrizen berücksichtigt. Das ist in den meisten Programmen nicht unbedingt der Fall. Oft werden in dünnen Matrizen zur Laufzeit Null-Einträge zu Einträgen und umgekehrt. Dies erschwert sowohl die Wahl eines geeigneten Speicherschemas als auch die Parallelisierung des generierten Sparsecodes. Genaueres hierzu kann man etwa in [BW95] finden.

Trotzdem bietet die Anwendung eines Sparse Compilers vor allem in Hinblick auf die Portierung von von Sourcecode auf andere Rechner oder die Verwendung von anders strukturierten dünnen Matrizen nicht zu vernachlässigende Vorteile. Der Entwicklungsaufwand sollte sich dadurch wesentlich verringern lassen.

Literatur

[Ban88] U. Banerjee. *Loop Transformations for Restructuring Compilers.* Kluwer, Boston, 1988.

[Ban93] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer, Boston, 1993.

[BBW] A. J. C. Bik, P. J. H. Brinkhaus, and H. A. G. Wijshoff. *The Sparce Compiler MT1: A Reference Guide.*

[Bik96] A. J. C. Bik. *Compiler Support for Sparce Matrix Computations.* PhD thesis, Department of Computer Science, University of Leiden, 1996.

[BW93] A. J. C. Bik and H. A. G. Wijshoff, editors. *Compilation techniques for sparse matrix computations*, Proceedings of the International Conference on Supercomputing, 1993.

[BW94] A. J. C. Bik and H. A. G. Wijshoff. On automatic data structure selection and code generation for sparse computations. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Lecture Notes in Computer Science*, number 768, pages 57–75. Springer-Verlag, Berlin/New York, 1994.

[BW95] A. J. C. Bik and H. A. G. Wijshoff. Advanced compiler optimizations for sparse computations. *Journal of Parallel and Distributed Computing*, 31:14–24, 1995.

[BW96a] A. J. C. Bik and H. A. G. Wijshoff. Automatic data structure selection and transformation for sparse matrix computations. *IEEE Transactions on Parallel and Distributed Systems*, 7(2):109–126, 1996.

[BW96b] A. J. C. Bik and H. A. G. Wijshoff, editors. *Simple Qualitative Experiments with a Sparse Compiler*, Proceedings of the 9th International Workshop on Languages and Compilers for Parallel Computing, 1996.

[BW96c] A. J. C. Bik and H. A. G. Wijshoff. The use of iteration space partitioning to construct representative simple sections. *Journal of Parallel and Distributed Computing*, 34:94–110, 1996.

[Kuc78] D. J. Kuck. *The Structure of Computers and Computations*, volume 1. Wiley, New York, 1978.

[Pis84] S. Pissanetsky. *Sparse Matrix Technology.* Academic Press, London, 1984.

[Pol88] C. D. Polychronoupolos. *Parallel Programming and Compilers.* Kluwer, Boston, 1988.

[PW86] D. A. Padua and M. J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29:1184–1201, 1986.

[Qui87] M. J. Quinn. *Designing Efficient Algorithms for Parallel Computers.* McGraw-Hill, New York, 1987.

[Wol89] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers.* Pitman, London, 1989.