配列データに関する依存解析のための整数解探索

峰尾 昌明* 上原 哲太郎† 齋藤 彰一† 國枝 義敏†
*和歌山大学大学院システム工学研究科 †和歌山大学システム工学部

概要 並列化を行う際にプログラムの中でもっとも効果が望めるのはループである。ループを並列化する際に単純に並列化を行うとデータの参照順が変わってしまう可能性がある。そのため、間違った実行結果になることがある。このような並列化が不可能なループを特定するためにはデータ依存解析が必須となる。このデータ依存解析手法には、既にいくつかの手法が提案されている。ここではその内、GCD テスト、Banerjee テストを実装し、新たに Banerjee テストを拡張し、依存距離と依存方向を求めること、そして第 3 の手法として線形計画法を応用することを提案し、これらの手法について考察する。

Integer Solution Search for Data Dependence Analysis on Array References

Masaaki MINEO* Tetsutaro UEHARA† Shoichi SAITO† Yoshitoshi KUNIEDA†
*Graduate School of Systems Engineering,
†Faculty of Systems Engineering,
Wakayama University

Abstract *The first target of parallelization is a loop structure. However, when executing, the order of data access differs between parallel processing and an original sequential processing. Thus the execution result may change between them. Therefore automatic parallelizng compilers analyse the data access patern in loops. This analysis is called Data Dependence Analysis. Several analysing methods are already proposed for this data dependence analysis. GCD test, Banerjee test in those are implemented in our parallelizing compiler. This paper disserts these two and then newly proposes the extension of Banerjee test and the application of a linear programming as the 3rd method and studies all of them.*

1 はじめに

一般にプログラムの中でもっとも並列化の効果があるのはループである。しかし、ループを並列実行することで元のプログラムと比べてデータへのアクセスの順序が変わり、ひいては実行結果が変わってしまうことがある。これを防ぐためデータ依存解析が必要となる。しかし現状では、提案されている一種類の解析法でループ内のデータ依存を完全に解析することは難しい。そのため、簡単なテストを複数行うことで解析の精度の向上と解析時間の短縮を図る。本論文では、Banerjee テストと GCD テストの組み合わせを示し、Banerjee テストを拡張する方法、そして第 3 の手法として線形計画法を応用することを提案し、これらの手法について考察する。

2 配列データの依存解析

Fortran の do ループ構造などを並列化するためには、ループ内のデータ依存解析が必要となる[1]。例えば、4 つの文 (A,B,C,D とする) を 2 回繰り返す場合、逐次処理する場合は図 1、各ループの繰り返しを並列処理する場合は図 2 の順となる。逐次処理と並列処理で文の実行順序が異なるため、逐次処理の結果と並列処理の結果が異なる可能性がある。言い換えれば、逐次処理のデータ参照の順と並列処理のデータ参照の順が同じでないと、元プログラムの意味を変える可能性がある。

データ依存関係とは、この例で示すようなデータ参照関係から生じる実行順序の制約を言う。

A1 → B1 → C1 → D1 → A2 → B2 → C2 → D2

図 1 逐次処理

A1 → B1 → C1 → D1
A2 → B2 → C2 → D2

図 2 並列処理

2.1 アクセスの種別による分類

図 3 の配列 B の各要素の定義と引用の順は逐次処理のときには図 4 のように引用-定義なのに対し、並列処理のときは図 5 のように定義-引用と逆転するため、並列化すると実行結果が変わる。

```
do i = 1, n
    B[i] = A[i] + C[i]     …  S1
    D[i] = B[i+1] - C[i]   …  S2
end do
```

図 3 プログラム例 1

```
B[1] = A[1] + C[1]
D[1] = B[2] - C[1]
B[2] = A[2] + C[2]
D[2] = B[3] - C[2]
   …
```

図 4　プログラム例 1 の逐次処理

```
プロセッサ 1          プロセッサ 2
B[1]=A[1]+C[1]    B[2]=A[2]+C[2]    …
D[1]=B[2]-C[1]    D[2]=B[3]-C[2]
```

図 5　プログラム例 1 のループボディの並列処理

しかし、代入文 S1 と S2 の順を逆にすると、並列化後のデータアクセスの順が引用-定義になる。このように、元のプログラムの文を操作することで並列化が可能となる場合もある。

依存の種類	先のアクセス	→	後のアクセス
フロー依存	定義	→	引用
逆依存	引用	→	定義
出力依存	定義	→	定義
入力依存	引用	→	引用

表 1　依存の種類

このようなデータの依存関係はアクセスの種別によって、次の表 1 のように分類することができる [1]。

2.2　ループの回転による分類

ループの並列化ではループをまたぐ場合も考慮しなくてはならない。異なった繰り返しのループボディ間でのデータ依存を**ループ繰越し依存**という [1]。また、同一の繰り返しのループボディ内のデータ依存を**ループ独立依存**という [1]。

2.3　依存方向と依存距離

二配列の添え字の差を**依存距離**(厳密な定義は文献 [1] 参照) といい、図 6 の配列 A の場合は 4 となる。この例のように依存距離が整定数になれば、この距離ごとにループを分割することで、依存関係を保持したままループを並列化できる。しかし、依存距離に変数が含まれる場合は単純にループを分割することはできない。

依存距離の正負の向きを**依存方向**という [1]。この依存方向がわかることで、2.1 で触れた文の順序交換のによる並列化アルゴリズムを適用できる。

```
do i = 1,n
    A[i+4] = B[i+2] + X … S1
    B[3*i] = A[i] + 3*X … S2
end do
```

図 6　プログラム例 2

3　定式化と依存解析手法

配列データのデータ依存を解析するには、同一配列の二出現が同一要素を参照するか、すなわち、二出現の添え字式がループの回転に伴って同じ値となるかを調べる。

3.1　ディオファンタス方程式

以下、配列データの依存解析問題を定式化する。図 6 の同一配列 B の二出現について図 7 のように添え字式 Se1、Se2 から方程式を作成する。これをディオファンタス方程式という。一般に k 次元の配列の場合は k 個の方程式からなる。このディオファンタス方程式を満たす整数解 i_1、i_2 が存在するかを調べることで、同一要素を参照するかどうかを調べることができる。

$Se1(i) = i + 2$　…$S1$ 側添え字式
$Se2(i) = 3 * i$　…$S2$ 側添え字式
$i_1 + 2 = 3 * i_2$

図 7　線形ディオファンタス方程式

なお、この方程式で元ループの制御変数 i を二出現で i_1、i_2 と別の変数として扱うのは、二出現が独立に、すなわち並列にどのような順序で処理されても検査できるようにするためである。また、i_1、i_2 は元の制御変数の制約を受け継ぐ。すなわち、制御変数 i の上下限式を定義域とする。よって、配列データの依存解析はこの不等式制約下で整数解の存在を調査することであると定式化できる。

3.2　GCD テスト

ディオファンタス方程式における変数の係数の GCD(Greatest Common Divisor:最大公約数) で方程式右辺の定数が割り切れるかどうかによりテストする [2]。図 8 の式 (1) の左辺の係数の最大公約数 $GCD(2,-1,1) = 1$ で、同式の右辺の 2 が割り切れるため、式 (1) には整数解が存在すると判定でき、依存のある可能性があると判る。もし割り切れなければ、整数解が存在しないと判定でき、依存はないと判る。

```
do i = 1, n
    do j = 1, n
        A[2i+j-2] = …
        … = A[i] - B[i]
    end do
end do
```

図 8　プログラム例 3

$$2i_1 - i_2 + j_1 = 2 \qquad (1)$$

ループ内の配列が二次元配列の場合は、各次元ごとに同テストを行いすべての次元において整数解が存在するとき、依存のある可能性があると判る。

ただし、このテストは前項で述べた各変数の定義域をまったく考慮しないため、依存がないことは検査できるが、依存が本当にあるのかどうかを検査することはできない。GCD テストは処理が簡

単なため、高速に解析を行うことができる。そのため、処理が複雑で時間がかかる他のテストを行う前にGCDテストを行い、依存の可能性のある場合を絞り込み、その後、複雑なテストを行うことで、解析処理全体の時間を短縮することができる。

3.3 Banerjeeテスト

不等式テスト[1, 2]とも呼ばれ、ディオファンタス方程式の変数の定義域内で方程式を成り立たせる解が存在するかを調べるテストで、成り立たせる解が無ければ依存がないことがわかる。

```
do i = 0, 100
    do j = 0, 100
        A[i+j+1] = ···
        ··· = A[i+j+3]
    end do
end do
```

図9 プログラム例4

$$i_1 - i_2 + j_1 - j_2 = 2 \tag{2}$$

$$\begin{cases} 0 \le i_1, i_2 \le 100 \\ 0 \le j_1, j_2 \le 100 \end{cases} \tag{3}$$

$$-200 \le i_1 - i_2 + j_1 - j_2 \le 200 \tag{4}$$

図9の各変数の定義域として式(3)を得る。この制約条件の下で、線形ディオファンタス方程式(2)の左辺のとりうる値の最小値と最大値を求めると式(4)を得る。この範囲に式(2)の右辺の値が存在すれば、依存のある可能性がある。逆にこの範囲に右辺の値が存在しなければ、依存は無い。

ループ内の配列が多次元配列の場合は、GCDテストの場合同様に各次元毎での解析を繰り返せば良い。

以上述べてきたBanerjeeテストでは、ディファンタス方程式を実数緩和して判定している。すなわち、ディオファンタス方程式の実数解が存在することは確認できる。その解が整数解であるかどうかまでは判断できないため、依存が無いことは確実にわかるが、依存が本当にあるかどうかまではわからない。

3.4 Banerjee Testの拡張アルゴリズム

本節では，新たにBanerjeeテストを拡張し，2.3節で述べた依存方向，依存距離までも求めてしまうアルゴリズムを提案する。

[依存方向，依存距離を求めるアルゴリズム]

Step 0: 通常のBanerjeeテストを実施。 実数解の存在可能性がないと判定された場合は，終了。このとき，各制御変数の上下限式から得られた不等式 $min_i \le \sum_{j=1}^{i} c_{ij}x_j \le max_i$(c_{ij}, min_i, max_i:整定数) それぞれについて，変数消去し，整理すると次のようになる。
$min'_i \le x_i \le max'_i$ (min'_i, max'_i:整定数)

Step 1: 一般に多次元配列の場合，複数個の等式から成る線形ディオファンタス方程式の各式について，$a \times i_1 - a \times i_2 = b$(a,b：整定数)の形のものを探す。もし，あれば元の制御変数 i の依存距離 b/a (これは先にGCDテストを実施することにより，整数値となることが保証される) および依存方向をその結果の正負符号とし，登録。

Step 2: まだ依存距離，依存方向が未定の制御変数が存在する間、以下を繰り返す。未定の制御変数を一つ取り上げ，j とする。

Step 3: 線形ディオファンタス方程式の各式について，j_1, j_2 を含み，かつ，含まれるその他の制御変数の総数が最小の等式を取り出す。

Step 4: 上で選ばれた一つ以上の等式で，$j_1 = J + j_2$ ならびに，$j_2 = J - j_1$ を代入し，それぞれ j_1,j_2 を消去する。J 以外の残った変数について，Step0 で求めた各変数の変域を用い，消去する。

Step 5: 得られた J に関する不等式が複数存在する場合，すべてのandを取る(積集合)。

Step 6: 得られた値域を依存方向および依存距離として登録する。Step 2 へ戻る。

3.5 線形計画法と全探索によるテスト

線形計画法のいくつかある解法の内の1つのシンプレックス法を用いる新しい手法を提案する。

[シンプレックス法を用いたアルゴリズム]

Step 0: 与えられた制約下で線形ディオファンタス方程式を解く。凸空間であるn次元(n: 式内の変数の数)の解空間の頂点の座標を全て求める。この頂点の中からもっとも距離の離れた2点間の距離 (d_{max}) を求めておく。

Step 1: 解空間の全ての頂点に対して近傍の整数格子点を求め、候補リストに追加する。

Step 2: リスト内の全候補がチェックされるまで以下の操作を繰り返す。すべてチェックされると依存が無いことが分かる。

Step 3: リストの中から1点を選び($\mathbf{c}$ とする)、方程式、不等式に代入して整数解があるかどうかを調べる。

Step 4: 整数解が見つかれば、依存があることがわかる。見つからなければ、"次"の近傍点 $\mathbf{c}'$ を求める。

$$\{\ \mathbf{c}' | \forall i, \mathbf{c}'_i = \mathbf{c}_i \pm 1 and \mathbf{c}'_j = \mathbf{c}_j (j = 1, n; j \ne i)\}$$

Step 5: 求めた $\mathbf{c}'$ が元の頂点より d_{max} 以上離れているときは除外し、そうでなければ候補リストに追加する。その際に既にリスト内にその頂点の座標が存在する場合、追加しない。Step 3 へ戻る。

4 考察

文献[3]の中で、Banerjeeテストと Omegaテスト[5]の性能比較を行っている。ベンチマークとして一般的なPerfect Benchmarksに対してBanerjeeテストとOmegaテストはほぼ同じ精度であることが示されている。具体的にはBanerjeeテストはOmegaテストより依存があることを判定する能力は低いことが示されている。しかし、実際に最適化で必要なのは「依存があることが分かる」ことではなく「依存がないことが分かる」ことである。よって、これは重大な欠点とは言えない。このことから通常、一般的なプログラムに対する依存解析ではBanerjeeテストより処理の軽いGCDテストを行った後、Banerjeeテストを行えば十分であると言える。

同じ文献で科学技術演算等の高度な数値演算に対してはBanerjeeテストはOmegaテストに劣ることが示されている。しかし、Omegaテストにかかる時間はBanerjeeテストに比べて65〜80倍に長いことも同時に示されている。したがって、Omegaテストは解析にかかる時間が長く、実装が困難であるため、高度な数値演算を行うプログラムを対象としたコンパイラに対してでなければ、実装する必要はないと考える。さらに、Omegaテストより高速で、Banerjeeテストより高性能な依存解析手法が望まれる。そこで我々は、3.5節で線形計画法と全探索によるテストを提案した。

3.4節で述べたアルゴリズムでは，線形ディオファンタス方程式から依存方向，依存距離を直接求めている。上述のとおり文献[3]によれば，Banerjeeテストで，十分な判定が可能であることを実際に確認している。換言すれば，実際のプログラム中に出現する添え字式は単純な形が多いと推測できる。よって，多くの場合に，特に，Step 1の単純な調査でも有効にふるい分けられることが期待できると思われる。

同じくこのアルゴリズムのStep 3では，着目する変数以外の変数が少ないことが望ましい。従って，与えられた線形ディオファンタス方程式で，i_1とi_2，j_1とj_2，という変数のペアだけを残すように，連立一次方程式の変数消去のアルゴリズム(後退代入)を可能な限り実施しておく前処理が有効であると考えられる。この前処理は，通常のBanerjeeテストにおいても，有効であるので，等式を作成，格納した段階で，実施すべきであると考える。この処理では、凸多面体である解空間のi_1-i_2平面上への写像を求めていると言える。

3.4節のアルゴリズムは，依存距離をも同時に求めるため，前記のように処理をすすめた。しかし，もし依存方向のみを求めるだけで，良い場合には，より単純な次のアルゴリズムが考えられる。このアルゴリズムにおいても，上述の前処理としての変数消去は通常、総解析時間を短縮できることから，有効であると思われる。

[依存方向を求めるBanerjeeテスト]

Step 1: 調べたい依存方向を示す不等式($i_1 < i_2$または$i_1 > i_2$)または，等式($i_1 = i_2$)を，与えられた線形ディオファンタス方程式および制御変数の上下限式からの制約の不等式群に，$i_1 - i_2 < 0, = 0, > 0$の形に整理して追加する。

Step 2: 通常のBanerjeeテストを実施。解がないと判定されれば，当該方向に解がないと言える。逆に，解がある可能性があると判定されれば，その方向を登録する。

Step 3: すべての制御変数ごとに，Step 1の三方向$(<, =, >)$を組み合わせ，すべての組み合わせがすむまで，Step 1へ戻る。

5 おわりに

配列データの依存解析では本研究により、提案しているGCDテストとBanerjeeテストを併用することで、一般的なプログラムに対しては十分な精度があると考える。しかしながら、高度な数値演算に対してはOmegaテストに比べて劣る。そこでOmegaテストに近い解析精度の線形計画法と全探索によるテストを提案した。

また、Banerjeeテストの枠組み内で依存方向と依存距離を求めるための2つの手法を提案した。

謝辞

本研究は一部日本学術振興会の未来開拓事業(No.00505)と科学研究費(No.13480085)のサポートを受けている。

参考文献

[1] 中田育男,"コンパイラの構成と最適化,"朝倉書店(1999).

[2] UTPAL BANERJEE,"DEPENDENCE ANALYSIS," KLUWER ACADEMIC PUBLISHERS(1997).

[3] Kleanthis Psarris and Konstantinos Kyriakopoulos,"Data Dependence Testing in Practice," 1999 International Conference on Parallel Architectures and Compilation Techniques (Cat. No.PR00425) , Page: 264-73.

[4] Alexander Schrijver,"Theory of Linear and Integer Programing," John Wiley & Sons(1987).

[5] W.Pugh and D.Wonnacott, "Eliminating False Data Dependences using the Omega Test," Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation, San Francisco, CA(1992).