ハイパフォーマンス
コンピューティング 88－1
（２００１．１０．２６）

自動並列化コンパイラ MIRAI における 配列データ依存解析部の実現方式

北村 隆光 * 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 並列化の主な対象となるのは逐次プログラム中のループ部分である。しかし、逐次プログラムのループ処理を単純に並列化してしまうと、データへのアクセス順序が本来の逐次処理とは変化してしまう場合があり、本来の実行結果と異なる結果となる場合がある。したがって、逐次プログラムのループ部を並列化するにはデータ依存の有無を判定する必要がある。データ依存を解析する手法として、既にいくつかの手法が考案されている。本依存解析部ではその内、GCD テスト、Banerjee テストを実装し、また新たな手法として、線形計画法と全探索によるテストを実装したので報告する。

The Realization System of Data Dependence Analysis on Array References for the Automatic Parallelizing Compiler,MIRAI

Takamitsu KITAMURA* Masaaki MINEO* Tetsutaro UEHARA†
Shoichi SAITO† Yoshitoshi KUNIEDA†

*Graduate School of Systems Engineering,
†Faculty of Systems Engineering,
Wakayama University

Abstract *Generally loops in a given program are the target for almost all the most parallelization techniques. However, in some cases,when loops are parallelized without dependence analysis, the result of the given program may be changed comparing with an original execution result. Therefore, in order to parallelize the loops of a given program, it is necessary to analyze the existence of data dependence. As a method of analyzing data dependence, some techniques are already devised. In this dependence analysis module described here,GCD test and Banerjee test are adopted. And,the test of Linear Programming and exhaustive search was implement as a new technique,which this paper reports.*

1 はじめに

近年汎用コンピュータを複数台接続したクラスタシステムによる並列計算が身近になりつつある。こうした並列計算機の使用に際し、並列プログラムをユーザが記述する場合、その環境に応じたプログラムを書くことで、パフォーマンスを向上させることが可能である。しかし、その並列処理を意識したコーディングには多大な時間と労力が必要となる。よって、ユーザが並列処理を意識せずプログラムを記述したのち、自動並列化コンパイラによって自動的に並列化するというアプローチが有効となる。従来、並列化の主な対象となってきたものはループである。これはループには多くの並列性が期待できること、ループの処理時間がプログラムの実行時間に占める割合が大きいこと、解析が比較的行いやすいことなどの理由による。一般にループを複数台の計算機で並列実行することにより実行時間を大幅に短縮することが期待できる。しかし、中には並列実行することでデータへのアクセス順序が異なり、実行結果が変わってしまう場合がある。このような並列化不可能なループを特定するために、依存解析を行う必要がある。MIRAI コンパイラの配列データ依存解析部では、従来の GCD テストと Banerjee テストをまず実装している [3]。今回新たに、これらに加え、新しく設計、実装した線形計画法と全探索によるテストについて述べる。

2 依存解析手法

この章では、ループの繰り返し全体に渡り、特に困難な配列に関するデータ依存 [3] の有無を解析する手法について詳解する。

2.1 ディオファンタス方程式

図 1 のプログラム例 1 では同一の繰り返しで S1 の配列 A と S2 の配列 A の添え字式が同じ値になることはないので、フロー依存は生じない。ループ繰越し依存の存在を解析するには、S1 の配列 A の添え字式と S2 の配列 A の添え字式が独立した

```
do i = 1,n
      A[2*i+2] = B[i] + C[i]   :S1
      C[i]   = A[3*i] + B[i]   :S2
end do
```

図 1　プログラム例 1

制御変数 i の値をとった時に、添え字式が等しくなるかどうかを調べればよい。よって、下の式 (1) の方程式を満たす整数解 $\mathbf{i_1}$、$\mathbf{i_2}$ が存在するかという問題に定式化できる。

$$\mathbf{2i_1 + 2 = 3i_2} \tag{1}$$

式 (1) を**線形ディオファンタス方程式**[1](以後ディオファンタス方程式と略記する) と呼ぶ。また、多次元の配列の場合は各次元ごとにディオファンタス方程式を作成し、解析する。このディオファンタス方程式の解の存在を判別する手法として、以下に示す GCD テスト [1, 2]、Banerjee テスト [1, 2, 4] などがある。

2.2　GCD テスト

以下では式 (1) を変形し、変数を含む項を左辺に、定数は右辺に移項し整理しているものとする。GCD テストでは、ディオファンタス方程式の各係数が整数であることを利用して、整数解が存在するかどうかを調べる。左辺の変数の係数の最大公約数 GCD(Greatest Common Divisor) で右辺の定数が割り切れれば整数解が存在し、依存の可能性がある。逆に、割り切れなければ整数解は存在せず、依存も存在しない。

$$\mathbf{aX + bY = c} \tag{2}$$

整数: a , b , c
変数 (整数): X , Y

ここで、式 (2) の係数 a,b の最大公約数 GCD(a,b) = d とする。d で式 (2) の両辺を割ると、式 (3) を得る。

$$\frac{a}{d}X + \frac{b}{d}Y = \frac{c}{d} \tag{3}$$

GCD(a,b) = d より、$\frac{a}{d}$ と $\frac{b}{d}$ は必ず整数となり、X,Y もまた整数であるので、式 (3) の左辺全体の値は整数となる。右辺の $\frac{c}{d}$ が割り切れない場合、$\frac{c}{d}$ は整数にならないので、左辺≠右辺となり、この方程式を満たす整数解 X,Y が存在しないことから、依存はないと判定できる。逆に $\frac{c}{d}$ が割り切れる場合は、$\frac{c}{d}$ は整数となり、左辺＝右辺となる可能性があるので、この方程式を満たす整数解 X,Y が存在しうる。よって、依存の可能性があると判定できる。

実際の例で見ると、図 2 のプログラム例 2 の場合、ディオファンタス方程式は式 (4) のようになる。式 (4) の係数の最大公約数 GCD(3,-9,3) は 3 であるので、同式の右辺の 3 をこの最大公約数で割ると 1 となり、割り切れる。よって、依存の可能性があることが判る。

```
do i = 1,n
   do j = 1,n
      A[3*i+3*j+2] =  ···
           ···       = A[9*i+5]
   end do
end do
```

図 2　プログラム例 2

$$\mathbf{3i_1 - 9i_2 + 3j_1 = 3} \tag{4}$$

また、図 3 のプログラム例 3 のようにループ内の配列が二次元配列の場合は、各次元ごとにディオファンタス方程式を作成し、それぞれにテストを行うことで解析が可能である。

```
do i = 1,n
   do j = 1,n
      A[2*i+2,2*j+1] =  ···
         ···     = A[2*i-4,2*j+2]
   end do
end do
```

図 3　プログラム例 3

$$\mathbf{2i_1 - 2i_2 = -6} \tag{5}$$

$$\mathbf{2j_1 - 2j_2 = 1} \tag{6}$$

よって、式 (5)、式 (6) の二つのディオファンタス方程式に対して GCD テストを行えばよい。まず、式 (5) は係数の最大公約数 GCD(2,-2)=2 で右辺の定数項-6 が割り切れるので、依存の可能性がある。式 (6) は係数の最大公約数 GCD(2,-2)=2 で右辺の定数項 1 が割り切れないので依存が無い。したがって、二次元配列上の同一データを参照することはないので、プログラム例 3 の配列 A のデータ参照に依存はないことがわかる。しかし、この

GCD テストは変数の変域を考慮しないため、本当に依存があるかどうかまでは判定できない。当然、依存の可能性ありと判定されても、実際には依存が無い場合も考えられる。したがって GCD テストの結果、依存の可能性が出た場合は、続けて他の解析テストを実行する必要がある。

GCD テストでは解の存在を厳密には判定できない。しかし、GCD テストは、Banerjee テストを含めた他の解析テストに比べ、実行コストが小さい利点がある。したがって、GCD テストを実行した結果、依存の可能性が出た配列参照に対してのみ続けて解析することで、解析時間全体の短縮化が期待できる。

2.3 Banerjee テスト

Banerjee テストでは、ディオファンタス方程式の変数の変域内に方程式を成り立たせる解が存在するかどうかを調べる。ここで、式 (7) のディオファンタス方程式があり、その変数 X,Y の変域を式 (8) とする。

$$\mathbf{aX - bY = c} \tag{7}$$

$$\mathbf{n \leq X, Y \leq m} \tag{8}$$

整定数: a , b , c , n , m
(a,b,c は正とする)
変数 (整数): X , Y

まず、式 (7) のディオファンタス方程式の左辺が取り得る最大値・最小値を各変数の変域から求める。最大値を求めるためには、係数が正の場合は変域の最大値を、負の場合には最小値をそれぞれ用いる。逆に最小値を求めるためには、係数が正の場合は変域の最小値を、負の場合には最大値をそれぞれ用いる。このようにして、ディオファンタス方程式の左辺の最大値・最小値が、式 (9) のように求まる。

$$\mathbf{na - mb \leq aX - bY \leq ma - nb} \tag{9}$$

式 (7) の右辺の定数 c が、この式 (9) の範囲内に存在すれば、依存の可能性があり、そうでなければ依存の無いことがわかる。
実際の例で見てみると、図 4 プログラム例 4 の場合、ディオファンタス方程式は式 (10) となる。また、do ループの初期値・終値から、各変数の変域は式 (11) となるので、この条件下で左辺の最大値・最小値を求めると式 (12) となる。この変域の範囲内に式 (10) の右辺の値が存在する。したがって、この例の場合、依存の可能性がある。

逆に、この範囲内に右辺の値が存在しない場合は、依存はない。また、ループ内の配列が二次元配列の場合は、GCD テストの場合と同じように、各次元ごとにテストを行うことで、依存の有無を判定することが可能である。以上述べてきた Banerjee テストは、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が、その解が整数解かどうかまでは判定することはできない。すなわち、このテストも GCD テストと同じく、依存がない場合は確実に判定できるが、依存があるかどうかは、その可能性があるということまでしか判定できない。

```
do i = 0,100
   do j = 0,100
      A[3*i+j-10] =  …
           …      = A[i-4*j+5]
   end do
end do
```

図 4　プログラム例 4

$$\mathbf{3i_1 + j_1 - i_2 + 4j_2 = 15} \tag{10}$$

$$\begin{cases} \mathbf{0 \leq i_1, i_2 \leq 100} \\ \mathbf{0 \leq j_1, j_2 \leq 100} \end{cases} \tag{11}$$

$$\mathbf{-100 \leq 3i_1 + j_1 - i_2 + 4j_2 \leq 800} \tag{12}$$

3　線形計画法と全探索によるテスト

2.1 節で述べたとおり、配列データの依存解析はディオファンタス方程式の整数解の存在判定問題に帰着できる。そこで我々は、線形計画法を利用した新しい解析手法を提案する。線形計画法 (Linear Programming) とは、複数の未知数に対して決められた制約条件の中で、ある関数 (目的関数) の最大値や最小値を求める最適化手法の 1 つである。線形計画法の解法には既にいくつかの手法が提案されている。ここではその内の 1 つであるシンプレックス法を用いた。

3.1 概略アルゴリズム

Step 1: ディオファンタス方程式を立て、与えられた制御変数に関する制約下で、シンプレックス法を用いて解空間のいずれかの頂点の座標を求め、その座標の近傍の整数格子点を探す。

Step 2: 求めた整数格子点から距離 1 の点を求め、変数の制約内なら、順にディオファンタス方程式に代入し、整数解があるかどうかを調べる。調査済みの点にはフラグを立てておく。

Step 3: 整数解が見つかれば、依存関係があるこ

とがわかる。見つからなければ、解空間の他の頂点の座標を求め、Step 2 へ戻る。これ以上求めることが可能な頂点が無い場合は Step 4 へ進む。

Step 4: 制約内のすべての座標をチェックす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関係が無いことが分かる。

Step 5: これまでに求めた解空間の頂点から、前回調査した距離＋1の整数格子点をチェックする。

Step 6: 整数解が見つからない場合は、次の頂点に移動する。この手続きを解空間内にある可能性のある格子点すべてを網羅するまで繰り返す。Step 4 に戻る。

この線形計画法と全探索のテストでは、GCD テストや Banerjee テストとは違い、ディオファンタス方程式の整数解の厳密な存在判定をおこなう。それにより依存の有無を確実に判定することが可能である。しかしながら、解空間中の整数格子点を全探索する必要があるため、膨大な解空間を探索する場合、例えば、整数解が探索を始めた頂点から離れていた場合や、整数解が存在せず、すべての整数格子点を調べる必要がある場合においては、実行コストが大きくなってしまう。

3.2 シンプレックス法

シンプレックス法は線形計画問題に対して比較的効率よく、解の存在する端点を探し出すアルゴリズムである。このアルゴリズムは、解が存在するか否かをはじめに判定する過程、および解空間が有界か否かを判定する過程も含むように工夫されており、収束性も保証されていることから、優れた一般性を持っているといえる。シンプレックス法は以下のようなアルゴリズムである。

制約式として式 (13)、目的関数として式 (14) が与えられたとき、不等号を持つ制約式にスラック (slack) 変数を入れ、等式化することで連立方程式を作成する。ただし、制約式が等式を含む場合には、等式を他の制約式に代入することで、変数の数を減らし、また、(左辺 $\geq$ 右辺) の形の不等式を含む場合には、この不等式の両辺に -1 をかけることで不等号の向きを統一してから、制約式に追加しなくてはならない。

この連立方程式から (15) の様な係数行列が得られる。この行列上の $\mathbf{x_1}$〜$\mathbf{x_n}$ の係数が 1 になるように掃き出し演算を行えばよい。しかし、完全に等式の方程式を解くわけではなく、制約条件を満たし、目的関数が最大（または最小）になるような掃き出し演算を行わなければならない。以下にそのアルゴリズムを示す。

$$\begin{cases} \mathbf{a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \leq b_1} \\ \mathbf{a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \leq b_2} \\ \quad \vdots \qquad \vdots \qquad\qquad \vdots \\ \mathbf{a_{m1}x_1 + a_{11}x_2 + \cdots + a_{mn}x_n \leq b_m} \end{cases} \tag{13}$$

$$\mathbf{z = c_1x_1 + c_2x_2 + \cdots + c_nx_n} \tag{14}$$

$$\begin{pmatrix} \mathbf{a_{11}} & \mathbf{a_{12}\cdots a_{1n}} & \mathbf{1} & & & \mathbf{b_1} \\ \mathbf{a_{21}} & \mathbf{a_{22}\cdots a_{2n}} & & \mathbf{1} & & \mathbf{b_2} \\ \vdots & \vdots & \vdots & & \ddots & \vdots \\ \mathbf{a_{m1}} & \mathbf{a_{m2}\cdots a_{mn}} & & & \mathbf{1} & \mathbf{b_m} \\ \mathbf{-c_1} & \mathbf{-c_2\cdots -c_n} & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} \end{pmatrix} \tag{15}$$

Step 1: 列選択：係数行列の最下行の中から最小なものがある列 y を探す。

Step 2: 最小値 ≥ 0 なら終了。この条件は最下行の係数がすべて正になったことを意味し、これ以上掃き出しを行っても目的関数の値は増加しないことを意味している。

Step 3: 行選択：Step 1 で求めた y 列にある各行の要素で各行の右端要素を割ったものが最小となる行 x を探す。

Step 4: x 行 y 列をピポットにして掃き出し演算を行う。Step 1 へ戻る。

変数 $\mathbf{x_1}$〜$\mathbf{x_n}$ の係数が 1 になっている行を右にたどった $\mathbf{b_1}$〜$\mathbf{b_m}$ にそれぞれの最適解が得られる。それが解空間の頂点に対応する。また、目的関数の最適解は右端最下段に得られる。

実際の例で見てみると、図 5 プログラム例 5 の場合、ディオファンタス方程式は式 (16) となる。また、do ループの初期値・終端値から、各変数

```
do i = 0,10
   do j = 0,20
      A[2*i+j]  =  …
           …    = A[4*i-3*j+5]
   end do
end do
```

図 5　プログラム例 5

の変域は式 (17) となるので、以上の条件から (18) のような制約式が得られる。次に制約式のなかで、(左辺 $\geq$ 右辺) の形の不等式を含む式の両辺に -1 を掛け、不等号の向きを統一する。そして、等号を持つ式は他の制約式に代入して、変数の数を 1 つ減らす。この例の場合は変数 j_2 を消去した。次

$$\mathbf{2i_1 + j_1 - 4i_2 + 3j_2 = 5} \tag{16}$$

$$\begin{cases} \mathbf{0 \leq i_1, i_2 \leq 10} \\ \mathbf{0 \leq j_1, j_2 \leq 20} \end{cases} \tag{17}$$

に、不等号を含む式にスラック変数を入れることで等式化する。そして得られた連立方程式と、任意に定めた目的関数：$\mathbf{Z = ax_1 + bx_2 + cx_3}$ から係数行列 (19) を作成する。

$$\begin{cases} \mathbf{0 \leq x_1, x_1 \leq 10} \\ \mathbf{0 \leq x_2, x_2 \leq 20} \\ \mathbf{0 \leq x_3, x_3 \leq 10} \\ \mathbf{0 \leq x_4, x_4 \leq 20} \\ \mathbf{2x_1 + x_2 - 4x_3 + 3x_4 = 5} \end{cases} \tag{18}$$

$$\begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 10 \\ -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 10 \\ 0 & -1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 20 \\ 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 2 & -4 & 3 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 15 \\ 2 & 4 & -3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 6 \\ -a & -b & -c & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & Z \end{pmatrix} \tag{19}$$

a , b , c , d : 整数

このようにして得られた係数行列に対し、先ほど述べた掃き出し演算をおこなうことで、目的関数の最適解 Z が求まり、解空間の頂点の 1 つを探し出すことができる。また、ループ内の配列が多次元配列の場合は、各次元ごとのディオファンタス方程式を作成し、同時に、各方程式に含まれる変数の初期値・終端値を制約式に追加することで多次元であることを意識せずに解析をおこなうことが可能である。

3.3 目的関数

線形計画法と全探索によるテストでは、通常の線形計画法と違い、与えられた目的関数を最大、または最小にする頂点を探すのではなく、解空間にある全ての頂点の座標を求めることが望ましい。これは、最初に発見したある頂点から解空間を全探索するのではなく、ある程度探索して、整数解を発見できなかった場合は、チェックリストから次の候補を選び、その頂点から再び探索する手法をとっているからである。しかし、全ての頂点をシンプレックス法で求めることは現実的でないので、数種類の目的関数を作成し、求めた頂点の近傍の整数格子点をチェックリストに追加する。目的関数の作成は変数が n 個の場合、

$$\begin{cases} \mathbf{Z = x_1, Z = -x_1} \\ \mathbf{Z = x_2, Z = -x_2} \\ \quad \vdots \qquad \vdots \\ \mathbf{Z = x_{n-1}, Z = -x_{n-1}} \\ \mathbf{Z = x_n, Z = -x_n} \end{cases} \tag{20}$$

式 (20) に示す $2n$ 個の目的関数が候補となる。この目的関数を用いることで、各次元の解空間内での各変数の値域を求めることが可能である。これによって整数解になり得る可能性のある全ての格子点を網羅したかを判定しやすくなる。また、これらの目的関数を用いて頂点の座標を求めることで解空間の端点を複数求めることができる。このように複数の頂点から解空間内の整数格子点を全探索することで、解析速度の平均化が可能であると考える。

3.4 実行手順

実際に目的関数を設定し、最適解を求めてみる。プログラム例 6 の場合、ディオファンタス方程式は式 (21) となり、do ループの初期値・終端値から、各変数の変域は式 (22) となる。

```
do i = 1,4
   do j = 1,i+3
      A[3*i-8]  =  …
           …    = A[2*j+3]
   end do
end do
```

図 6　プログラム例 6

$$\mathbf{3i_1 - 2j_2 = 5} \tag{21}$$

$$\begin{cases} \mathbf{1 \leq i_1 \leq 4} \\ \mathbf{2 \leq j_2 \leq i_1 + 3} \end{cases} \tag{22}$$

これらから、3.2 節で述べたように、不等号の向きを統一し、式 (21) のディオファンタス方程式で、等号を含む制約式で変数消去 (例では i_1 を消去) をおこなっておく。ここで、変数の数は 2 なので、3.3 節で述べたように目的関数を作成すると式 (23) となる。得られた目的関数も、同様に式 (21) のディオファンタス方程式で変数消去し、係数行列 (24) を作成する。得られた係数行列に対して掃き出し演算を行うと、最適解として式 (23) の目的関数 (i)(iii) のとき、$\mathbf{i_1 = 4, j_2 = 3.5}$ が得られる。また、式 (23)

$$\begin{cases} \mathbf{Z} = \mathbf{i_1} & (\text{ i }) \\ \mathbf{Z} = -\mathbf{i_1} & (\text{ ii }) \\ \mathbf{Z} = \mathbf{j_2} & (\text{iii}) \\ \mathbf{Z} = -\mathbf{j_2} & (\text{iv}) \end{cases} \tag{23}$$

$$\begin{pmatrix} 1 & 1 & 0 & 0 & 4 \\ -1 & 0 & 1 & 0 & -2.333 \\ \pm 1 (or) \pm 1.5 & 0 & 0 & 0 & 0 \end{pmatrix} \tag{24}$$

の目的関数 (ii)(iv) のとき、$\mathbf{i_1} = \mathbf{2.333}, \mathbf{j_2} = \mathbf{1}$ が得られる。得られた $(\mathbf{i_1}, \mathbf{j_2})$ の近傍点は、それぞれ $(\mathbf{4}, \mathbf{4})$,$(\mathbf{3}, \mathbf{1})$ であるが、$(\mathbf{3}, \mathbf{1})$ から距離 1 の点を求めると、$(\mathbf{3}, \mathbf{2})$ がディオファタス方程式を満たすことから、整数解が存在し、依存があることが判定できる。

4 考察

GCD テストは、解析にかかる時間は短いが、変数の変域を考慮しないため、本当に解が存在するかどうかまでは判定できない。また、Banerjee テストは、変数の変域を考慮してテストを行うので、解が存在することまでは判定できるが、それが整数解かどうかまでは判定できない。したがって、この二つの解析テストだけでは、ディオファンタス方程式における整数解の存在を厳密に判定することは難しいといえる。厳密な解析手法としては、Omega テスト [7] が有名である。

文献 [5] の中で行われている、Banerjee テストと Omega テストの性能比較によると、一般的なプログラムに対して Banerjee テストと Omega テストはほぼ同程度の精度で依存が無いことを判定している。また、依存の有無が判定できない場合が Banejee テストでは 43 %なのに対し、Omega テストでは 27 %である。しかし、実際に最適化で必要なのは「依存が無いことが分かる」ことであるので、一般的なプログラムに対して、Banerjee テストと Omega テストの解析精度はほとんど変わらないことが分かる。逆に高度な数値演算を含むプログラムに対しては Banejee テストは 26 %、Omega テストは 44 %依存の無いことを判定できる。この事から、科学技術演算など、高度な数値演算を含むプログラムに対しては Banerjee テストは Omega テストに劣っていることがわかる。

しかしながら、Omega テストは解析にかかる時間が長く、また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、実装する必要性は高くない。我々は、Omega テストよりも高速で、GCD テスト・Banerjee テストよりも高精度な依存解析手法を開発した。

シンプレックス法を用いて解空間の頂点を求めるには、目的関数を設定する必要がある。3.3 節の方法で作成した目的関数を使用することで、全てではないが、解析するには充分な解空間の頂点を得ることができる。また、全探索する場合に、1 つの頂点から、ある程度候補をチェックし、整数解を見つけられない場合は、次の頂点に移りチェックをすることを繰り返すことで、高速に整数解を発見することが期待できる。

5 おわりに

配列データの依存解析において、一般的なプログラムに対しては、GCD テストと Banerjee テストを組み合わせることで、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 テストに比べて精度は劣る。そこで Omega テストと同じ解析精度を持ちより高速な依存解析手法を考案し、実装した。

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] Hans Zima ／ Barbara Chapman 共著 村岡洋一 訳『スーパーコンパイラ』 (オーム社、1995 年).

[3] Masaaki Mineo,Tetsutaro Uehara,Shoichi Saito and Yoshitoshi Kunieda, “Integer Solution Search for Data Dependencd Analysis on Array References,” Proc. of Int. Conf. on Parallel and Distributed Processing Techniques and Applications(PDPTA’2001), Vol.III,CSREA press,pp.1312-1318(2001.6)

[4] UTPAL BANERJEE : “DEPENDENCE ANALYSIS,” KLUWER ACADEMIC PUBLISHERS (1997).

[5] 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.

[6] Alexander Schrijver,“Theory of Linear and Integer Programing,”John Wiley & Sons (1987).

[7] 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).