ハイパフォーマンス 90－9
コンピューティング
（２００２．５．２７）

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 自動並列化のための配列要素間データ依存を解析する手法が種々提案されている。Mirai コンパイラの依存解析部には、それらの内、GCD テストと Banerjee テストを実装している。しかし、この 2 つの解析手法だけではディオファンタス方程式の整数解の存在判定を厳密に行うことは不可能である。厳密な依存解析手法として Omega テストがあるが、解析にかかる時間が長く、実装が困難である。そこで、我々は厳密な解析を行う新たな手法として、線形計画法と全探索を組みあわせた手法を実装し、性能評価を行った

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 *Various dependence analysis methods of array elements have been proposed to realize automatic parallelization. Among them, GCD test and Banerjee's test are implemented in the phase of dependence analysis on MIRAI compiler. However, these two methods do not have capablities enough to judge the existance of integer solutions of the Diophantine equations exactly. Omega test was proposed as a method to obtain the precise solutions but it takes much time to analize the equations and considered to be complicated to implement. This report describes a new method which introduces Linear Programming and exhaustive search. Its implementation and experimental results are also described.*

1 はじめに

近年、パーソナルコンピュータやワークステーションを、安価な LAN 機器によって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、その実行環境に適したプログラムを記述することで、高いパフォーマンスを得ることが可能である。しかし、その並列処理を意識したプログラミングのためには高い技術力と多大な労力が必要となる。よって、ユーザが逐次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、このようなコンパイラによる自動並列化の主な対象となってきたものはループである。これはループには高い並列性と適度な並列粒度が期待できること、プログラムの実行時間の中でループの実行時間が占める割合が大きいこと、ループの並列実行可能性解析が比較的行いやすいことなどの理由による。自動並列化においては、ループを各繰り返しごとに分割し、複数台の計算機で並列実行することにより実行時間を短縮しようとする。しかしこのような分割を行うことによ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、従来から知られている GCD テストと Banerjee テストを実装している [4]。今回、独自の手法として、線形計画法と全探索のテストを実装し、その性能評価を行った。

以下では、第 2 章で実装の対象となった MIRAI コンパイラについて述べ、第 3 章で依存解析問題の定式化を行う。第 4 章で提案手法のアルゴリズムを紹介し、第 5 章で性能評価、第 6 章で考察を行い、最後に第 7 章でまとめを述べる。

2 自動並列化コンパイラ MIRAI

現在、MIRAI コンパイラは Microsoft Windows2000 上で 同社製 Visual C++ 6.0 を用いて開

発中である。コンパイラは同社の Win32 実行環境上で動作する。MIRAI コンパイラが生成した C 言語のプログラムは、Linux 上に転送され、gcc を用いてコンパイルされ、分散共有メモリソフトウェア Fagus[3] のライブラリを結合して実行される。図 1 は MIRAI コンパイラの処理の概要である。

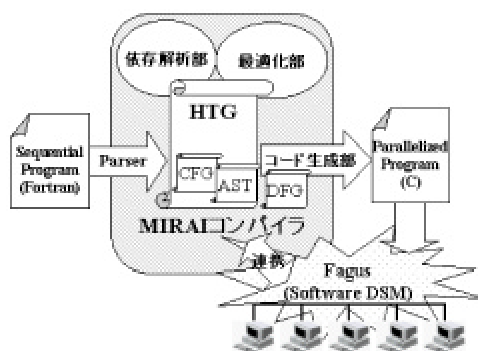

図 1　MIRAI コンパイラ概要

MIRAI コンパイラはパーザにより Fortran のソースプログラムを読み込み、コントロールフローグラフ（Control Flow Graph:CFG）[1] と抽象構文木（Abstract Syntax Tree:AST）[1] を生成する。現バージョンの並列化対象は多重ループに限定しているので、CFG からループ構造を抽出し、階層タスクグラフ（Hierarchical Task Graph:HTG）[9] に変換する。つぎに，後の解析処理などのため、HTG 内の AST を走査し、定数の畳込み [1] などを行う。こうして生成された中間表現について、最適化と依存解析を行い可能な限り並列化可能なループを特定する。さらにデータフローグラフ（Data Flow Graph:DFG）を生成し、並列化可能なループ内に使用されている変数の特定、配列データへのアクセスパターンなどを調べる。最後に HTG、DFG から得られた情報を元に並列化の方針を組み立て、コード生成を行う。

3　ディオファンタス方程式

ループの自動並列化において、データ依存解析の主な対象とされるのは配列要素である。配列要素の解析は、線形ディオファンタス方程式 [1] の解を求める問題に帰着できる。

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

図 2　プログラム例 1

図 2 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く、同一繰り返し内での依存は生じない。ループの繰り返し間にまたがる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、ループの上下限式から求まる変数の定義域内に存在するかという問題に定式化できる。

$$\mathbf{2i_1 + 2 = 3i_2} \qquad (1)$$

$$\mathbf{1 \leq i_1, i_2 \leq n} \qquad (2)$$

式 (1) の線形ディオファンタス方程式を、以後単にディオファンタス方程式と呼ぶ。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、GCD テスト [1, 2]、Banerjee テスト [1, 2, 5] などがある。しかし一般には、この 2 つのテストだけではディオファンタス方程式の整数解の存在判定を行うことは不可能であり、より厳密な手法として Omega Test[8] などが提案されている。しかし Omega Test は実装が複雑で実行時間もかかるため、より容易に実装できる、実行時間も少ない手法が求められている。

4　線形計画法と全探索によるテスト

3 章で述べたとおり、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。我々は、線形計画法と全探索を用いた新しい解析手法を提案する。線形計画法の解法にはシンプレックス法を用いている。アルゴリズムの概略は以下のとおりである。

4.1　概略アルゴリズム

Step 0: 準備として、GCD テストを用いて、与えられたディオファンタス方程式に対し、一般に整数解が存在しうるかどうかしらべる。存在する場合、次に制約条件を加え、Banerjee テストなどを用いて、実数解を持ちうるかどうか判定しておく。ここまでで解を持ち得ないと判定されれば、依存は存在しない。

Step 1: ディオファンタス方程式に対し、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、依存関係がありとして終了する。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、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、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。しかし、一般には探索するべき解空間は多次元空間上の凸多面体であり、空間が大きければ整数解も解空間の頂点の近傍に存在する場合が多い。また空間が小さければ全探索も小さな実行コストで行える。このため、全体として十分に小さな実行コストで実行できる。

4.2 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 (3)、目的関数として式 (4) 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 ≥ 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。

$$\begin{cases} \mathbf{a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \le b_1} \\ \mathbf{a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \le b_2} \\ \quad\vdots \qquad\quad \vdots \qquad\qquad\qquad \vdots \\ \mathbf{a_{m1}x_1 + a_{11}x_2 + \cdots + a_{mn}x_n \le b_m} \end{cases} \tag{3}$$

$$\mathbf{z = c_1x_1 + c_2x_2 + \cdots + c_nx_n} \tag{4}$$

この連立方程式から (5) 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。

$$\begin{pmatrix} \mathbf{a_{11}} & \mathbf{a_{12} \cdots a_{1n}} & \mathbf{1} & & & \mathbf{b_1} \\ \mathbf{a_{21}} & \mathbf{a_{22} \cdots a_{2n}} & & \mathbf{1} & & \mathbf{b_2} \\ \vdots & \vdots \quad \vdots & & & \ddots & \vdots \\ \mathbf{a_{m1}} & \mathbf{a_{m2} \cdots a_{mn}} & & & \mathbf{1} & \mathbf{b_m} \\ \mathbf{-c_1} & \mathbf{-c_2 \cdots -c_n} & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} \end{pmatrix} \tag{5}$$

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。

4.3 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必要がある。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、まだ探索に使用していない別の頂点を選び、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。頂点を求めるため、以下のように目的関数を作成し、求めた頂点の近傍

の整数格子点を探索に使用する頂点の候補に追加する。

目的関数の作成は変数が n 個の場合、以下のようになる。式 (6) に示す $2n$ 個の目的関数が候補と

$$\begin{cases} \mathbf{Z=x_1, Z=-x_1} \\ \mathbf{Z=x_2, Z=-x_2} \\ \quad\vdots \qquad\quad \vdots \\ \mathbf{Z=x_{n-1}, Z=-x_{n-1}} \\ \mathbf{Z=x_n, Z=-x_n} \end{cases} \tag{6}$$

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。

4.4 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 2 の場合、ディオファンタス方程式は式 (7) となり、do ループの初期値・終端値から、各変数の変域は式 (8) となる。

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

図 3 プログラム例 2

$$\mathbf{3i_1 - 2j_2 = 5} \tag{7}$$

$$\begin{cases} \mathbf{1 \le i_1 \le 4} \\ \mathbf{2 \le j_2 \le i_1 + 3} \end{cases} \tag{8}$$

これらから、4.2 節で述べたように、不等号の向きを統一し、式 (7) 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 (9) となる。得られた目的関数も、同様に式 (7) のディオファンタス方程式で変数消去し、係数行列 (10) 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 (9) 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 (9) の目的関数 (ⅱ)(ⅳ) のとき、$\mathbf{i_1 = 2.333, j_2 = 1}$ が

$$\begin{cases} \mathbf{Z=i_1} & (\text{i}) \\ \mathbf{Z=-i_1} & (\text{ii}) \\ \mathbf{Z=j_2} & (\text{iii}) \\ \mathbf{Z=-j_2} & (\text{iv}) \end{cases} \tag{9}$$

$$\begin{pmatrix} 1 & 1 & 0 & 0 & 4 \\ -1 & 0 & 1 & 0 & -2.333 \\ \pm 1(or) \pm 1.5 & 0 & 0 & 0 & 0 \end{pmatrix} \tag{10}$$

得られる。得られた $(\mathbf{i_1, j_2})$ の近傍点は、それぞれ $(\mathbf{4,4})$,$(\mathbf{3,1})$ であるが、$(\mathbf{3,1})$ から距離 1 の点を求めると、$(\mathbf{3,2})$ 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。

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。

5 性能評価

本章では、実際にプログラムの依存解析を線形計画法と全探索のテストを用いて行い、その実行時間を Banerjee 法と比較しながら測定した。この性能評価は表 1 の環境で実験を行った。

表 1 実験環境

計算機	PC-AT 互換機 CPU Celeron 600MHz 主記憶 256M バイト
OS	Microsoft Windows2000

実験に用いたプログラムは、図 2,3 に挙げた例と、LINPACK のサブルーチンの 1 つである “schdc.f” を選んだ。schdf.f には 7 つの Do ループが存在し、そのうち二重ループが 3 つと単独のループが 1 つという構成である。これらのプログラムを依存解析した結果、表 2 となった。

まず、2 重ループ (ⅰ) は、外部ループでは依存関係が存在するが、内部ループには依存関係が無い為、内部ループのみ並列化が可能である事が判定できた。2 重ループ (ⅱ),(ⅲ) に関しては、外部ループ、内部ループ共に依存関係が存在し、現状では並列化不可能である。単独ループに関しては依存解析の結果は依存ありとなったが、これはループ独立依存であるため、並列化は可能である。

また、測定に要した時間は表 3 の A となった。

表 2　実行結果

ループ	依存	並列化
2 重ループ (i)		
外部ループ	有り	
内部ループ	無し	可能
2 重ループ (ii)		不可
外部ループ	有り	
内部ループ	有り	
2 重ループ (iii)		不可
外部ループ	有り	
内部ループ	有り	
単独ループ	有り	可能
プログラム例 1	有り	不可
プログラム例 2	有り	不可

実行時間の測定方法は、各ループ全体の依存解析を 1000 回繰り返して、1 回当たりの実行時間を求めた。また、本来は Banerjee テストで依存がないと判定される場合には線形計画法は実行しないが、今回は比較のため強制的に実行させている。

2 重ループ (ii),(iii) のように、ディオファンタス方程式に解が存在する場合、すなわち依存がある場合は、解析に要する時間は短くなっている。これは、解空間の頂点付近に解が存在することが多く、探索を始めた早い段階で解が見つかり、解析が終了するからである。また、解空間のエリアを全探索した結果、解を発見するケースというのは稀であった。逆に 2 重ループ (i) のように依存が無いループを持つ場合には、解析に要する時間は長くなっている。これは、解空間のエリアを全探索するという性質上、解が無い場合では、全ての整数格子点を探索する必要があるので、実行時間の増大の原因となっている。

表 3　実行時間

ループ	A [ms]	B [ms]	比率
2 重ループ (i)	18.96	1.22	15.5
2 重ループ (ii)	7.25	1.33	5.4
2 重ループ (iii)	7.42	1.31	5.6
単独ループ	3.63	0.61	5.9
プログラム例 1	1.21	0.20	6.0
プログラム例 2	3.24	0.15	21.6

A：線形計画法と全探索のテストの実行時間
B：Banerjee テストの実行時間

表 3 の B は同じ条件で Banerjee テストを用いて依存解析を行った実行結果である。この結果から、線形計画法と全探索のテストの実行時間が Banerjee テストの約 5〜21 倍に相当することが分かる。Omega テストは提案手法と同精度の解析手法であるが、文献 [6] によると Banerjee テストの約 70〜80 倍の実行時間がかかる事が分かっている。したがって、線形計画法と全探索のテストは Omega テストよりも高速に同精度の解析が可能だと言える。

6　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不可能である。厳密に整数解の存在を判定できる解析手法は種々提案されているが、Omega テストがその代表的なものである。

文献 [6] 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。

線形計画法と全探索のテストでは、シンプレックス法を用いるが、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。4.3 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。

第 5 章の性能評価の結果から、線形計画法と全探索のテストの実行時間は Banerjee テストの約 5〜21 倍であることがわかった、解空間のエリアを全探索するという性質上、ディオファンタス方程式に解が無い場合や、膨大な解空間のエリアの整数格子点を探索する必要がある場合においては、実行時間の増大が見られるが、Omega テストの実行時間と比較した結果、高速であると言える。

7 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た。性能評価の結果、線形計画法と全探索のテストは Omega テストよりも高速に依存解析を行えることがわかった。

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] 横手聡，齋藤彰一，上原哲太郎，國枝義敏,「コンパイラによる制御が可能な DSM システム Fagus の実現」，情報処理学会研究報告 2000-OS-85, pp.47-54, 2000.

[4] 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)

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

[6] 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.

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

[8] 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).

[9] M.Girkar and C.D.Polychronopoulos.,“The hierarchial task graph as a uiversal intermediate representation，” International Journal of Parallel Programming, 22(5), pp.519-551 (October 1994).