プログラミング－言語・基礎・実践－ 13－6
(1993. 8. 20)

データフロートレーサによる並列論理型言語 Fleng のパフォーマンスデバッギング

舘村 純一, 小池 汎平, 田中 英彦
{tatemura,koike,tanaka}@mtl.t.u-tokyo.ac.jp
東京大学 工学部

概要

並列計算機の実用化と普及のためには、実際に用いられるアプリケーションが高速に動かなければならない。このためには、効率のよい並列プログラムのためのプログラミング技術・プログラミング環境の構築が重要であり、パフォーマンス悪化要因を修正するパフォーマンス・デバッギングの支援が必要である。我々の提案するパフォーマンス・デバッギング方式では、パフォーマンスを決定する要因として、プログラミングにおけるコントロール・データ依存関係と、それを実行するためのスケジューリング・負荷分散とを分類して対処する。本論文ではプログラマに特に必要とされる前者の要因を対象とし、並列論理型言語 Fleng について、実行されたプログラムにおけるデータ依存関係を解析・提示し、性能低下の原因となる不必要な逐次性の発見を支援する方式とツールについて述べる。

Performance Debugging for Parallel Logic Programs with a Dataflow Tracer

Jun-ichi Tatemura, Hanpei Koike, Hidehiko Tanaka
{tatemura,shiraki,koike,tanaka}@mtl.t.u-tokyo.ac.jp
Faculty of Engineering, The University of Tokyo,
Hongo 7-3-1, Bunkyo, Tokyo, 113 JAPAN

Abstract

To develop practical parallel computers which come into wide use, they should execute practical application programs efficiently. Thus, we should realize programming technique and environment for efficient highly parallel programs. One of our essential problems is to develop a performance debugging tool which finds causes of bad performance of complicated highly MIMD parallel programs. In this paper, we describe (1) how to analyze dataflow relationship within execution of a parallel logic program, (2) a performance debugging with this dataflow relationship to find unexpected sequentiality which causes bad performance, and (3) an example of debugging with our performance debugger.

1 はじめに

近年、並列計算機の研究開発が盛んに行なわれているが、これらの計算機が実用化されて広く受け入れられるためには、ハードウェアのみを考えて開発するだけでは不十分である。ユーザにとっては、自分の使いたいアプリケーションが動いた上ではじめて評価が可能になるので、計算機の評価のためには実際にユーザが用いるアプリケーションが高速に動かなければならない。このような背景において、並列計算機システムのソフトウェアの課題は、次のようになろう。

- プログラムを効率よく実行する並列処理技術
- 効率のよい並列プログラムのためのプログラミング技術・プログラミング環境

後者のプログラミング技術においては、並列プログラミングの手法、並列アルゴリズムの確立が重要な課題となっている。これらの知見を得るためには、並列プログラマが実用的な規模のプログラムを開発できるような並列プログラミング支援環境が必要となる。よって、まず課題となるのは、効率のよい高並列プログラムを開発するためのプログラミング環境の構築である。

並列プログラミング環境において、ユーザの要求を満たすプログラムの作成を支援するツールとしては、以下のものがあげられる。

- 結果を誤らせる部分を修正するデバッガ
- パフォーマンスを悪化させる部分を修正するパフォーマンス・デバッガ

並列計算機導入の大きな動機がプログラムの高速化である以上、並列プログラムは速く動かなければならない。このことから、後者のパフォーマンス・デバッガの構築は並列計算機システムにおいて必要不可欠な要件であるといえる。

我々は、並列プログラミングにおけるパフォーマンス・デバッギングとして、プログラマが何をなすべきか、これを支援するためのプログラミング環境はどうあるべきかを考察した [2]。我々の提案するパフォーマンス・デバッギング方式では、パフォーマンスを決定する要因として、プログラミングにおけるコントロール・データ依存関係と、それを実行するためのスケジューリング・負荷分散とを分類して対処する。現在この考え方に基づいて、実行性能の高いプログラムの作成を支援するプログラミング環境の開発をすすめている。

本論文では、並列論理型言語 Fleng について、実行されたプログラムにおけるデータ依存関係を解析・提示し、性能低下の原因となる不必要な逐次性の発見を支援する方式とこれを実装した試作ツールについて述べる。

2 Fleng による並列プログラミング

Concurrent Prolog や GHC などの Committed-Choice 型言語 (CCL) は論理型プログラミングを並列に実行するためガードの概念を導入して通信・同期が記述できるように制御機能を強化した並列論理型言語である。Fleng [1] もこの CCL の一つであり、他の CCL に較べてその言語仕様が簡潔になっている。Fleng はガードゴールを持たず、ヘッドのみがガードの働きをする。よってヘッドユニフィケーションだけで定義節がコミットされる。

Fleng プログラムは次のような定義節の集合である。

$$H{:-}B_1,\cdots,B_n.\quad n\geq 0$$

:- の左側はヘッド部、右側はボディ部と呼ばれ、B_i はボディゴールと呼ばれる。

Fleng プログラムの実行は、ゴールの集合の書き換えによって進められる。各ゴールについてパターンマッチが成功するようなヘッド部を持つ定義節が一つ選ばれ、これに基づいて新しいゴールに書き直される。この書き換え操作をリダクション、マッチング操作をユニフィケーションと呼ぶ。CCL の場合、ユニフィケーションは、2 つの種類に分けられる。一つはヘッド部で行なわれるガードつきユニフィケーションで、ゴール側からのデータを待つ時に用いられる。もう一つはボディ部で行なわれるアクティブ・ユニフィケーションで、ゴールの持つデータに値を代入する。このように、Fleng プログラムは、ゴールリダクションによる制御依存関係と、ガードつき / アクティブ・ユニフィケーションによるデータ依存関係を定義節という形で記述するものである。

Fleng によるプログラミングとは、コントロールフローとデータフローの依存関係を記述することだといえる。

3 パフォーマンス・デバッギング

3.1 パフォーマンスを意識した並列プログラミング

Fleng は並列性を自然に記述することができ、記述されたプログラムの並列性を最大限にひきだすことのできる言語であるが、「並列論理型言語でプログラムを書けば、問題のもつ並列性が自然と記述できて自然に高並列な実行ができる」というわけではない。実際には、問題をどのように記述するかがパフォーマンスに大きな影響を与えるので、並列プログラマはパフォーマンスを意識したプログラミングをしなければならない。

並列プログラミングに対して、仕様を記述してアルゴリズムを自動生成するようなプログラミングの自動化の手法や、ユーザがどのようなプログラムを書いてもその問題にとって最高のパフォーマンスを持つコードを生成するような最適化コンパイラなども考えられる。しかし、いかなるコンパイラでもユーザがプログラムや仕様を記述することで指定した制限内でしか最適化を行なうことはできないので、ユーザがプログラミング対象となる実際の問題をモデル化・仕様記述した時点で、すでに得られるパフォーマンスが限定されてしまう。このことから、ユーザがある問題を記述する際にも、どのようにモデル化した

らパフォーマンス面でも要求を満たすプログラムが得られるかという問題は並列処理において極めて重要な課題となる。

パフォーマンスを意識して並列プログラミングを行なうことは従来の逐次プログラミングとは違った技法をプログラマに要求する。この負担を軽減するには、プログラミング環境が重要な役割を果たさなければならない。

3.2 プログラマの意識すべき要因

プログラマが何をどのように意識すればよいのかを考えるには、パフォーマンスを決定する要因を明確にする必要がある。Flengのような並列言語を用いる場合、最適な並列プログラムの実行を得るまでの過程は、次の各段階に分けられる。

1. プログラミング
   コントロールフロー・データフローの依存関係を記述する。
2. 時間・空間上へのマッピング
   スケジューリング・負荷分散を行なう(データ・プロセスの静的配置および動的戦略の決定[4], [5])。
3. 逐次化部分の最適化処理
   同一プロセッサ内に割り当てられた処理の最適化。

これらは互いに関連しており、フィードバックも必要と思われるが、パフォーマンスの問題点を明確にするには、これらの要因を区別して段階的にプログラム作成を進めるべきである。

以上の考察から、プログラマが行なうパフォーマンスを意識したプログラミングとは、上記1の段階において、不必要な逐次性を招くコントロールフロー / データフローの依存関係をなくし、できるだけ並列性が出るように記述することであるといえる。プログラミングが行なわれた次の段階として、実際の環境のパラメータに応じて、スケジューリング・負荷分散を行ない、適度な逐次化をおこなう。この処理は、システムによって行なわれることが望まれる。

3.3 Flengのパフォーマンス・デバッギング

パフォーマンスの問題をコントロール / データフローの依存関係の記述の問題とスケジューリング・負荷分散の問題に分割して段階的にとらえ、また、プログラマと処理系の責任分担を明確にすると、プログラマが行なうパフォーマンス・デバッギングの役割は図1のように表されるだろう。プログラムとしてコントロールフロー / データフローの依存関係を与え、そのマッピングとしてスケジューリング・負荷分散を決定することで、一つの実行が決まる。プログラマがプログラムを記述することで、起こり得る実行の集合が限定される。処理系が最適化を行なっても十分なパフォーマンスが得られない場合は、プログラマがプログラムを変更する必要がある。この変更点を見つけ出し修正する作業がパフォーマンス・デバッギングである。

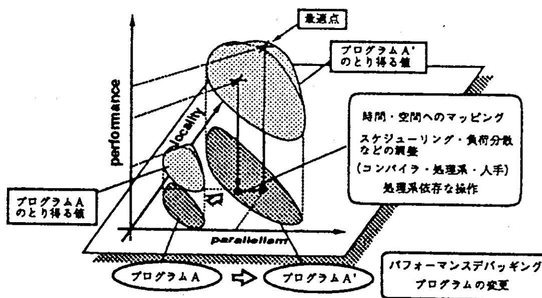

図 1: プログラムのパフォーマンスとパフォーマンス・デバッギング

4 パフォーマンス・デバッギング・ツール

4.1 必要とされる機能

MIMDプログラム、特に大規模知識処理を目指したアプリケーションなどの実際的アプリケーションプログラムは不均質な構造を持つことになろう。このようなプログラムが全体としてどう動くか把握するのは難しい。特に、大規模な高並列プログラムでは、その実行情報は大量なものとなる。パフォーマンス・デバッギングの最初の課題として、大量の実行情報の中から問題となる部分を絞り込み、その部分が全体とどう関わっているかを把握するためのツールが必要とされる。

次の段階として、問題部分がどのように実行性能に影響しているかを詳しく分析する必要がある。不必要と思われる逐次性はないか、どのデータの出力を優先して速くすべきか、ネックとなっている制御・データの依存関係はどれかなどを分析する。この分析をもとに、プログラマは適切なモデル化、適切なアルゴリズムでその部分を実現する。

以上のような作業を支援するためには、

- 問題点の絞り込み
- その問題点の解析

の2機能が必要である。本節の以降では、それぞれの機能について開発されたデバッギング・ツールについて述べる。

4.2 問題点の絞り込み

我々の研究室では、プログラム中のパフォーマンスバグの部分を絞り込むためのパフォーマンス視覚化ツールPafを開発した[3]。これは、膨大な実行データをフィルタリングして、適切なビューをユーザに提供する。プログラムの実行モジュール毎の並列度の時間推移が把握できるので、実行のネックとなっている部分をプログラム中に発見することができる。

4.3 問題点の解析

プログラム全体から問題部分が抽出されたら、その部分がどのように実行性能に影響しているかを詳しく分析する必要がある。そこで、実行履歴とプログラムコードからデータ・制御依

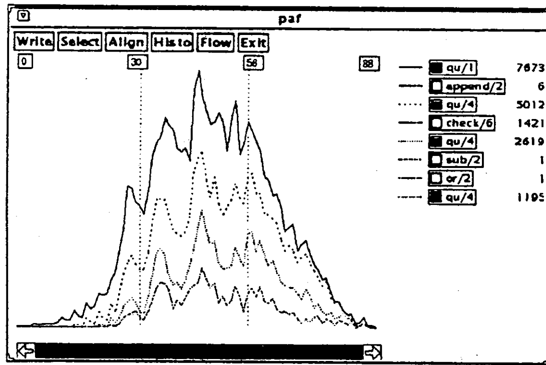

図 2: パフォーマンス・デバッガ Paf

存関係のグラフを作って提示する。これをユーザが分析し、どの依存関係が問題となっているかを発見する。これをもとに、データ依存関係や制御依存関係による不必要な逐次性を取り除く。

次節では、Fleng プログラムの実行におけるデータ依存関係の解析手法、解析結果に基づいたパフォーマンス・デバッギング手法を提案し、これを実現するために試作されたデータフロー・トレーサについて述べる。

4.4 仮想時間による計測

前記の 2 機能を提供するパフォーマンス・デバッガでは、スケジューリング・負荷分散の問題を最初は考えずに、プログラムに内在する並列度を対象としたい。そこで、プロセッサ無限大で実行可能なゴールは直ちに実行されるような理想的環境で実行させた履歴を考える。ここで測定される時刻を仮想時刻と呼ぶことにする。

仮想時間の代わりに、あるマシン上でスケジューリング・負荷分散などを決定して実行した実際の時間を当てはめて実行履歴を表示してみることも考えられる。これにより、パフォーマンス・デバッギングの効果を確かめるというフィードバックの効果が得られる。

5 データフロー・トレーサによる問題点の解析

Fleng のゴールの実行 (そのサブゴールの実行を含む) は、互いに依存関係を持った入出力の履歴として表現できる。入出力の履歴は、ある出力が行なわれるまでに必要となる入力がわかるような半順序関係を持った入出力の組で表される。これは、プログラムの宣言的意味として与えられる [6] ので、プログラムの仕様を変えない限り、その内部を最適化しても不変なものである。これにより、問題となる部分が現在注目しているレベルに存在するのか、さらに内部に存在するのかを区別できる。

そこでこの入出力の依存関係を表示し、逐次性を招く不必要な依存関係の解消を支援する手法を開発し、これにもとづくデータフロー・トレーサを試作した。

5.1 ユニフィケーションと入出力の関係

論理型言語においてデータの入出力を議論する時には、実行されるゴールの各引数のデータについて、そのゴールの内部 (サブゴール) においてデータが代入された場合を出力と考え、外部においてデータが代入された場合を入力と考えている。ユニフィケーションはデータの代入に対して双方向性をもつので、同じプログラムが実行時に入力としても出力としても働きうる。これに対し、Fleng や GHC などの CCL では同期・通信を表現し並列プログラミングを可能にするためにユニフィケーションの方向性に制限を加えている。ガード部におけるユニフィケーションは、外部のデータに対する代入を行なわないので、常に入力として作用する。ボディ部におけるアクティブ・ユニフィケーションは外部のデータに代入が行なえるので出力として働くことができる。しかし、変数とデータに対しておこなわれるアクティブ・ユニフィケーションで、変数に対して外部からデータが与えられた場合、このアクティブユニフィケーションは入力の働きをする。

現在の実装における解析プログラムでは、対象とする Fleng プログラムについて次の仮定をおくことにする。

> データ同士のアクティブ・ユニフィケーションが存在しない (すなわち、データの出力が競合しない)

実際上、バグ以外でこのようなことが起きる場合は少ない。存在する例の中でも、ガードつきユニフィケーションとアクティブ・ユニフィケーションに分けて記述すればすむ場合が多いと考えられる。このような考察から、現在の方式では、解析結果が複雑になるのを避けるために、このような仮定をおくことにする。この仮定によって、変数とデータとのアクティブユニフィケーションでは代入の方向性が静的に決定される。

このとき、ユニフィケーションは以下の 3 種類に分類される。

- in(Variable, Term) : ガード付きユニフィケーションによるデータの入力
- out(Variable, Term) : アクティブ・ユニフィケーションによる未定義変数へのデータの代入 (データの出力)
- unify($\mathrm{Variable}_1, \mathrm{Variable}_2$) : アクティブ・ユニフィケーションによる変数 (ポインタ) 同士の単一化

5.2 入出力依存関係

一つのゴール (プロセス)G の入出力依存関係は、入力条件と出力の組を要素とする集合

$$IO(G) = \{io_i | io_i = (ICondition_i \rightarrow O_i)\}$$

で表現される。ここで、出力 O_i はアクティブ・ユニフィケーションであり、out(V_i, T_i) または unify(V_{i1}, V_{i2}) と表される。

入力条件 $ICondition$ は、入力の AND 関係である。

$$\begin{aligned} ICondition &= (I_1 \wedge \cdots \wedge I_n) \\ I_i &= \begin{cases} \mathrm{in}(V_i, T_i) \\ ICondition_{i1} \vee \cdots \vee ICondition_{im_i} \end{cases} \end{aligned}$$

ここで、I_i が入力条件の OR 関係になるのは、以下の 2 条件がともに成り立つときのみである。

- unify(X,Y) によって単一化された二つの変数がともに外部から参照可能である
- この変数 (X,Y) に外部から与えられたデータが入力条件となっている

このとき、外部からどちらの変数を通じてデータが入力されるかによって条件がことなることがある。例えば、以下のプログラム a(A,B,C,D) において、外部からデータが B に対して与えられるか C に対して与えられるかによって入力の条件がことなる。

```
a(A,B,C,D) :- b(A,B,C),c(B,D).
b(true,B,C) :- B = C.
c(true,D) :- D = true.
```

$\{((\mathrm{in}(B,true) \vee (\mathrm{in}(A,true) \wedge \mathrm{in}(C,true))) \rightarrow \mathrm{out}(D,true))\}$

5.3 タイムスタンプつき入出力依存関係

データフローのネックとなる部分 (クリティカル・パス) を表現するため、入出力依存関係中の各入力・出力について、次のような時刻を付加する。

- 入力: そのデータがガード付きユニフィケーションによって初めて参照された時刻
- 出力: そのデータがアクティブ・ユニフィケーションによって書き込まれた時刻

この時刻には仮想時刻を用いるが、スケジューリングなども考慮したモデルや、実際の測定値での時刻を用いることも可能である。

5.4 入出力依存関係の解析手法

入出力依存関係は、計算の履歴から求めることができる。ここで計算の履歴とは、実行されたゴールの親子関係 (これによって形成される木構造を計算木という) と、各ゴールの実行のために選択された定義節を意味する。本節ではこの解析アルゴリズムの概略を示す。

入出力依存関係を求める処理の全体構成は以下のようになる。

1. 定義節の展開
2. アクティブ・ユニフィケーション、システム・ゴールの入出力
3. ボディゴールの各入出力から入出力を求める

   (3-1) ボディーゴール間の入出力の合成

   (3-2) 外部参照不可能なデータの削除

   (3-3) ガードにおける入力条件の付加

まず 1 によって計算木が構成され、これに基づいて解析が行なわれる。計算木の末端はユニフィケーション、及びシステム述語であり、これらの入出力は 2 によって求められる。これについて 3 を繰り返し適用することによって各ゴールに関する入出力依存関係をボトムアップに解析することができる。

以下では、各処理の概要について述べる。

定義節の展開 以下のような定義節を考える。

$$H \ :- \ U_1, \cdots, U_j, B_1, \cdots, B_i.$$

ここで H はヘッド部、U_j はアクティブ・ユニファイ述語、B_i はそれ以外のボディゴールとする。H、B_i はそれぞれ次のような引数を持つとする。

$$H(t_1, \cdots, t_k, V_1, \cdots, V_m),\ B_i(t_{i1}, \cdots, t_{ik_i}, V_{i1}, \cdots V_{im_i})$$

t は変数でないデータ、V は変数である。このとき H と B_i をそれぞれ以下のように展開する。

$$\begin{cases} H \Rightarrow \{H'(V'_1, \cdots, V'_k, V_1, \cdots, V_m),\ G\} \\ G = \{V'_1 = t_1, \cdots, V'_k = t_k\} \\ B_i \Rightarrow \{B'_i(V'_{i1}, \cdots, V'_{ik_i}, V_{i1}, \cdots, V_{im_i}),\ U\} \\ U = \{V'_{i1} = t_{i1}, \cdots, V'_{ik_i} = t_{ik_i}\}\} \end{cases}$$

G はガード、U はボディー部でのアクティブ・ユニファイ述語として扱う。これにより展開された定義節は次のように表される。

$$H' \ :- \ G_1, \cdots, G_k \mid U_1, \cdots, U_{j'}, B'_1, \cdots, B'_i.$$

この展開を計算木のすべての定義節についておこなう。ここで、親ゴールに関する定義節のボディゴールとそのボディゴールに関する定義節のヘッドは同じゴールを表すから、その引数の各変数をそれぞれ単一化して一致させておく。

アクティブ・ユニフィケーションとシステム・ゴールの入出力 V、V_1、V_2 を変数、T を変数でないデータ (項) とすると、ボディー中のアクティブ・ユニファイ述語の入出力依存関係は以下のように表現される。

$$V = T \quad \Rightarrow \quad \{(true \rightarrow \mathrm{out}(V,T))\}$$
$$V_1 = V_2 \quad \Rightarrow \quad \{(true \rightarrow \mathrm{unify}(V_1, V_2))\}$$

ここで、各 out、unify には、このユニフィケーションが行なわれた時刻が付加される。ユニフィケーション以外のシステム述語は、システムによって定められた各引数の入出力モードに従って以下のように入出力依存関係を表現する。

$$\begin{cases} a(I_1, \cdots, I_n, O_1, \cdots, O_m) \\ \{(IC \rightarrow \mathrm{out}(O_1, t_{O1})), \cdots, (IC \rightarrow \mathrm{out}(O_m, t_{Om}))\} \\ IC = (\mathrm{in}(I_1, t_{I1}) \wedge \cdots \wedge \mathrm{in}(I_n, t_{In})) \end{cases}$$

ここで、a はシステム述語、I_i は入力モードの引数、O_i は出力モードの引数、t_{Ii} は I_i に与えられたデータ、t_{Oi} は O_i に出力されたデータである。各 in、out にはシステム述語が実行された時刻が付加される。

ボディーゴール間の入出力の合成　ある2つのボディゴール B_i、B_j の各入出力依存関係 $IO(B_i)$、$IO(B_j)$ から入出力依存関係 $IO(B_i, B_j)$ を合成することを考える。

$$io_1 = (IC_1 \rightarrow O_1) \in IO(B_i),\ io_2 = (IC_2 \rightarrow O_2) \in IO(B_j)$$

における IC_1 の中の入力 $I = \mathrm{in}(V_{in}, T_{in})$ と $O_2 = \mathrm{out}(V_{out}, T_{out})$ について $V_{in} = V_{out}$ が成り立つとき、入力 I は出力 O_2 を受けとっていると考えられる。ここで、T_{in} と T_{out} を単一化するのに必要な T_{in} の変数への代入を $\theta(T_{in})$、T_{out} の変数への代入を $\theta(T_{out})$、T_{in} と T_{out} の変数同士の単一化を θ_v とする。このとき、入出力の合成のために以下のような処理をおこなう。

- $\theta(T_{in})$: 入力 I で受けとった変数 (ポインタ) に対する出力。この各出力と入力条件 IC_2 からなる入出力の組が作成される。
- $\theta(T_{out})$: 出力 O_2 以外にさらに入力として必要な条件。入力条件 IC_1 から I を削除し、かわりに $\theta(Tout)$ および入力条件 IC_2 を挿入する。
- θ_v: 出力側の変数 (ポインタ) を入力側が受けとることを表す。変数の名前だけの問題なので単一化を行なってしまう。

また、$I = \mathrm{in}(V, T)$ に対して入出力の組 $(IC_O \rightarrow \mathrm{unify}(V, V'))$ が存在する時、I は以下の入力条件に置き換えられる。

$$((\mathrm{in}(V, T)) \vee (IC_O \wedge \mathrm{in}(V', T)))$$

外部参照不可能なデータの削除　ボディゴールの入出力を合成した $IO(B_1, \cdots, B_n)$ から外部とのデータのやりとりに関係する入出力の組だけを選択する。このためには外部参照可能な変数に関する入出力だけを取り扱い、他を削除すればよい。ここで外部参照可能とは次のように定義される。

- 定義節のヘッド部に与えられた変数は外部参照可能である。
- V が外部参照可能な変数である時、
  - in(V,T) のデータ T の中の変数は外部参照可能である。
  - out(V,T) のデータ T の中の変数は外部参照可能である。

ガードにおける入力条件の付加

$$G = \{V'_1 = t_1, \cdots, V'_k = t_k\}$$

を

$$ICondition = \mathrm{in}(V'_1, t_1) \wedge \cdots \wedge \mathrm{in}(V'_k, t_k)$$

として各入出力の組の入力条件に AND 関係で付加する。このとき、各 in(V,T) に対しては、リダクション時にこの変数を参照した時刻を付加する。

5.5　入出力依存関係の表示例

試作したデータフロー・トレーサでは入出力依存関係を以下のような書式で表示する。

```
ゴール:<リダクションされた時刻>
input[
[識別番号] 入力 V = T <入力時刻>
...
]
output[
[入力条件] -> 出力 V = T <出力時刻>
...
]
```

ここでは、このトレーサを用いて表示した入出力の依存関係の例として、以下のプログラムを実行した場合の解析結果をあげる。

```
flatten(Tree,List) :- flatten(Tree,List,[]).
flatten(node(A,B),X,Y) :-
        flatten(A,X,Z),flatten(B,Z,Y).
flatten(leaf(A),X,Y) :-
        X = [A|Y].
```

これは木構造を線形リストに変換するプログラムである。第一引数に木構造データを与えた時の解析例は以下のようになる。

```
?- iotrace(flatten(node(node(leaf(1),
                   leaf(2)),leaf(3)),X)).
flatten(A,B):<0>
input[
  [1] A = node(C,D) <1>
  [2] C = node(E,F) <2>
  [3] D = leaf(G) <2>
  [4] E = leaf(H) <3>
  [5] F = leaf(I) <3>
]
output[
  [] -> J = [] <0>
  [1,3] -> K = [G|J] <2>
  [1,2,4] -> B = [H|L] <3>
  [1,2,5] -> L = [I|K] <3>
]
```

ここで、?-iotrace(*Goal*) という行が解析プログラムを起動する部分であり、*Goal* に対象プログラムが与えられる。これ以降の行の表示が解析結果である。

5.6　パフォーマンス・デバッギングへの応用

本節では、以上のようにして得られた入出力依存関係をもとにパフォーマンス・デバッギングを行なう手法について述べる。

ここで行なわれるパフォーマンス・デバッギングは、必要とされる出力のためのデータフローのクリティカルパス中から、

ユーザにとって不必要な依存関係を発見する過程であり、この依存関係がパフォーマンス・バグである。これを探索するためには、以下の処理を繰り返す。

1. ゴールGの入出力依存関係を調べ、ネック(クリティカルパス)となっている出力に着目する。
2. 外部との入出力依存関係で不必要な依存関係のために出力遅延が生じている場合、この依存関係を定義する部分がパフォーマンスバグである。
3. 出力遅延についての不必要な依存関係がない場合、サブゴールを調べる。サブゴールのうち着目した出力を行なっているゴールをGとする。
4. 必要な入力の与えられる時間がネックとなって着目する出力が遅延している場合、この入力を与えているゴールをGとする。

6 具体例

本節では、前節で述べたパフォーマンス・デバッギング手法の具体例を示す。例題として以下の二つのプログラム t1(A,B,C)、t2(A,B,C) を比較してみる。これは、第1引数に与えられたリストを順次第2引数に返し、第1引数のリストが完結すると第3引数にその逆順リストを返すプログラムである。

```
t1(A,B,C) :-
        a1(A,A,B,D), b(D,C).
a1([X|A],B,C,D) :-
        C = [X|C1], a1(A,B,C1,D).
a1([],B,C,D) :- C = [], D = reverse(B).

t2(A,B,C) :- a2(A,B), b(reverse(A),C).
a2([X|A],B) :-
        B = [X|B1], a2(A,B1).
a2([],B) :- B = [].

b(reverse(A),B) :- rev(A,[],B).
rev([X|A],B,C) :- rev(A,[X|B],C).
rev([],B,C) :- C = reversed(B).
```

t1([1,2],X,Y) t2([1,2],X,Y) をそれぞれ実行した場合の入出力の結果は次のようになる。

```
t1(A,B,C):<0>
input[
  [1] A = [D|E] <1>
  [2] E = [F|G] <2>
  [3] G = nil <3>
]
  [1] -> B = [D|H] <1>
  [1,2] -> H = [F|I] <2>
  [1,2,3] -> I = nil <3>
  [1,2,3] -> J = nil <4>
  [1,2,3] -> K = [D|J] <5>
  [1,2,3] -> L = [F|K] <6>
  [1,2,3] -> C = reversed(L) <7>
]

t2(A,B,C):<0>
input[
  [1] A = [D|E] <1>
  [2] E = [F|G] <2>
  [3] G = nil <3>
]
  [1] -> B = [D|H] <1>
  nil -> I = nil <1>
  [1,2] -> H = [F|J] <2>
  [1] -> K = [D|I] <2>
  [1,2,3] -> J = nil <3>
  [1,2] -> L = [F|K] <3>
  [1,2,3] -> C = reversed(L) <4>
]
```

第3引数の結果に関する入出力の依存関係はそれぞれ同様で、第1引数のリストがすべて与えられたことを条件に第3引数の結果が返される。しかしその時刻に大きな差がある。この変数Cへの出力に関して着目し、t1 のサブゴールの入出力依存関係を調べる。

```
a1(A,A,B,M):<1>
input[
  [1] A = [D|E] <1>
  [2] E = [F|G] <2>
  [3] G = nil <3>
]
output[
  [1] -> B = [D|H] <1>
  [1,2] -> H = [F|I] <2>
  [1,2,3] -> M = reverse(A) <3>
  [1,2,3] -> I = nil <3>
]
b(M,C):<4>
input[
  [1] M = reverse(A) <4>
  [2] A = [D|E] <5>
  [3] E = [F|G] <6>
  [4] G = nil <7>
]
output[
  [1] -> J = nil <4>
  [1,2] -> K = [D|J] <5>
  [1,2,3] -> L = [F|K] <6>
  [1,2,3,4] -> C = reversed(L) <7>
]
```

Cに対する出力はbが行なっているが、これは第1引数の入力を必要とする。この入力を受けた時刻が4となっていて、これ

はa1が時刻3までそのデータを出力していないからである。a1はAのリストの末端までの入力を待ってからこの出力を行なっているが、プログラマの意図としては、本来その必要はない。これがパフォーマンスを低下させる不必要な依存関係となっている。

7 課題

本研究では、入出力依存関係を考慮したパフォーマンス・デバッギングの方式を示すためにデータフロー・トレーサを試作したが、これを実用的なツールに発展させるには以下のような課題がある。

7.1 グラフィカル・インタフェース

データフロー・トレーサでユーザが把握しなければならない関係には以下の3種類がある。

- 入出力間の依存関係
- データ間のポインタによる参照関係
- 各入出力の時間関係

現在の試作版では、入出力間の依存関係は各入力に番号をつけて区別しているが、これでは関係を把握しにくい。また、各入出力によってデータ構造が構成されるが、変数で表されたポインタの参照関係をたどっていかなければならない。現在は変数を文字で記述しているが、これを把握するのは大変困難な作業である。これを解決するために、図3のようにグラフィック表示を行なう。図中で白い矩形が出力データ、灰色の矩形が入力

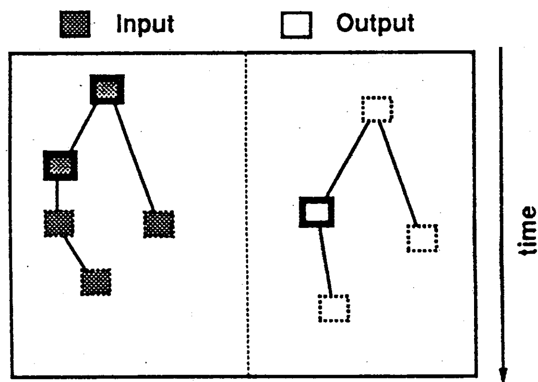

図 3: 入出力依存関係のグラフィック表示

データを表す。これによりユーザが把握すべき上記3種類の関係をそれぞれ以下のように表す。

- 各入出力の時間関係を縦方向の座標で表す。
- データ間のポインタによる参照関係を直線でつないで表す。
- マウスカーソルで出力データの矩形を指示した時に対応する入力データの矩形の色を変えることで入出力間の依存関係を表す。

7.2 パフォーマンス・デバッギング・ツールの統合化

パフォーマンス・プロファイリング・ツールPaf[3]と統合し、問題点をPafで絞り込み、データフロー・トレーサで解析を行なうことを可能にする必要がある。

また、シミュレータや実マシン上のパフォーマンス・モニタからの時刻情報を反映し、様々な実行モデルに基づいた実行結果を得る。これにより、スケジューリング・負荷分散の影響を反映することも可能となる。

8 おわりに

本論文では、実行されたFlengプログラムにおけるデータ依存関係の解析手法を提示し、これを用いて性能低下の原因となる不必要な逐次性の発見を支援する方式とツールについて述べた。

今後は、ここで試作されたデータフロー・トレーサをもとに、グラフィカル・インタフェースを備えた実用的なツールを開発する予定である。

参考文献

[1] Nilsson, M. and Tanaka, H.: *Fleng Prolog - The Language which turns Supercomputers into Prolog Machines.* In Wada, E.(Ed.): Proc. Japanese Logic Programming Conference. ICOT, Tokyo. June 1986. pp.209-216.

[2] 舘村, 白木, 小池, 田中: 並列論理型言語 Flengのパフォーマンスデバッギング – Toward a Programming Environment for Efficient Highly Parallel Programs, 情報処理学会研究報告 プログラミング－言語・基礎・実践－, Vol.92, No. 67 (1992).

[3] 白木, 舘村, 小池, 田中 : 高並列プログラムのパフォーマンス・デバッギング・ツール Paf, 情報処理学会第8回プログラミング – 言語・基礎・実践 – 研究会 SWoPP '92, 1992年8月.

[4] 日高, 小池, 舘村, 田中 : 実行プロファイルに基づくコミッティドチョイス型言語の静的負荷分割手法, 情報処理学会論文誌 Vol.32 No.7, pp. 807-815 (1991).

[5] 日高, 小池, 田中 : PIE64の並列処理管理カーネルのアーキテクチャ, 情報処理学会論文誌 Vol.33 No.3, pp. 338-348 (1992).

[6] Murakami, M. : A Declarative Semantics of Flat Guarded Horn Clause for Programs with Perpetual Processes, Theoret. Comp. Sci, Vol.75 No.1/2, pp. 67-83 (1990).