プログラミング-言語・基礎・実践- 8－18
(1992. 8. 21)

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

舘村 純一, 白木長武, 小池 汎平, 田中 英彦
{tatemura,shiraki,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。しかし、実用規模の複雑な MIMD 並列プログラムの開発手法は現在不明確で、パフォーマンス悪化の原因の発見は困難である。プログラマはいかにパフォーマンスデバッギングを行なうべきか、これを支援するプログラミング環境への要件を考察し、実行性能の高いプログラムの開発のためのプログラミング環境の実現に必要なデバッギング手法を提示する。

Performance Debugging for Parallel Logic Programs
– Toward a Programming Environment for Efficient Highly Parallel Programs

Jun-ichi Tatemura, Osamu Shiraki, 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 discuss what programmers should do for the performance debugging, how the programming environment can help them, and propose our performance-debugging technique to develop efficient parallel programs.

1 はじめに

現在、並列計算機の研究開発が盛んとなっているが、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トイ・プログラムを高速に動かして性能を競うのではなく、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。ハードウェアに関しても、システム全体としての性能を考えてアーキテクチャを設計・評価する必要があり、ソフトウェアに関しても、実際的な並列アプリケーションを効率よく開発・実行するための技術が必要とされる。

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。

では、並列プログラミング環境として、プログラマに対してどの様な支援が必要となるであろうか。開発したプログラムが「まともに動く」状態になるには、

- 正しい結果を出す
- 充分な速度で動く

ことが必要となる。プログラムが正しい結果を出さなければならないのはもちろんであるが、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、「まともに動く」プログラムの二つの要件を満たすために、それぞれ以下の作業が行なわれる。

- (ロジカルな) デバッギング : 結果を誤らせる部分を修正する
- パフォーマンス・デバッギング : 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・デバッギングという言葉には、「パフォーマンス面での要求を満たさないプログラムは、バグがあって動作に制限があるのと同様に問題のあるプログラムだ」といった考えがこめられている。このようなプログラム中のパフォーマンス悪化の原因は「パフォーマンス・バグ」ということになる。

われわれの目標とするところは、ループ処理を基本にした数値計算の並列化だけでなく、大規模知識処理などを含む、より一般的で不均質非定型な問題を対象にしたアプリケーションについても並列計算機で高速に実行することである。しかし、このようなプログラムは多くのコントロールフローとデータフローによって構成される複雑なMIMDプログラムとなるであろう。これを高速に実行するためにどのようにプログラムを開発していけばよいかは現在明確になっておらず、パフォーマンスを悪くするような原因を見つけ出すのは難しい問題である。

本論文では、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、実行性能の高いプログラムを開発するためのプログラミング環境を実現するのに必要なデバッギング手法を提示する。

2 従来のパフォーマンスチューニング

従来、実行性能の高いプログラムを開発するのに、並列計算機上でプログラムを走らせながら、パフォーマンスモニタを用いてパフォーマンス・データを測定をし、これをもとにプログラムのチューニングをする手法がとられてきた。これは、測定したデータを参考に書き換えたプログラムを実行してみてパフォーマンスの変化を測定し直し、試行錯誤を繰り返して最適なものを得るというものであった。このために、パフォーマンスデータを測定するツールが開発されてきた [6] が、パフォーマンスを解析・評価するものとして、並列度・通信量・プロセッサ稼働率などを測定するだけなので、プログラムの実行性能がどのように悪いか、その症状はわかるが、プログラム中のどこを直せば良いかをこのことから直接知るのは難しい。

そこで、パフォーマンスを悪化させる要因を特定するためのツールとして、パフォーマンス・データをプログラムの場所毎に集計するプロファイリング・ツール [7] や、プログラムの実行に沿ってデータを記録するトレーシング・ツール [8][9] が開発されてきた。

パフォーマンスの最終的な評価値は実行時間であるが、これを決定する要因にはさまざまなものが存在する。しかし、従来のパフォーマンスチューニング・ツールでは、実行速度を決定するさまざまな要因が混在したままの実行結果を観察して、どこに問題が存在しているかを発見しなければならなかった。プログラムの規模が比較的小さく、あまり複雑でないような場合には、プログラムを変更する選択肢は少ないのでチューニングが可能であろうが、大規模で複雑な並列プログラムでは、諸要因が混在する中で、発見的にパフォーマンスを向上させる手法は通用しない。パフォーマンスを決定する諸要因を分析して、明確な方針のもとにパフォーマンスの問題点を探索する必要がある。

プログラムの実行情報をプログラマの視点から観察し、これをもとにプログラムのどの部分がどのように悪くて、どう直せるかをつきとめるというような、パフォーマンスデバッギングのためのツールが必要である。

3 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。

4 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。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。

ユーザとシステムの責任境界　プログラマがパフォーマンスを意識しなくとも、最高のパフォーマンスを得るプログラムをシステムが自動的に生成できないであろうか。これには、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が考えられる。しかし、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。いかなるコンパイラでもこのユーザの指定した制限内で最適化を行なうしかない。いずれにせよ、プログラマとシステムが互いに分担すべき責任は存在しており、異なるのは、どこに分担の境界があるかの問題である。プログラマが扱うのはより抽象的なレベルであることが望ましいが、ユーザが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が残る。これを解決するためにも、効率の良い並列アルゴリズム・並列プログラミング技法と、それに応じた問題のモデル化手法などを蓄積していくことが、現在の重要な課題である。

プログラム並列化と並列プログラミング　プログラマが最初から並列を意識したプログラムを記述するのに対し、逐次プログラムを記述して、あるいは既存の逐次プログラムをそのまま使って、それをコンパイラによって並列化する方式も研究されている。このような方式は、従来からのソフトウェア、プログラミング技術が使えると言う利点があるが、これは、逐次的な処理量を減らすようにプログラマが記述したものをもとにコンパイラが並列化を行なうものである。コンパイラは、記述された逐次プログラムの意味を変えない範囲で、もとの問題に対して不必要な制限を加えられたまま最適化を行なわなければならない。現在、プログラム並列化の対象の中心となっているのは、ループ処理を中心とした比較的単純なコントロールフローをもつプログラムであるが、対象が非定型な処理になるほど、プログラム並列化に対する不必要な制限は大きくなると思われる。知識処理のような分野では、問題自体に不均質な並列性を含み、動的で複雑なコントロールフローのグラフを持つので、これを従来の逐次プログラミングで書くと、もとの問題のもつ並列性を失う可能性が高いと思われる。

並列プログラマに与えられた課題　このように、並列処理で高いパフォーマンスを得るには、プログラマが適切なアルゴリズムを用いてプログラミングを行なわなければならない。しかし、逐次プログラミングと比べて、並列プログラミングではパフォーマンスを意識したプログラミングの方法論がまだ確立されていない。図 1は、プログラマが問題対象を並列プログラムとして記述し、これが並列コンパイラによって最適化を受けて実行される過程を表している。ユーザプログラミングの段階では、定量的な考察を行ないながら、アルゴリズムを選択し、効率の良いプログラムを記述する。このとき、逐次プログラムでは 1 次元的な量を考えれば良かったが、並列プログラムでは、定量的な考察を行なうのに空間的感覚が必要になってくる。これは、並列プログラムではコントロールフローやデータフローの依存度が問題になってくるからである。これらは複雑に絡み合っており、1 次元的な量では表現し切れない。

並列プログラマの負担の軽減　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。その上で、将来的には、並列プログラム開発に有用なプログラミング技法やライブラリの蓄積がなされればよい。

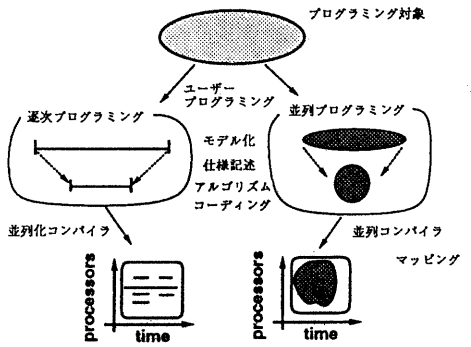

図 1: 並列プログラミング

5 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

パフォーマンスを意識した並列プログラミングは実際にはどのようにして行なえば良いのであろうか。プログラマが何をどのように意識すればよいのかが問題としてあがってくる。

Flengプログラムの実行のされ方 Flengプログラムは、コントロールフロー・データフローの依存関係の記述と見なすことができる。ゴールやデータをどこに配置するかという負荷分散の指定や、ゴールをどのような順番で処理するかというスケジューリングの指定はこのプログラム中には含まれていない。これらは以下のようにして、プログラムとは別のレベルで決定される。

- プログラマや解析プログラムが付加するアノテーションによる指示
- コンパイラによる最適化処理
- 実行時の決定

これをもとに、どのゴールをいつどこで実行するかが決定されてひとつの実行が決定される。

最適なプログラムが走るまでの作業工程 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 (静的配置の決定・動的戦略の決定)。
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、基本的には段階的に進めることのできるものである。

1はプログラマの仕事であり、実際の問題をモデル化して、本来不必要な逐次性のない、並列度の高いプログラムを記述する。2で行なわれるマッピングの自由度が大きいように、細粒度で高並列なプログラミングが期待される。

2は将来的にはシステムが行なうことが望まれる仕事である。我々の研究室では、実行プロファイルに基づく静的負荷分割手法 [2] や、並列処理管理カーネルがプログラム実行時に行なう動的負荷分割・優先度を用いたスケジューリング [3] などの研究を行なっており、また、プログラムの静的解析を行なって静的スケジューリングを行なう並列最適化コンパイラの検討も行なっている。これらは、細粒度で高並列に記述されたプログラムを実機のパラメータにあわせて適度に逐次化し、並列処理による余計なオーバヘッドを除去する処理である。これらによって、実際の並列計算機の環境に応じたプログラム実行の割当が行なわれる。

3は、2でスケジューリングが決まり、実際のマシンの上では同一プロセッサで逐次的に実行されることになったゴールの列について、ゴール生成のオーバヘッドの除去などを行なうものである。プロセッサ内の処理については、逐次プログラムの最適化技術が適用できる。

これらをまとめると、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、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。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のが望ましい。

Flengのパフォーマンスデバッギング このように、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2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。

6 ケーススタディ

パフォーマンスデバッギングをする際に、プログラミング環境にどのようなことが求められるだろうか。Flengによる並列

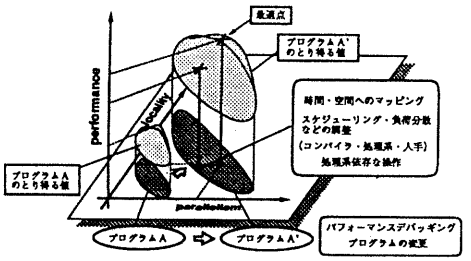

図 2: プログラムのパフォーマンスとパフォーマンスデバッギング

アプリケーション開発における実経験から、その問題を考察してみる。

6.1 マルチウインドウデバッガ HyperDEBU

Flengのマルチウィンドウデバッガ HyperDEBU[4] は Fleng自身で記述されており、それ自身が実用規模の並列アプリケーションである。

HyperDEBU の持つ機能について高速化をはかる場合、一つの機能を実現するのにはいくつものモジュールが協調して並列動作しているので、どの部分をどのように改良すれば効果を発揮するかが問題となる。例えば、ユーザが実行履歴を観察するときには、実行プログラムを管理するモジュール、実行履歴を管理するモジュール、履歴データをユーザの意図に応じた表現にして表示するモジュール、実際に表示を行ないユーザからのイベントを受けつける各ウィンドウ、ユーザからの入力を解釈してプログラムや履歴を操作するモジュールなどが並行して動作している。しかも各モジュール内も並列にプログラムされおり、どの部分が実行性能を規制しているのか判断するのは難しい。

また、HyperDEBU の開発の過程で、新しく開発するある部分をデザイン(モデル化)してプログラミングする時には、どのような方針のもとに行なえばよいかが問題となる。具体的な例としては、実行履歴やプログラムに関する情報など、デバッガが管理するデータを処理する際にも、

- 構造データとしてデータを持っていて、これを並列に処理するようにデータにそってゴールを生成する
- ゴールを使ってプロセスを表現し、ネットワークを構成することでデータ構造を表現し、メッセージを並列に流すことでデータを処理する

などいくつかモデル化の仕方に選択がある。このような場合に、パフォーマンスを考慮して最適な選択を行ないたい。

6.2 非単調推論システム

我々の研究室では、Flengで記述されたアプリケーションシステムの一つとして、並列非単調推論システムを開発している。これは、推論部分(プロダクションシステム)と真理値管理部分(ATMS)が協調動作することで、互いに矛盾するような推論を仮説から構成される多重環境のもとで行なうシステムである。プロダクションシステムでは、Rete ネットワークを用いた推論ルールの照合部分を並列化している。ATMS では、仮説と導き出された事実とからなるグラフをプロセスのネットワークで表現し、並列に管理を行なっている。

このシステムのパフォーマンスを改善するには、まずこれらの各部分がそれぞれシステム全体にどう影響を与えているのかを把握すしなければならない。パフォーマンスのネックになっているのは、ATMS 側かプロダクションシステム側か、どちらか一方がネックになっているとしたら、さらにその内部でどの部分がネックとなっているのかなどを調べることで、パフォーマンスに影響を与えるような改善すべき部分を絞り込んでいくことが必要である。

6.3 最短経路探索問題 (bestpath)

より課題を明らかにするために、もう少し小さなプログラムについてパフォーマンスでバッギングの様子を具体的に見ていく。前述したアプリケーションが数十から数百程度のモジュールからなるのに対し、2から3モジュール程度の構成要素を持つプログラムである。

ここでは、最短経路探索問題(ベストパス問題)をとりあげる。これは、コストつきの有向グラフが与えられていて、あるノードから他のすべてのノードまでそれぞれの最短ルートを調べると言うものである。

- アルゴリズム A
  (searchers and an evaluator — data-oriented model) :
  グラフ間を移動するデータを Fleng のゴールで表現する。searcher はノードから次の各ノードに関して searcher をフォークする。searcher はコストを計算しながら evaluator と通信し、現在のコストを報告する。evaluator はコスト情報を管理していて、報告された情報が最短値を更新するかどうかを検査する。

実際には、この方法ではデータを一括管理している evaluator がネックとなり、よいパフォーマンスが得られない。

そこで、モデル化をしなおして、違ったアルゴリズムを適用してみる。

- アルゴリズム B
  (processes as nodes – process-oriented model):
  グラフの各ノードをゴールで表現する。隣のノードに対して自分の持つ最短記録のメッセージを送る。データは分散管理されている。

アルゴリズム B を導入することで、並列度が上がり、実行性能が向上する。しかし今度は、初期化ルーチンであるプロセスの生成部分がネックになる。

7 デバッギングツール

以上で述べてきた実例において、パフォーマンスデバッガの支援すべき点を考察する。

これらの場合に、対象とする部分が実行全体の中でどのような位置を占めるかを知ることが最初の課題である。全体の中でどの部分がクリティカルになっているかを知るためには、実行頻度だけでなく、他の部分に与える影響が問題である。この問題を解決するには、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望まれる。

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。

以上のような支援を実現するため、

- 問題点の絞り込み
- その問題点の解析

の2点について、それぞれデバッギングツールを構築するための課題点と、ツールをいかに利用してパフォーマンスデバッギングを行なうかについて述べる。

7.1 問題点の絞り込み

課題点 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、大量の実行情報から問題点を絞り込んでいかなければならない。そこで、プロファイリングを行なって問題点を抽出する必要がある。

実行データのプロファイリング プログラム中のどこに重みがあるかを知るために、実行データをわかりやすい形にしてプログラマに示すのがプロファイリングである。

逐次プログラムでは、プログラムの各部分にかかった時間の和がそのまま全体の時間であったので、従来の逐次プログラム用プロファイラはプロシージャ毎の時間を集計するだけで効果があった。しかし、逐次プログラムと並列プログラムとではプログラム中の「重み」の考え方が次のように異なってくる。

- 逐次プログラム: 直線的
  ループ回数、呼びだし回数を考慮しながら記述する
- 並列プログラム: 空間的
  他の部分へのデータ依存度を考慮しながら記述する

並列プログラムのプロファイリング情報は、一次元的な量の集まりでは表し切れない。プログラムのコントロールフロー、データフローに沿ったプロファイリングが必要となる。

パフォーマンスデバッガ Paf 我々は、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 を開発中である [5]。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。

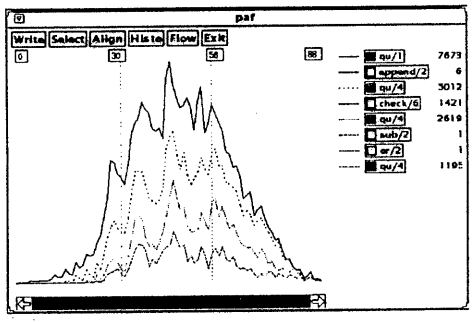

図 3: パフォーマンスデバッガ Paf

仮想時間による計測 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。

7.2 問題点の解析

課題点 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。効率化のためのプログラムの逐次化は、別の段階で行なわれる問題である。

7.2.1 タイムスタンプつき IO-Tree とその表示

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。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[10]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。

この入出力依存関係にプロファイラからのタイムスタンプをもとにした時間的推移を導入したタイムスタンプつきIO-Treeをプログラマに提示する。これは、存在する入出力それぞれにそれが行なわれた時刻を付加したものである。

このタイムスタンプつきIO-Treeを導入して、あるゴールの実行を、入出力の履歴を持ったプロセスとして以下のように表す。

$$Proc = \{Goal, Time\} : IOlist$$

$$Goal = predicate(V_1, \ldots, V_n)$$

$Goal$は生成されたゴールであり、互いにことなる変数$V_1, \ldots, V_n$を引数にもつ。$Time$は$Goal$が生成された時の時刻である。

入出力履歴$IOlist$は以下のような構造をしている。

$$\begin{aligned} IOlist &::= [IO|IOlist] \mid \square \\ IO &::= \{I, O\} \text{ -> } IOlist \\ I &::= [Unify|I] \mid \square \\ O &::= [Unify|O] \mid \square \\ Unify &::= \{Variable = Term, Time\} \end{aligned}$$

$\{I, O\}$ -> $IOlist$は、入力Iが存在するという条件で、出力Oおよび入出力$IOlist$が存在するという意味である。例えば、

```
{I1,O1} -> [{I2,O2},{I3,O3}]
```

は入力I1が存在する条件でO1が、I1,I2が存在する条件でO2が、I1,I3が存在する条件でO3が出力されることを表す。

単純な具体例として、appendの例をあげる。第1引数に入力があるたびに第3引数に出力が存在する。

```
append([H|X],Y,Z) :-
        Z = [H|Z1],append(X,Y,Z1).
append(□,Y,Z) :- Z = Y.

{[A,B,C],t0} :
   [{[{A = [D|E],t1}], [{C = [D|F],t2}]}
    -> [{[{E = [G|H],t3}],[{F = [G|I],t4}]}
        -> [{[{H = □,t5}],[{C = B,t6}]}]]]
```

入力Iの場合のUnifyの時刻Timeは、データTermを最初に参照した時刻を表し、出力Oの場合のUnifyの時刻Timeは、変数Variableにデータ Termを書き込んだ時刻を表す。この時刻は、仮想時刻を基本とするが、実時間を用いて表示することも考えられる。これによって、実際に実行してみた場合の情報からフィードバックを得ることもできる。

このようなタイムスタンプつきIO-Treeを、図4のように時間軸に対応して表示する。{I,O}を対になった矩形で表し、{I,O} -> IOlistで得られる構造を矢印を用いて表現している。モジュール間でデータのやりとりのある場合は、対応する入力/出力をつなぐ。

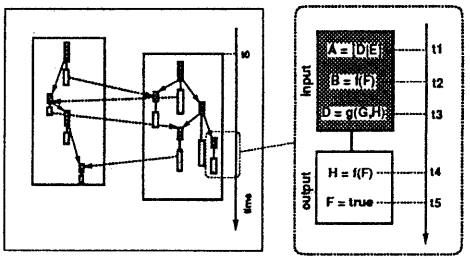

図4: タイムスタンプつきI/O-Treeの表示

この表示を観察することにより、プログラムの各モジュールがどのように関わっているかを知ることができる。どのゴールでモジュールの内外を分けるかは任意なので、抽象度が任意に変えられ、内部のサブモジュールも階層的に調べられる。各依存関係の組には、ソースコードである定義節が対応するので、プログラム中で問題となる部分に関係する部分を知ることができ、ソースレベルでのパフォーマンスデバッギングが可能になる。

7.2.2 IO-Treeを用いた問題点の分析

モジュール内部の依存関係 入力と出力それぞれについての時刻が仮想時間で表されている。この時間差が、データを参照してから結果を出力するまでの時間を表す。内部での不必要な依存関係が不必要な時間の遅れの原因となっている可能性がある。以下の二つのプログラムt1とt2はその例である。

```
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(A,B,C,D) :- C = □, B = D.

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

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

t1とt2では、外部とのデータの依存関係は同じである。しかし、t1ではa1が処理を終えるまでbの処理は行なわれないが、t2では並列に処理を行ないうる。より早い世代でデータが確定されるはずのものが、後の方の世代からの不必要な依存関係をうけて遅くなってしまった例である。

この場合では、タイムスタンプに仮想時間を用いているので、プログラムに内在する並列性を示している。これが実際に並列に実行されるかどうかは、実行環境のパラメータに基づいたスケジューリングなどの要因で決定される。タイムスタンプ

に実時間を用いる場合では、スケジューリングの調整などによる出力時刻の変化が観察できるであろう。

このような場合、モジュール内部にある依存関係を調べるには、さらに抽象度を落したサブモジュールに関して入出力の依存関係を調べる必要がある。

モジュールの入出力依存関係　表示された入出力依存関係を調べて不必要な依存関係を取り除くことにより、パフォーマンスを悪化させる要因を取り除くことができる。例えば、次のような依存関係があったとする。

```
{[I1,I2,I3],[O1,O2,O3]}
```

これは、O1,O2,O3 の出力をするには I1,I2,I3 すべての入力がなければならないことを意味する。実際には、O1 には O2 と O3 が、O2 には I3 が、O3 には I2 が不必要だとする。これらの出力がプログラム中でネックとなっている場合、プログラムを変更して次のようにすれば、その解消が期待できる。

```
{[I1],[O1]} -> [{[I2],[O2]},{[I3],[O3]}]
```

これにより、例えば O1 は I2 と I3 を待つことなしに出力できるようになる。

モジュール間の依存関係　IO-Tree の表示では、モジュール間のデータのやりとりが把握できる。モジュール間の通信にそもそも逐次性があり、その部分がネックになっている場合には、その部分のアルゴリズム全体の変更をする必要がある。例として、前述した最短経路探索問題のアルゴリズム A による実行を取り上げてみる。右側の矩形が searcher 全体を表し、左側

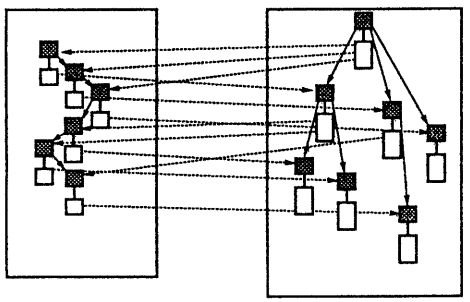

図 5: 最短経路探索問題のデータ依存関係の表示

の矩形が evaluator を表す。searcher は問題の経路にそって並列に探索しているが、evaluator はデータを一括管理しているので、それを検索するのにパイプライン的な並列度しか得られていない。searcher は evaluator から問い合わせの結果を受けとってからでないとその先を探索できないので、結局データを一括管理する部分がネックとなってパフォーマンスが得られないことがわかる。

8　おわりに

本論文では、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を分析し、これを支援するためのプログラミング環境に対しての要件を述べ、これを実現するための手法を提示した。

これからの課題としては、まず、各ツールが扱う実行モデルの詳細を検討しなければならない。さらにこれを実装し、必要に応じて諸機能の拡張を行なうことにより、実用になるパフォーマンスデバッガを開発・改良していく。実際のアプリケーション開発に適用してその有効性を評価する必要がある。

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] 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ッ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).

[3] 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).

[4] 舘村, 小池, 田中 : 並列論理型言語 Fleng のマルチウインドウデバッガ HyperDEBU, 情報処理学会論文誌 Vol.33 No.3, pp. 349-359 (1992).

[5] 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.

[6] H. Burkhart and R. Millen, *Performance-Measurement Tools in a Multiprocessor Environment*, IEEE Trans. Computers, Vol.38,No.5,May 1989,pp.725-737.

[7] T. E. Anderson and E. D. Lazowska : *Quartz : A Tool for Tuning Pararell Program Performance*, ACM SIGMETRIC,1990.

[8] R. H. Halstead, Jr. and D. A. Kranz, *A Replay Mechanism for Mostly Functional Parallel Programs*, Proc. Int. Symposium on Shared Memory Multiprocessing, 1991.

[9] T. Lehr, Z. Segall, D. F. Vrsalovic, E. Caplan, A. L. Chung and E. Fineman, *Visualizing Performance Debugging*, IEEE Computer Octover 1989,pp.38-51.

[10] 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).