プログラミング－言語・基礎・実践－ 18－1
（1994. 7. 21）

flengの動的粒度制御のための静的解析手法

中田 秀基, 小池 汎平, 田中 英彦
{nakada,koike,tanaka}@mtl.t.u-tokyo.ac.jp
東京大学 工学部

概要

高並列実行において重要なものの一つに粒度の制御がある。細粒度言語の場合には、動的状態に応じて実行単位を適切に融合して、粒度を荒くすることが重要である。本稿では、並列計算機 PIE64 上での Committed-Choice 型言語 fleng の実行における粒度制御手法を示し、そのために必要となる静的解析手法を紹介する。

Static analysis for dynamic granularity control of fleng

Hidemoto Nakada, Hanpei Koike, Hidehiko Tanaka
{nakada,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 achieve high-speed execution on parallel machines, execution granularity is one of the most important thing. **Fleng** is a fine-grained language and naive execution-granularity of fleng is too small.

In this paper, we show dynamic granularity control for committed-choice language **fleng** on parallel inference engine **PIE64**. We analyze programs statically and according to the analysis, transform the programs into two kind of programs; for high-loaded situation and low-loaded situation. We control execution granularity dynamically by switching the two codes.

1 はじめに

プログラムを並列に実行する際に問題になるものの一つに粒度がある。粒度は計算を実行する大きさであり、実行時に得られる並列性に大きな影響をおよぼす。一般に粒度が小さければ小さいほど、高並列な実行が可能になり、並列実行による恩恵を受けやすくなるが、分割された実行単位の間で実行を切替えるコストが大きくなり、これがオーバーヘッドになる。

ある計算を実行する際の最適な粒度は、すべてのプロセッサが常に稼働状態になる最大の粒度である。しかし、高並列な実行のためには、ある程度粒度を細かくして並列度を稼ぐ必要がある。最適な粒度は実行時の状況、すなわちプログラムの他の部分による負荷の大小に応じて変化する。

最適な粒度を実現する方法として、通常の粗粒度言語のデータフローを解析し、分割を行なう方法も研究されている。しかしこの手法では記号処理のように、計算対象に依存して動的に計算木の構造が変化する非定型的な問題に関しては、自動的に並列性を抽出することは難しく、したがって粒度の制御も難しい。

本稿では、Committed-Choice 型言語 fleng の並列推論エンジン PIE64 上での実行を例にとり、細粒度言語の粒度を合成して適切な粒度を得る方法について論じる。fleng は、単一代入変数を用いた細粒度の同期が行なえるため、その実行粒度は非常に細かい。このため、非定型的な問題に関しても容易に並列度を抽出することができるが、その反面細粒度であるためのオーバーヘッドが大きく効率的な実行のためには、粒度を粗くすることが必要となる。

このために、粒度の融合を行なうことが考えられる。しかし、fleng の各実行粒には同期が含まれる可能性があるため、安易な粒度融合を行なうとデッドロックを引き起こす可能性がある。このため、安全な粒度融合には同期を考慮に入れた静的解析が必要となる。

また、実行時の状態に応じて最適な粒度が変化するため、動的な粒度制御も必要となる。

2 章で、Committed-Choice 型言語 fleng を紹介し、fleng の実行粒度について述べる。3 章で、並列推論エンジン PIE64 を概説し、fleng の PIE64 での動的粒度制御法をしめす。4 章で、ゴール融合のための静的解析手法を示す。5 章で、我々の手法の予備評価の結果を示す。

2 fleng と PIE64

2.1 Committed-Choice 型言語 fleng

fleng は並列論理型言語の子孫である Committed-Choice 型言語の一つである。fleng の実行単位はゴールと呼ばれるものである。ゴールは、名前と引数を持つ。fleng のプログラムは、クローズと呼ばれる書き換えルールの集合である。クローズの名前は、ゴールの名前に対応する。クローズはヘッド部とボディー部からなり、ヘッド部にマッチしたゴールをボディー部のゴールへと書き換えるルールである。

fleng の変数は単一代入であり、未束縛な状態と束縛された状態の二つの状態を持つ。一度束縛されると未束縛な状態には戻らない。通信 / 同期は、単一代入変数を共有することで行なわれる。

fleng の実行モデルを図 1 に示す。fleng の実行は、計算対象となるゴールを任意に選択し、それを書き換えることで行なわれる。あるゴールが書き換え可能であるかどうかは、そのゴールの引数が書き換えに必要なだけ束縛されているかどうかに依存する。ゴールが実行の対象になった場合には、まず実行可能かどうかを調べ、実行が可能でなければサスペンドという処理を行なう。これはゴールをスケジュールの対象から外すことである。

サスペンドしているゴールは、その実行に必要な変数が束縛されると、再びスケジュールの対象となる。これをアクティベイトという。

2.2 fleng の実行粒度

fleng をナイーブに実装した場合、最小の実行単位は一つのゴールのリダクションになる。ゴールは、一種のコンティニュエーションであり、計算の継続に必要な情報、すなわち実行すべき計算と引数をメモリ上に退避したものだと考えることができる。

図 2 にゴールの実行の様子を示す。

ゴールの実行は大別して、3 つのフェイズからなる。

1. サスペンドチェックとパターンマッチ
2. システムゴール / ユニフィケーションの実行
3. ボディーゴールの作成とフォーク

まず、ヘッド部のサスペンドチェックとパターンマッチとが行なわれる。この時、必要なデータを参照し、束縛されていなければサスペンドする。他のゴール

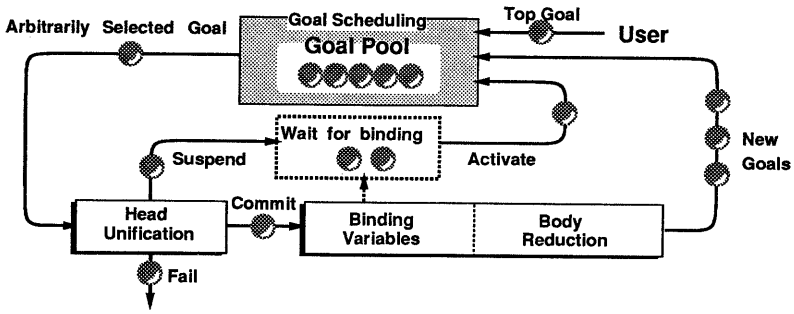

図 1: fleng の実行モデル

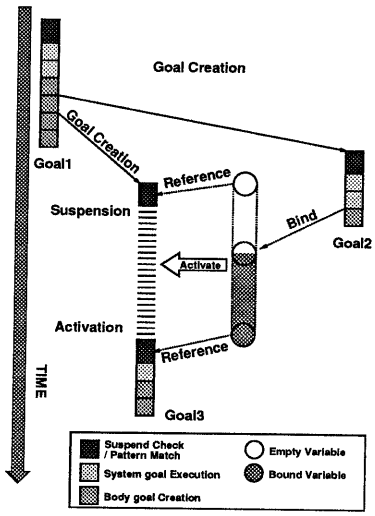

図 2: ゴールの実行

からのデータの受信はここで行なわれる。

このチェックに成功したら、次にシステムゴールの実行とユニフィケーションを行なう。このフェイズで変数に値を束縛するわけであるが、これが他のゴールへのデータの送信に相当する。もしその変数に対してサスペンドしていたゴールがあれば、この束縛によってアクティベイトされる。

最後にボディゴールを作成し、ゴールプールに戻す。

ここで、システムゴールとは数値演算などを行なう処理系に組み込みのものである。システムゴールは、必要なデータがすでに揃っていることを前提としているので、外部で同期をとってこの条件を保証する必要がある。例えば足し算を行なう述語は以下のように記述される。

```
add(#A, #B, R):- compute('+', A, B, R).
```

ヘッド部の引数についている記号 (#) は Non-Variable Annotation (以下 NVA) といい、その引数が何らかの具体化を行なわれている、という条件を示している。この NVA により数値演算を行なうシステム述語 compute に束縛されていない引数が渡されないことを保証している。

2.3 実行時のオーバーヘッド

fleng のゴール実行には以下のようなオーバーヘッドが存在する。

1. ゴール作成 (ゴールへのレジスタ退避) のオーバーヘッド
2. ゴールのキューイングのオーバーヘッド
3. ゴール転送のレイテンシ
4. ゴールの内容をレジスタに展開するオーバヘッド

1 と 5 を合わせたものがが通常のサブルーチン呼び出しのオーバーヘッドと同等であると考えられる。ただし、fleng は大域変数をもたないので通常の言語のサブルーチン呼び出しよりも退避する引数が多くなる傾向がある。

2 は、ゴールをゴールプールに蓄えたり、蓄えたゴールをスケジュールしたりするコストである。

3 は、ゴールを並列実行する場合にゴールをプロセッシングエレメント間で転送するコストである。2 は並列 / 逐次双方にかかるが、3 は並列実行時のみにかかる。

また、粒度とは本来は直接には関係ないが

5. サスペンド / アクティベイトのコスト

も関係してくる。サスペンド、アクティベイトはゴールの実行順によって生じるが、静的にスケジューリングすることである程度避けることができる [1]。粒度を大きくすることはスケジューリングの側面をもつので、このコストも低減できる。

さらに、実行粒度が小さいことによって

6. レジスタ・アロケーションが最適化されない

7. 命令スケジューリングの余地がない

というデメリットが存在する。

2.4 flengにおける粒度制御

flengの実行粒度を制御する手法として、複数のゴールを融合して一つのゴールとすることでひとつの実行のランレングスを伸ばすことが考えられる。

ここで問題になるのは、flengのゴールはすべて先頭部分に同期を持ちうることである。単に複数のゴールを一つのプロセッシング・エレメントに割り当て、連続して実行させようとしても、同期に失敗してしまうとサスペンド処理が生じてしまい、実行が分断されてしまいランレングスを伸ばす効果は得られない。したがって、ゴールを融合する際には同期が内部に含まれないようにする必要がある。

ゴールを融合させる方法には以下の二つが考えられる。

- 子ゴールどうしの融合
- 親ゴールと子ゴールの融合

以下前者をゴール連結、後者をゴール展開と呼ぶ。図3、4にそれぞれを示す。

ゴール連結 ゴール連結は、あるゴールの複数のサブゴールを一つのゴールとして呼び出す方法である。

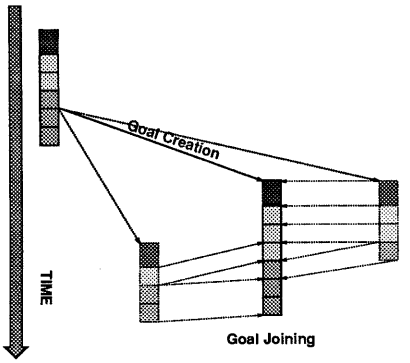

図 3: ゴール連結

この処理は以下のようにプログラムを変換することで実現する。

```
foo:- bar, baz,..
bar:- bar1.        =>   foo:- joined,..
baz:- baz1.             joined :- bar1, baz1.
```

ゴール連結により、前述の1,2,4のコストが低減できる。3に関しては、複数のサブゴールのレイテンシは重複可能なので、低減できない。また、サブゴールの間にデータの依存関係が存在する場合には、実際には逐次にしか処理できず、この場合にはさらに5のコストも避けられる。

ゴール展開 ゴール展開は、サブゴールを親ゴールのなかに展開してしまう方法である。

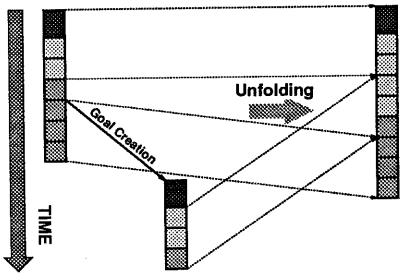

図 4: ゴール展開

この処理は以下のようにプログラムを変換することで実現する。

```
foo:- bar, baz ..
bar:- bar1,bar2.  =>  foo:- bar1,bar2,baz ..
```

この処理によって、前述の1-4のコストが低減できる。

2.5 ゴールの融合と並列性

ゴールの融合は、一般には並列性の低下を引き起こす可能性がある。しかし、上記コストの低減による高速化が、並列性の低下による速度の低下を上回る場合には、ゴールの融合が正当化される。

また、ゴール間にデータ依存関係が存在する場合には、実際にはそれらのゴールを同時に実行することができないため、それらを融合しても並列性に悪影響をあたえることはない。例えば、図2のGoal2とGoal3にはデータ依存関係があるため、これらを融合しても並列性は低下しない(図3)。

また、二つのゴールにデータ依存関係が存在せず、並列に実行できる場合にも、実際には逐次に実行した方が高速である場合もある。ゴールを融合し、逐次に実行した方がいいかどうかは、ゴールを他のプロセッシング・エレメントに送った場合の実行開始までのレイテンシ、そのゴールの粒度に依存する。

粒度制御をする際にはこれらのことを考慮に入れなければならない。

2.5.1 ゴール展開とオーバーヘッド

ゴール連結では連結されるゴール間にデータの依存関係が存在する場合には、並列性の低下は生じない。これに対して、ゴール展開では、最後のサブゴール以外を展開する際には確実に並列性の低下が生じるため、とくに慎重にならなければならない。

ここでは、あるクローズ

$$H : -B_1, B_2, B_3 ... B_n.$$

に対して、すべてのボディゴールの実行が終了するまでの時間を最小にすることを前提に、ボディーゴール B_i を展開するべきかを検討してみる。ボディゴールはすべてサスペンドすることなく実行が可能であると仮定する。

ボディーゴール B_j の生成 / 送出にかかる時間を T_{Gj}、これは前述のコストの 1 に相当する時間である。ボディーゴールを送出してから実際に実行されるまでの時間を T_L とする。こちらは、2,3,4 のコストに相当する。ここでボディゴールの実行時間をそれぞれ T_{Ei} とする。

ボディ B_i 以降の実行時間は、展開しなければ $\sum_{j=i}^{n} T_{Gj}$ で、ボディ B_i の本体が実行されるまでには、$T_L + T_{Ei}$ である。従って、全体の実行時間 $T_{nounfold}$ は、$max(\sum_{j=i}^{n} T_{Gj}, T_L + T_{Ei})$ である。

これに対して、展開した場合の実行時間 T_{unfold} は $T_{Ei} + \sum_{j=i+1}^{n} T_{Gj}$ である。これが $T_{nounfold}$ より小さければ、展開を実行して良いことになる。

$\sum_{j=i}^{n} T_{Gj} > T_L + T_{Ei}$ の場合は

$$\sum_{j=i}^{n} T_{Gj} > T_{Ei} + \sum_{j=i+1}^{n} T_{Gj}$$

すなわち

$$T_{Gi} > T_{Ei}$$

また $\sum_{j=i}^{n} T_{Gj} < T_L + T_{Ei}$ の場合は

$$T_L + T_{Ei} > T_{Ei} + \sum_{j=i+1}^{n} T_{Gj}$$

すなわち

$$T_L > \sum_{j=i+1}^{n} T_{Gj}$$

要約すれば、$T_{Gi} > T_{Ei}$ のとき及び、$\sum_{j=i+1}^{n} T_{Gj} < T_L$ のときには展開したほうがよいことになる。

T_{Gi},T_{Ei}, は、実行状況に依存しない一定値をとるのに対し、T_L は、計算機全体に対する負荷が上昇するにつれ大きくなると考えられる。従って、後者の条件は高負荷になるにつれ、容易になりたつようになることがわかる。

また、プログラム全体の並列性が十分大きく、他のプロセッシングエレメントすべてに十分な負荷が存在する場合には、ある程度並列性を低下させても実際の並列度には影響しない。この場合には並列実行できるゴールを融合させて構わない。

3 PIE64 における動的粒度制御

3.1 並列推論エンジン PIE64

PIE64 は、fleng の実行を念頭において開発された並列計算機である。PIE64 は 64 個のプロセッシング・エレメントを持つ。各エレメントは 3 段のクロスバネットワークで相互に結合している。

プロセッシング・エレメントは、3 種類のプロセッサをもっている。fleng の実行を行なうタグアーキテクチャをもつ専用のプロセッサ UNIRED (Unifier / Reducer)、通信 / 同期を専門に行なう NIP(Network Interface Processor)、並列計算にともなう雑多な仕事を請け負うマネージメント・プロセッサ (MP) を持っている。MP には SPARC を使用している。UNIRED は、レイテンシ隠蔽のために同時に 4 コンテクストを実行できる。これらのプロセッサは、コマンドバスと呼ばれるバスで密に結合している。

メモリは分散して実装されているが、UNIRED に対しては NIP によって分散共有メモリとなっている。また、NIP は単一代入変数による同期をハードウェアレベルでサポートしている。

3.2 PIE64 での fleng の実装

PIE64 は fleng の実装を意識して設計されたため、ナイーブな実装は非常に容易である。MP はランタイムカーネルを実行する。ランタイムカーネルは、ゴールのハンドリング、メモリ管理などを行なう。MP がこれらの仕事を行なうので、UNIRED で実行されるプログラムは実行のことだけを考慮すれば良い。

3.3 PIE64 での動的粒度制御

前章で述べた通り、最適な実行粒度は他のプロセッサにフォークした際のレイテンシに依存する。レイテンシは計算機全体の稼働率、ネットワークの閉塞率などに依存するがこれは結局プログラム全体の持

つ並列性に依存する。しかし、プログラムの並列性は実行時に与えられるデータによって大きく変化するので、静的に解析することは困難である。したがって、動的に粒度を制御する必要がある。しかしプログラムの並列性に従って連続的に粒度を制御することは、オーバーヘッドの観点から困難である。

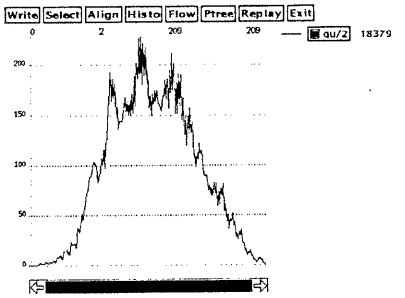

図 5: fleng プログラムの並列度

図5は、8クイーンの実行時の並列度を示したものである。fleng プログラムの実行時の並列度は一般に、このような形状をとることが多い。実行開始時、終了時の近辺は並列度が非常に低く、中盤では非常に高くなる。このように並列度の変動が低並列、高並列の両極端になることを仮定すれば、二つの典型的なコードを作成し、これを切替えることでほとんどの時刻においてほぼ最適な粒度での実行ができるはずである。

この考えに基づき、PIE64 では一つのプログラムに対して2つのコードを用意し、これらを実行時の状況に応じて切替えることによって粒度制御を行なう。

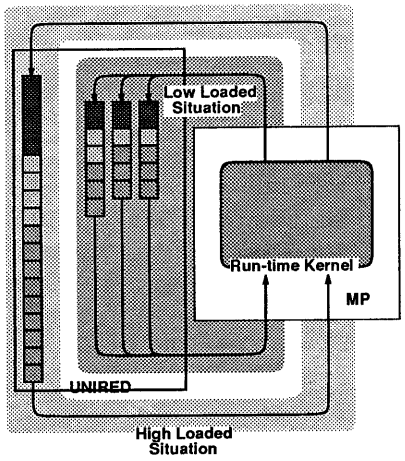

図 6: PIE における粒度制御

3.4 fleng コンパイラシステム

図7に fleng のコンパイラシステムの概要を示す。

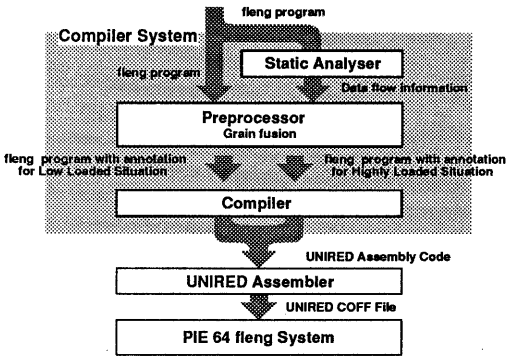

図 7: fleng コンパイラシステム

われわれのシステムは、スタティックアナライザ、プリプロセッサ、コンパイラ本体、アセンブラからなる。スタティックアナライザは、プログラムを静的に解析し粒度を融合することができる場所を探しだす。プリプロセッサはその情報に従って、アノテーションつきのコードを生成する。このとき、粒度の上限を2種類与えることで、高負荷時、低負荷時の2つのコードを得る。コンパイラはアノテーションにしたがってそれぞれのコードをコンパイルし、アセンブリ言語のコードを出力する。得られた2つのコードをマージし、双方のコードを含んだコードを作成し、これをアセンブラにかけ、最終的なオブジェクトファイルを得る。これを PIE64 上の fleng 処理系が読み込み実行する。

スタティックアナライザ、プリプロセッサ、コンパイラ本体は fleng 自身で書かれている。

4 ゴール融合のため静的解析

4.1 プログラム変換可能な条件

2章で述べた通り、ゴールを融合する際に問題になるのは、ゴールの先頭での同期の取り扱いである。

ある種の述語はヘッド部に同期をもたず、これらの述語のゴールは展開、連結してもコンテクストのなかで同期をとってしまう可能性はない。例えば、`test(A):-...` という述語は、引数がどのような状態でもサスペンドを生じない。しかし、この種の述語は非常に稀であり、この種の述語だけを扱っているのでは、祖粒度にまで粒融合を進めることはできない。

ヘッド部に同期を持つゴールが、サスペンドせずに実行できるかどうかは、そのゴールが実行される環境に依存する。同期が記述されていても、実行環境が特定できれば、サスペンドが生じるかどうかが特定でき、ゴール融合が可能になる。例えば、
`test([A]):-..` という述語は、第一引数に未束縛の値を与えられるとサスペンドする。しかし、この述語に対して、

```
test0:- test([1]).
```

という呼び出し方をした場合には、このゴールはけしてサスペンドしない。なぜなら、第一引数として、確実にリストセルが渡されるからである。従って、この場合 test の呼びだしはゴール融合の対象となる。

このように、実行される環境を静的に解析、特定することでゴールの融合の対象になる部分を抽出する。

4.2 fleng の形式的意味論

本節では、[2] に従って、fleng のセマンティクスを定義し、粒融合が可能な場合の検出とプログラム変換について述べる。

Var を変数の集合、$Func$ を関数記号の集合、$Term$ を Var、$Func$ 上で定義されるすべての項の集合とする。束縛 θ は Var から $Term$ へのマッピングとする。

あるプログラム (述語集合) のセマンティクスを、そのプログラムが許す入出力履歴であるガード付ストリームの集合で表す。ガード付ストリームは、ガードつき単一化と呼ばれるものの集合のうち fleng プログラムとして意味をもつものである。ガード付単一化とは、束縛の対で、右辺の束縛が行なわれるための必要条件を左辺の束縛に書いたものである。$<\sigma|\theta>$ のように書く。すべてのガード付単一化の集合を Gu とする。この場合、σ が行なわれた後、θ が行なわれることを示す。

例えば、次の述語

```
test(a, H2):- H2 = b.
```

で実行される可能性のある入出力履歴は $\{<H1 = a|H2 = b>\}$ というガード付ストリームであらわされる。

次に、ガード付ストリームの合成演算同期つきマージを導入する。同期つきマージは、$||$ で表され、ガード付ストリーム $gu_i, ..., gu_n$ に対して、新たなガード付ストリーム $(gu_i||...||gu_n)$ を与える演算である。

この同期付マージを用いて、ゴールのマルチセット B_0 に対してプログラム P の意味関数 $T_P(B_0)$ を定義できる。この T_P は連続であるため、Prolog と同様の手法で不動点セマンティクスが定義できる。

4.3 ゴール融合の条件

ゴールの融合は上記のように定義されたプログラムのセマンティクスを保存しながら行なわなければならない。

4.3.1 ゴール連結の条件

対象となる親ゴールのガード条件を θ_g、親ゴールのシステムゴールにより行なわれる束縛を θ_{sys}、サブゴールのガードを θ_{g1}、θ_{g2}、サブゴールのシステムゴールにより行なわれる束縛を θ_{sys1}、θ_{sys2} とする。

これらのサブゴールが連結できるのは以下の条件が成り立つ時である。

$$\theta_g \cup \theta_{sys} \cup \theta_{g1} \cup \theta_{sys1} \supseteq \theta_{g2}$$

以下の例の foo、bar がこの条件を満たす。

```
test(A, B):- foo(A, C), bar(C, B).
foo(a, C):-  C = c.
bar(c, B):-  B = b.
```

したがってゴール連結が可能で、

```
test(A, B):- baz(A, B, C).
baz(a, B, C):-  C = c, B = b.
```

と書きかえることができる。

4.3.2 ゴール展開の条件

対象となる親ゴールのガード条件を θ_g、親ゴールのシステムゴールにより行なわれる束縛を θ_{sys}、親ゴールが実現可能なガードつきストリーム $Gs0$ を

$$<\theta_1^+|\theta_1^->, ..., <\theta_n^+|\theta_n^->$$

とする。また、サブゴールのガードをそれぞれ θ_{gi}、サブゴールのシステムゴールにより行なわれる束縛を θ_{sysi} とする。

サブゴール Gi の展開が可能なのは、以下のどちらかの条件を満たす場合である。

1. $\theta_g \cup \theta_{sys} \supseteq \theta_{gi}$
2. $\forall j(\theta_j^+ \supseteq (\theta_{gi} \cup \theta_g) \vee \theta_j^- = \theta_{sys})$

下の例は、条件 1 を満たす。

```
foo:- bar(B), B = a.
bar(a):- baz(b).
```

このとき $\theta_g = \{\}$, $\theta_{sys} = \{B = a\}$, $\theta_{gi} = \{B = a\}$ であるから、明らかに条件 1 を満たす。

下の例は、条件2を満たす。

```
test(A, x, B):- baz(A, B, C).
baz(a, B, C):-  C = c, B = b.
```

このとき、$Gs0 = \{< A = a, X = x | C = c, B = b >\}$、$\theta_{g1} = \{A = a\}$、$\theta_g = \{X = x\}$ である。これは、以下のように展開することができる。

```
test(a, x, B):- C = c, B = b.
```

下の例は、条件を満たさない例である。

```
test(A, B, E):- baz(A, B, C), boo(E).
baz(a, B, C):-  C = c, B = b.
boo(E):- E = e.
```

このとき、$Gs0 = \{< A = a, | C = c, B = b >< \{\} | E = e >\}$、$\theta_{g1} = \{A = a\}$、$\theta_g = \{\}$ で、条件を満たさないことがわかる。もしこれを展開すると

```
test(a, B, E):- C = c, B = b, boo(E).
boo(E):- E = e.
```

となるが、この場合 $Gs0$ は $\{< A = a, | C = c, B = b, E = e >\}$ となりプログラムのセマンティクスが変化してしまうことがわかる。

このプログラムは、boo がE に束縛する値が、外部で第一引数のA に束縛されているとすると、本来デッドロックを起こさないプログラムを、デッドロックを生じるプログラムに書き直してしまったことになる。

5 予備評価

われわれの手法を確認するために、解析結果に基づいてプログラム変換を手でおこない、その効果とプログラムの並列性の関係を調べた。

5.1 対象プログラム

3階層の8進木のリーフを足しあわせるプログラムを対象とした。このプログラムに関して、解析結果に基づき、連結、展開を用いて、粒度を調整し5種類のプログラムを得た。それぞれの粒度は 加算がそれぞれ 1, 7, 15, 31, 63 個となった。

この操作をバリア同期をとりながら、100回繰り返すものを、並列に幾つか動かすことによって、任意の並列性のプログラムを合成し、実験に用いた。

5.2 実験環境

実験はPIE64 上のインタプリタで行なった。インタプリタでの実行ではレイテンシと実行時間の比率が、コンパイラでの実行と大きく異なるが、定性的なデータを得るためには十分である。プロセッサは50 台を用いて実行した。

5.3 結果

実験の結果を図8に示す。

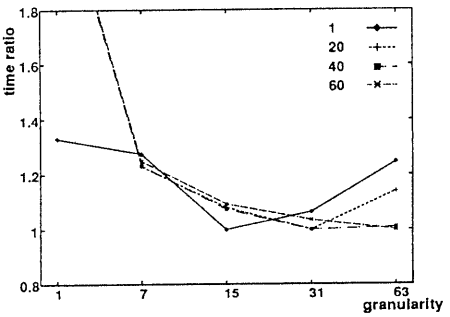

図 8: プログラムの並列性と最適粒度

横軸にプログラムの粒度、縦軸に最短実行時間で正規化した実行時間をとっている。それぞれのグラフは、プログラムを並列に動かす数を調整することで、それぞれ異なる並列性での状態を示している。プログラムの並列性が低い場合には、最適な粒度は15 近辺である。プログラムの並列性が上昇するに従って、最適な粒度が上昇することがわかる。

6 おわりに

PIE64 での fleng 実行の動的粒度制御のための静的解析手法について述べた。また、プログラムの並列性と最適な実行粒度の関係を考察し、実機を用いた実験で確認した。

当面の課題としては、プログラム変換の自動化があげられる。

また、本稿では、粒度の調節手法として、プログラム変換のみを考察したが、他にもさまざまな実現が考えられる。今後他の手法も検討していく予定である。

参考文献

[1] Hidemoto NAKADA, Takuya ARAKI, Hanpei KOIKE, and Hidehiko TANAKA. A Fleng Compiler for PIE64. In *Proceeding of PACT '94, To appear*, Aug. 1994.

[2] Kazunori Ueda and Masaki Murakami. Formal Semantics of Flat GHC. In 平成元年 電気・情報関連学会連合大会, 1989.