細粒度高並列言語flengの高性能な処理系構築に関する研究

中田秀善

①

細粒度高並列言語 fleng の高性能な処理系構築に関する研究

中 田 秀 基

概要

細粒度高並列言語は、問題に内在するすべての並列度を抽出できるため、非定型的な問題を高並列計算機上で実行するためのプログラミング言語として有望である。しかし細粒度高並列言語には、データ / プロセスなどの負荷をいかに分散させるかという問題、プロセスをどのような順番で実行するかというスケジューリング問題、といった並列言語に共通の問題点に加え、コンテクスト切替のオーバーヘッドという細粒度言語特有の問題点がある。これらの問題を解決する高性能な処理系実装の手法を確立することが課題になっている。

本研究では、並列推論エンジン PIE64 上への Committed-Choice 型言語 fleng の高性能な処理系実装を目的とし、上述の問題を解決した言語処理系方式の提案と実装を行なった。

上述の問題の特徴は、最適な対応が実行時の計算機全体における負荷レベルによって著しく異なるという点である。ある状態に対して最適にチューニングを行なうと別の状態においては逆にオーバーヘッドが大きくなってしまう可能性がある。定型的な問題に対しては、コンパイル時に負荷状態を予想することがある程度は可能であるが、知識処理に代表されるような非定型的すなわち実行状態が動的に変化するような問題に対しては、予測は非常に困難である。したがって、細粒度高並列言語の高効率な実行のためには、コンパイル時の静的な最適化だけでなく、実行時の計算機全体の状態に依存する動的な制御を行なうことが不可欠である。

本研究では、fleng プログラムの実行状態が、2つの典型的な状態、すなわち、十分な並列度がある場合と完全に並列度が不足している状態のいずれかに属している時間が長いことに着目した。これらの典型的な2つの状態に最適化した実行コードを動的に切替えることで、小さい動的制御のオーバーヘッドで、大部分の実行時間において最適な実行を行なうことが可能になる。

並列度が不足している状態すなわち低負荷時には、プロセッサの稼働率を最大にするために、得られる並列度を基本的にはすべて用いなければならない。このとき重要なのは、プログラムの字面上は並列度があるように見えながら、実際には並列度がない部分を識別することである。逐次にしか実行できない部分を単一のプロセッサに割り当てることでプロセッサ稼働率とメモリ参照の局所性を両立することができる。

十分な並列度がある状態すなわち高負荷時には、余分な並列度があるため、実行粒度を向上させ、逐次に実行する部分を大きくすることでコンテクストスイッチのオーバヘッドを減少させることが重要である。粒度制御には、動的制御と静的制御の二つの手法が考えられる。これらはそれぞれ異なる特徴を持つので相補的に用いることができる。本研究ではプログラム変換による静的制御を行なった。プログラム変換は同期の移動をともなうため、デッドロックの可能性をはらんでいる。本研究では、プログラムの表示的意味を定義し、意味を変更しない変換だけを用いることで、デッドロックの危険のない粒度制御を行なった。

これらの手法を用いた処理系を実機上で評価し、手法の有効性を確認した。また、粒度制御に関しては静的粒度調整の評価とともに、動的粒度制御と組み合わせての評価も行なった。

目次

図目次

表目次

第 1 章

序論

1.1 背景

プログラムを高速実行する方法として高並列に実行することが考えられる。昨今では商用の高並列計算機の市販も行なわれるようになったが、普及しているとはいいがたい。この原因の一つとして、並列計算機上でのソフトウェアの記述法が確立していないことがあると考えられる。

並列実行のためのプログラミング手法としては、大別して逐次言語の自動並列化による方法と、並列言語による方法の二つが考えられる。逐次言語の自動並列化は、科学技術計算などの特定の分野ではすでに実用化段階に入っており、一定の成果を収めている。しかし、これらの多くは 数値演算を主目的とする言語 Fortran を対象としており、記号処理をはじめとする非定型的なプログラムを記述することは難しい。

並列言語は大別すると、データ並列言語と制御並列言語に分けられる。データ並列言語には、やはり数値演算を対象とするものが多い。これは、数値演算以外の分野ではデータ並列性を抽出することが難しいためである。制御並列言語には、並行オブジェクトや future などの明示的な実行モジュールを並列実行の単位とする中、粗粒度の並列言語と、並列関数型言語や Committed-Choice 型言語などの細粒度並列言語がある。中、祖粒度の制御並列言語は、効率的には有利であるが、データのコンシステンシの維持がプログラマの責任である場合が多く、これがプログラミングを困難にしている。また、実行粒度が粗いために十分な並列度の抽出は困難である。

これに対して Committed-Choice 型言語や 並列関数型言語などの細粒度の制御並列言語

では、すべてのプログラムの部分が並列に動作することを基本としており、構造メモリを用いた細粒度の同期機構により、細粒度でしかも容易な並列記述を実現する。これらの特徴によって、自動的に並列性の抽出を行なうため、問題のもつ並列性をすべて抽出することができ、高並列実行に最も適していると考えられる。

このように高並列実行に適した細粒度高並列言語に関しては、従来から、並列実装の研究がなされてきた。これは、細粒度並列言語を高並列に実行すること自体は容易であるので、並列実装によって高性能の処理系が期待されたためである。Committed-Choice 型言語は、日本の第五世代言語プロジェクトで採用され、専用マシンとともに開発、実装された。また、関数型言語はデータフローマシンのための言語として、開発、実装が行なわれた。その結果、細粒度並列言語を高性能に実装するのは、そう簡単なことではない、という結論が得られた。

その理由として以下が考えられる。一つは、細粒度並列言語の言語構造そのものが逐次言語よりも非効率的であるからである。これは並列実行を実現するため、あらゆる場所に同期点を埋め込んであるためである。このため、一般に、細粒度並列言語で記述したものを逐次実行すると、逐次言語で記述したものよりも遅くなる。また、仕事をプロセッサ間で均等に分配することが困難であることもポイントである。仕事の分配が不均衡だと暇なプロセッサが生じてしまい、プロセッサ稼働率が低下するため、高速実行はできない。もう一つのポイントとしてプロセッサが実際に意味のある仕事をしている時間の割合いがある。この割合いをここで実効率と呼ぶ。各々のプロセッサの実効率が低ければ、見かけの並列度が高くても有効な計算の並列度は低下していることになり、実行速度の向上は得られない。実効率を低下させる要因はさまざまである。例えば、リモートのメモリに対する参照には一定のレイテンシがかかるが、このレイテンシの間はプロセッサは有効に稼働していない。また、同期の失敗にともなう処理も本質的な計算ではなく、実効率を低下させる原因になる。

従来の研究では、これらの問題点はあまり考察されず、素朴に言語モデルを忠実にハードウェアで実現しており、その結果十分な速度向上は得られなかった。このことから、これらの問題をハードウェアのみで解決することは不可能である、という知見が得られた。この反省に基づき、コンパイラとハードウェアの役割分担を変更する研究は、関数型言語で広く行なわれつつある。これは、ハードウェアは最低限の同期機構のみを提供し、あとはコンパイラがデータフローグラフを通常のプロセッサと、同期機構で実行されるようにコンパイル

するという手法である。しかしこれらの研究では、稼働率や実効率に関する考察は不十分のように思われる。また、Committed-Choice 型言語に関してはこの種の研究は行なわれていない。

問題を整理してみよう。細粒度並列言語を高並列に実行するためには、同期点を減少させることと、プロセッサの稼働率と実効率を向上させることが必要となる。同期点を減少させるためにはプログラムの実行粒度を大きくすることが考えられる。プロセッサ稼働率を向上させるためには、負荷の分配を均等に行なうことが必要である。プロセッサ実効率を向上させるためには、メモリ参照の局所性を向上させることと、同期の失敗を減少させることが重要になる。これらをまとめると以下の三項目をトレードオフをとりながら最適に行なうことが重要であるということになる。

- 実行粒度制御
- 負荷分散
- スケジューリング

これらの間のトレードオフのポイントは実行時の負荷状態に依存して大きく変化する。このため、何らかの形で負荷状態を仮定しなければ、適切な最適化はできない。しかし、非定型的なアプリケーションを対象とすると場合には、実行時の負荷状態を静的に推定することは困難であり、コンパイル時の最適化のみで、適切な実行を行なうことはできない。

コンパイラによる静的な最適化と、実行時系による動的な制御、ハードウェアによるサポートが、相補的に機能することによって、はじめて細粒度並列言語の高効率な実装が可能になるのである。

Committed-Choice 型言語を高速に実行するためには、これらの 3 つの役割分担の切りわけと、実行時処理系のための実行手法、静的最適化のためのコンパイル手法の確立が重要な課題となる。

1.2 本研究の目的

本研究では、細粒度高並列言語の一つである Committed-Choice 型言語 fleng の効率的実装手法を確立することをめざし、並列推論エンジン PIE64 のハードウェアサポートを前

提とした実装手法の提案と、その手法に対応したコンパイラ系の実装を行なう。

1.3 本論文の構成

第 2 章では本研究で用いる言語である Committed-Choice 型言語を詳説し、従来行なわれた実装手法を概説する。第 3 章では本研究で実装のターゲットとなった並列推論エンジン PIE64 について述べる。また、PIE64 上の fleng の実行時処理系について説明する。第 4 章では fleng の実行の様子を詳説し、効率的な実装に要求される点を整理する。第 5 章では 本研究で構成したコンパイラシステムについて述べる。第 6 章では粒度制御のバックグラウンドとなる fleng の表示的意味論について述べ、粒度調整プログラムについて解説する。第 7 章では本研究で提案した手法を PIE64 で評価した結果を示す。第 8 章では他の研究との関連に関して考察を行なう. 第 9 章は結論である。

第 2 章

Committed-Choice 型言語 とその実装

本章では本研究の対象である Committed-Choice 型言語 fleng に関して述べる。まず、一般の Committed-Choice 型言語に関して述べ、次に fleng に関して述べる。さらに、Committed-Choice 型言語の従来の実装について述べる。

2.1 Committed-Choice 型言語

Committed-Choice 型言語は並列論理型言語 [58] の子孫で、論理型言語としての側面より、操作的な側面に着目して設計された言語である。代表的なものに Flat GHC[67]、Flat GHC の 実用化版である KL1[23]、PARLOG[24] がある。

Committed-Choice 型言語の実行は、ゴールと呼ばれる実行フレームを書き換えルールにしたがって書き換えることで行なわれる。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。同名のクローズの集合を述語と呼ぶ。クローズはヘッド部、ガード部とボディー部からなり、ガード条件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。

代表的な Committed-Choice 型言語の一つ KL1 を例にとって説明しよう。KL1 のクローズの一般形は以下のような形をしている。

$$H : -G_1, .., G_n | B_1, .., B_m.$$

ここで、G_1 から G_n をガードゴール、 B_1 から B_m をボディーゴールという。あるゴールが、ガード条件にマッチするとは、ゴールがヘッド部 H とユニフィケーション可能であり

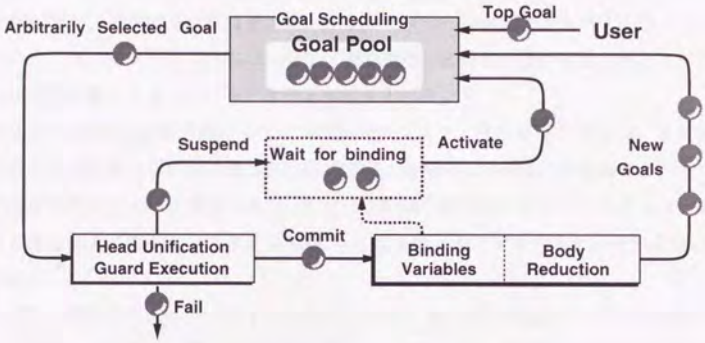


図 2.1: Committed-Choice 型言語の実行モデル

かつ、ユニフィケーションした状態ですべてのガードゴールの実行が失敗しない場合をいう。

ここで注意しなければならないのは、ガードゴールに用いることができるのは組み込み述語だけであることである。これは、ガードゴールに一般の述語の記述をゆるすと、ガードゴールの失敗に備えて環境を複製しなければならず、処理系が著しく複雑になるためである。この制約によって、処理系は単一の環境のみをサポートするだけで済むことになる。これが、 Flat GHC の 'Flat' の由来である。

Committed-Choice 型言語の特徴の一つに単一代入変数がある。単一代入変数は未束縛な状態と束縛された状態の二つの状態を持ち、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。送信側は、単一代入変数に対して値を代入する。受信側は単一代入変数から読み出しを行なうが、値の書き込みが終了していない場合には、読み出しはブロックする。この機構によって、同期も実現される。

Committed-Choice 型言語の実行モデルを図 2.1に示す。 Committed-Choice 型言語の実行は、ゴールプールにある複数のゴールから、書き換えの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を行なうのに必要なだけ、束縛されているかどうかによる。書き

換えの対象を選ぶと、まずそのゴールが書き換え可能かどうかを調べる。可能であれば書き換えを実行する。可能でなければそのゴールをスケジュールの対象から外す。これをサスペンドという。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。

Committed-Choice 型言語のもう一つの特徴にユニフィケーションがある。これは、論理型言語から受け継いだ非常に強力な言語機構で、基本的な意味は二つのものが同一になるように操作を行なうという意味である。例えば、変数と値をユニファイすることは、変数に値を代入することに等しい。たとえば変数 A と値 1 をユニファイすると、それ以降、A は 1 になる。

また構造体同士をユニファイするということは、構造体の中身どうしを再帰的にユニファイすることによって、構造体全体同士が同一になるようにすることである。例えば、リストセル `[A | b]` と `[a | B]` をユニファイすることを考えてみよう。この操作は、A と a, b と B をユニファイすることに帰着する。この結果として、両者は、`[a | b]` という同じ構造になる。ここで注意するべきことは、値の束縛が双方向に行なわれるということである。通常の変数に対する代入では、値の移動は片方向であるが、ユニフィケーションでは双方向に値が移動する。従って演算に方向性がない。

さらに変数同士のユニフィケーションも行なうことができる。これは将来それぞれの変数に束縛される値同士をユニファイするということである。単一代入変数と変数同士のユニフィケーションは非常に強力な枠組であり、複雑なデータ依存関係が容易に記述できる。

2.1.1 述語の実行

Committed-Choice 型言語の最小の実行スレッドは述語による一つのゴールのリダクションである。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。

述語の実行は大別して、以下の 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、ガードゴールの実行
2. システム述語の実行
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。さらにガードゴールを実行する。ガードゴールの実行が失敗すれば、他のクローズでのチェックを行なう。ガードゴールの実行が成功するとコミットする。

このチェックに成功したら、次にボディのうちシステム述語であるものを実行する。ボディ部に複数のシステム述語がある場合には左優先で実行する。システム述語は、ユニフィケーションや数値演算などの基本的なもので、変数への値の束縛を行なう。また、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。

最後にボディ部のゴールを新たに作成し、ゴールプールに戻す。

2.2 Committed-Choice 型言語 fleng

fleng[48] は Committed-Choice 型言語 の一種である。他の Committed-Choice 型言語との大きな違いは、ガードゴールが存在せず、ヘッドのパターンのみでガードを構成する点である。このため、KL1 ではかろうじてガード部のみで意味を持っていた述語の成功、失敗の概念が完全に消失している。この結果 fleng の言語仕様は、並列言語としてより整理され、コンパクトでエレガントなものとなっている。このため、ナイーブな解釈系の実装は非常に容易になっている。

fleng のプログラミングに用いる型を表 2.1に示す。

fleng は言語仕様を縮小したため、記述力が低下しているが if マクロなどでそれを補っている。これらのマクロは、実行時に動的に展開、もしくは実行前に静的に展開して、実行される。

fleng の if マクロ fleng の if macro は下のシンタックスを持つ。

condition -> *then-part* ; *else-part*

例をあげてみよう。下の KL1 プログラムは第一引数と第二引数の最大値を第三引数に返すものである。

```
max(A, B, C):- A >= B | C = A.
max(A, B, C):- A <  B | C = B.
```

表 2.1: fleng の型

型名	表記	例
整数	数字	1, 3
浮動点小数	数字	1.0, 3.2
シンボル	大文字以外で始まる文字列	a, sys
	またはシングルクォートで括った任意の文字列	'TEST' '::'
変数	大文字で始まる文字列	A, B
リスト	かぎ括弧 ([]) で括った列	[A, B, C]
	最後のデリミタを | にすると、	
	最後の要素をセルの cdr 部に直接入れる	[A, B | D]
ファンクタ	ファンクタシンボル につづく括弧で括った列	test(a, b, c)
文字列	ダブルクォートで括った文字列	"test", "abc"
	内部的には、ASCII コードのリストとなる	

これを fleng で記述すると以下のようになる。

```
max(A, B, C):-
    ((A >= B)->
          C = A
    ;     C = B).
```

このプログラムは下のように展開されて実行される。

```
max(A, B, C):-
     eval(A >= B, R), $1$max(R, A, B, C).
$1$max(true, A, B, C):- C = A.
$1$max(false,A, B, C):- C = B.
```

このように fleng はガードゴールを省略することで失った記述力をマクロを用いて補っている。

2.2.1 システム述語と Non-Variable Annotation

fleng のプログラムは最終的には、システム述語の集合に展開される。fleng のシステム述語には、大別して unify と compute がある。 unify は ユニフィケーションを行なう述語であり、compute は雑多な数値演算、操作を行なうものである。'=' は、unify の省略記法である。以下にいくつかの例を示す。

```
A = B.                  % A と B とをユニフィケーションする。
unify(R, A, B).         % A と B とをユニフィケーションし、成功したかどうかを R
に返す
compute('+', A, B, R). % A と B との加算を行ない、結果を R に返す
```

fleng のシステム述語は同期機構を持たない。すなわち、述語の引数が束縛されている必要がある場合でもそのチェックを行なわずに実行する。unify は、束縛されている必要はないので問題はないが、compute の引数は束縛されている必要があるので外部でそれを保証する必要がある。

このための記法が Non-Variable Annotation (NVA) と呼ばれる記法で、'#' であらわす。NVA はヘッド部で変数に対して用い、NVA が付加された変数が束縛されている、というガード条件を表す。この記法を用いると、例えば加算を行なう述語 add は以下のように書ける。

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

fleng では、ライブラリレベルで NVA を用いて安全な算術演算を一般のユーザに提供している。したがって、一般のユーザは直接 compute を使うことはない。

2.2.2 fleng のプログラミングスタイル

fleng のプログラミングは、変数をゴールが共有する形で記述される。しかし、変数は単一代入であり、状態を持つことができないため、ある程度以上の粒度のデータを保持するには向いていない。このために行なわれるプログラミングスタイルが、プロセス指向プログラミングと呼ばれるものである [35]。これは、再帰的に自分自身を呼び出すゴールを永続的なプロセスとみなして、このプロセスをデータとみなし、リストを用いたストリームを通じてアクセスするものである。このスタイルでは、プロセスは並行オブジェクトのように働く。このようなプロセスを明示的に並行オブジェクトして記述する試みもなされている [84, 85, 91, 74]。

例をあげてみてみよう。下のプログラムはストリームに到着するメッセージに応じて動作するスタックである。

```
stack([push(A)|Rest], S):-
      stack(Rest, [A|S]).
stack([pop(A)|Rest], [B|S]):-
      A = B,
      stack(Rest, S).
```

第一引数が、ストリームに見立てたリストセルである。car 部からメッセージをとりだし、それに応じた動作を行なっている。cdr 部が次のストリームになる。

プロセス指向プログラミングの最大の利点は、データへの非決定的な順序での書き込み、呼びだしが可能である点である。それには、プロセスへの入力ストリームをマージャーと呼ばれる特殊なプロセスで複数の入力ストリームに分割して分配すれば良い。マージャーは、非決定的にストリームをマージしてプロセスに送るので、プロセスが管理する情報への書き込み、読みだしが非決定的な順序で実行されることになる。Committed-Choice 型言語の通常の変数は単一代入であるから、通常の変数を用いてこの種のことを実現することはできない。

このように Committed-Choice 型言語においては、粗粒度並列の記述はプロセス + ストリームで、細粒度並列の記述はゴール + 変数で行なわれる。粗粒度から細粒度までを一つの言語の枠組でスマートに表現できるのが Committed-Choice 型言語の特徴の一つである。

2.3 従来の Committed-Choice 型言語の実装

2.3.1 KL1 on PIM

代表的な Committed-Choice 型言語のひとつである KL1 は、並列推論マシン PIM[64] 上に実装されている [23, 98]。ここでは、PIM 上の KL1 の実装に関してとくに負荷分散の手法に重点をおいて述べる。

PIM の構成 PIM には PIM/p, PIM/m, PIM/c, PIM/i, PIM/k があり、それぞれハードウェア構成が異なる。PIM/m は、メッシュ状のネットワークに PE が配置されている。メモリ構成は完全に分散メモリとなっている。PIM/m 以外の PIM は、基本的に 8 台の PE からなるクラスタをネットワークで接続した構成になっている。クラスタ間のネットワークの構造は各々異なるが、クラスタ内はメモリをバスで共有する形式になっている。またクラスタ間では、メモリは分散となっている。

VPIM での負荷分散 PIM/m 以外の PIM は、KL1 の抽象言語 KL1-B[38] を実行する抽象マシン VPIM (Virtual PIM)[79] を実装している。KL1-B は、WAM[73] に似た抽象言語である。

VPIM のメモリシステムは、クラスタ内ではグローバルアドレスを用いている。クラスタ間では、輸出 / 入テーブルを用いて管理している。つまり、クラスタ外へのメモリ参照は、ローカルな輸出テーブルへの参照になっており、輸出テーブルには他のクラスタの輸入テーブルの参照が書かれている。さらにその輸入テーブルの中に目的のメモリへの参照がある。したがって、クラスタ外へのメモリ参照は、2 段階に間接的になっているわけである。このような構成になっているのは、クラスタごとにローカルな GC を可能にするためである。

VPIM では、クラスタ内とクラスタ外で負荷分散の方法が異なる。これは、PIM のメモリ構成が、クラスタ内のメモリアクセスが共有メモリであるためレイテンシがほとんど問題にならないのに対して、クラスタ間ではメモリアクセスが非常に遅いという heterogeneous な 構成であるためである。

クラスタ内部では、PE ごとに独立なキューをもち、それぞれ独立に実行を行なう。キューの中身がなくなると、同じクラスタ内の他の PE に対して要求を出す。この要求にはじめに気付いた PE が、ゴールキューの先頭のゴール、すなわち次に実行されるべきゴールをわけ与える。この方式では、キューにゴールを持っている PE が、暇な PE によってゴールの分配という比較的負荷の高い仕事を加えられてしまうことになり、実行性能が低下してしまう。これに対して PIM/p では、ゴールの分配の負荷を減らすために、各 PE にゴールキューを 2 つずつ持たせ、片方を分配用とすることで負荷を減らしている [96]。しかし、これも結局要求駆動型の負荷分配であり、負荷が均等になるようにはなっていないため、ある PE が繰り返し負荷要求を出すような実行になることがあり、十分であるとはいえない。また、ゴールキューの先頭のゴールは、計算木の葉に近いゴールである可能性が高い。このため、このゴールを送ってもすぐに実行を終了してしまい、負荷の不均衡が修正されない場合が多いと思われる。

クラスタ間の負荷分散は、基本的にユーザーに任されている。VPIM では、他の PE への参照のレイテンシが大きいため、適切でない負荷分散を行なった場合のペナルティは非常に大きい。これを避けるため、負荷分散をプログラマが明示的に指定する方針をとる。下に示したものは、負荷分散を指定する pragma である。

- `@node(` X`)`: ゴールをノード X で実行する

ユーザが、クラスタ間の負荷分散を記述するのを補助するために、ライブラリのレベルで、スタック分割動的負荷分散方式 (STB) と、マルチレベル動的負荷分散方式 (MLB) が実装されている [76, 30]。

STB は、深さ優先探索アルゴリズムなどのスタックを用いた問題に対象を絞っている。探索木はスタックで表現されている。ユーザは問題に対して処理を行ない、新たな副問題と部分解をかえすプログラム expand と、部分解から全体の解を作り出すプログラム combine を作成する。システムは、あらかじめ対象を処理するプロセスをすべてのクラスタに分散させておく。それぞれが、スタックから取り出した問題に expand を適用し、副問題をスタックに戻し、部分解を combine でマージする。仕事がなくなると乱数で選択した他のクラスタに対して仕事を要求する。そのクラスタに仕事がないと、新たに乱数で他のクラスタを選択してそちらに要求を行なう。この方式は、対象となる問題が限られるという欠点を持つ。また、乱数で負荷の要求先を決めるため、要求先も負荷を持っていない可能性がある。この時は、再び乱数で負荷要求先を設定し直すが、場合によっては自分以外のすべてのクラスタに対して負荷を要求してしてしまう可能性がある。

MLB は、負荷バランサというサーバプロセスを特定の PE におき、このバランサを用いて、負荷を均等化する方式である。暇になった PE はバランサに申告をおこなう。忙しい PE はサブタスクができると、バランサに問い合わせを行ない、暇な PE を探して、この PE に対して pragma @node を用いて負荷分散を行なう。単純にこの方式を行なうと、サーバプロセスがボトルネックになるため、これを階層化したものが MLB である。MLB では、負荷の分散自体はシステムが行なうが、負荷を分散するポイントの指定をユーザが指定しなければならない。また、実際に負荷を分散するコード、バランサへの通信のコードはプログラムに埋め込まなければならない。これはユーザにとって大きな負担になり、プログラムの可読性を低下させる原因にもなっている。

これらの方式に共通する欠点は、要求駆動式であることである。負荷が不足してから負荷を要求するため、負荷の到着までの間はアイドルにならざるをえないため、実行効率が低下する。

PIM/c では、自動負荷バランシングを行なっている [43]。この手法は、乱数で選択したクラスタに対して負荷の問い合わせを行ない、そのクラスタの負荷が自分の負荷よりも小さければそのクラスタに対して、負荷を分配するという方式である。PIM/c はクラスタに対

する負荷の問い合わせを、CPU を経由せずに行なう専用のハードウェアを持っている。この手法は、乱数で選択することで問い合わせのオーバーヘッドを減少させているが、負荷バランスは不完全になり、またゴール間の局所性を無視した分散を行なうため、対象問題を選ぶ負荷分散方式となっている。

PIM/m での負荷分散 PIM/m[44] は、他の PIM と異なりクラスタ構造をとらない。いわば、1 クラスタあたり 1PE であるような構造になっている。このため負荷分散問題はより深刻である。PIM/m では、ネットワークがメッシュで局所性を持つ構造になっていることを利用して、動的負荷浸透方式 (LLS)[81] という方式を用いた負荷分散を行なっている。この方法は、プロセッサの近傍 (物理的に直接接続しているプロセッサ) に対して自分の負荷を報告し、近傍の負荷の平均よりも自分の負荷が高ければ、自分の負荷を近傍のプロセッサに分配するものである。この方式によって、ある一点においた負荷が時間にしたがってメッシュ上のプロセッサに次第に拡散していく。

この手法では、負荷分散のポイントをユーザが指定する必要はない。反面、余計な負荷分散が生じてしまう傾向をもつ。また、負荷分散があらゆる点で起きる可能性があるので、十分な局所性をもつ計算 (例えば OR 探索問題) 以外での有効性は疑問である。

以上に述べたように、PIM はそのハードウェア的な制約から、とくに動的な負荷分散問題に対する解決が不十分である。また、静的な問題の分割は、プログラマに任されている。これは本来ならばシステムがするべきことである。

2.3.2 KLIC

KLIC は、KL1 のポータブルなインプリメントをめざしてつくられた処理系で、C 言語へのコンパイラとなっている [28]。C 言語へのコンパイラとなっている理由は、

- ポータビリティ
- レジスタアロケーションなどの下位レベルの最適化をコンパイラに依存できる

である。

KLIC は処理系核 [99] と呼ぶ逐次処理系に対して generic object と呼ぶ拡張機構によって 並列、分散環境での実装を行なっている [88, 100, 92]。

generic object は新しいデータ型とそれに係わる操作を定義するものである。generic object には以下の三種類がある。

- data object : 新しいデータを作るためのオブジェクト。
- consumer object: プロセス形式のオブジェクト。変数にフックし、変数が具体化されると unify メソッドが呼ばれる。
- generator object: プロセス形式のオブジェクト。値の生成要求によって generate メソッドが起動される。

分散環境での実装で必要になる外部参照ポインタは、generic object の consumer と generator で実装されている。述語を実行するために、外部に参照を行なうと外部参照ポインタの generate メソッドを起動される。このとき述語は一度サスペンドする。外部参照ポインタの generate メソッドは、参照先の PE に対して read メッセージを送る。このメッセージを受信したプロセッサは、参照先が具体化されていれば、参照値を answer メッセージで返信する。参照先が変数である場合には、返信先を記録した consumer object を生成し変数にフックする。この変数が具体化されると、answer メッセージをもとの PE に返信する。もとの PE はこの answer メッセージをうけとると、先ほどサスペンドしていた述語をアクティベイトし、再実行する。このように、ポータブルな実装法をとることによってに容易にさまざまな並列機で実装することが可能になっている。

KLIC では、比較的高速な外部参照と外部参照時のレイテンシの隠蔽を一般性を失わずにたくみに実現している。しかし、負荷分散に関しては、とくにサポートを行なっていないため、VPIM と同様に MLB を用いての負荷分散をおこなっている。この手法は、有効なアプリケーションが限られる上、プログラマに、負荷分散の記述と MLB のチューニングという負荷をあたえる。これは十分とはいえない。

2.3.3 メッセージパッシングパスに沿った実行

Committed-Choice 型言語の高速実行には、コンテクストスイッチのコストを削減することが不可欠である。上田ら [68, 71, 69] は、FGHC を対象にして、ストリームとプロセスに着目してこのコストを削減するメッセージ指向実装を唱えている。

この手法は、ストリームには生産者プロセスと消費者プロセスが存在することに着目し、生産者 がストリームに値を書き出す際に、直接 消費者 を起動することで、消費者 プロセスのサスペンド / アクティベイト のコストを低減するものである。同時にメッセージを伝搬するためのストリーム用のリストセルも省略することができる。この手法を実現するためには、生産者プロセスと消費者プロセスを静的な解析によって検出する必要がある。このため、上田らは 対象を Moded FGHC と呼ぶ、すべての変数の入出力方向を定めることができる FGHC のサブセットに限定し、その上に洗練されたモード解析手法を構築することで静的な検出を可能にしている。

メッセージ指向実装は、当初は逐次処理系のために考案されたものであるが、並列機上にも実装されている。その際の実装の方針として、以下の二つが提案されている。

- Distribute goal method
  プロセスネットワークを分割して、それぞれをプロセッサに割り付ける。

- Shared goal method
  プロセスネットワークにながれるメッセージごとにプロセッサに割り付ける。

[71] では後者を採用している。例えば、素数を求めるプログラム primes の例を考えてみよう。 primes は、それぞれの素数に対応する篩がプロセスになり、その間を数列が値としてながれていく構造をもつ。通常の実装では、プロセスは単一プロセッサ上に留まり、値がプロセッサ間を移動するが、後者の実装ではそれぞれの値は常に同一のプロセッサ上にあり、プロセスである篩がプロセッサ間を移動するイメージになる。

確かに、 primes の場合はこの手法を用いることで、高速化されるが、この手法で高速化されるかはアプリケーションに依存する。メッセージ指向実装では、プロセス間のレイテンシは減少するが、プロセスのループ間のレイテンシは増大する。これらのどちらがボトルネックになっているかはアプリケーションによって異なり、プロセス間レイテンシがボトルネックになっているプログラムにはこの手法は有効であろうが、そうでないものに関しては、逆に実行時間が増大することも考えられる。

また、実際のプログラムを記述した感触では、ストリームを用いてプログラムを記述する部分は、支配的な量ではない。多くの場合、 Committed-Choice 型言語では、通常の変数と述語を用いたプログラミングスタイルで記述したものを、上部構造としてストリームで

接続するスタイルをとる。したがって、特に大規模なプログラムでは、ストリームの部分を最適化することによって得られる速度の向上には限界がある。

2.3.4 Committed-Choice 型言語の双対変換による最適化

Committed-Choice 型言語のプログラムには、プロセスとメッセージの間に双対性がある [40, 75, 97, 78]。この双対性を利用し、データで表現されているものをゴールで、ゴールで表現されているものをデータで表現することが可能である。この変換を双対変換と呼ぶ。この変換によってプログラムの計算量は変化しないが、サスペンドの回数が減る場合がある。この変換には双対性があるので、もちろんサスペンドが増える場合もある。

久門らは、この変換を用いてプログラムの最適化を行なうことを提唱している。この変換は、もとのプログラムではストリームによってデータ依存として表現されていた依存関係を通常の制御依存に、制御依存をデータ依存へと変換する。結果として、前項で述べたメッセージ指向的な実行形態が実現される。

この研究は、非常に興味深い理論的側面を持つが、プログラムの最適化に用いることに関しては実用性に疑問が残る。この変換では、プログラムの実行が高速化される場合があるだけで、どのような場合に高速化が可能なのかは必ずしも明らかでない。さらに、一つのプログラムに対して複数の場所にこの双対性が見い出せるため、どこをどのように双対変換すれば高速化するのかを見い出すのは非常に難しい。また、前項の実装と同様に、この変換による最適化はストリームを用いる部分にのみ有効であるため、最適化の対象領域の大きさにも疑問が残る。

第 3 章

PIE64 と PIE64 上の fleng 実行時処理系

本章では、本研究のターゲットとする計算機、並列推論エンジン PIE64 について述べる。さらに、PIE64 上に実装されている fleng の 実行時処理系 piexf に関して述べる。

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 [59, 3]。PIE64 は 64 個の要素プロセッサを持つ。各要素プロセッサは 2 つの 3 段のクロスバネットワークで相互に結合している。PIE64 は、通常のワークステーションをホストマシンとして持つ。ホストマシンは、I/O やシンボルとシンボルの印字文字列の対応などの一元的に管理する必要がある資源の管理を受け持つ。また、64 台全体での同期をとるための専用線を持っている。PIE64 の構成を図 3.1に示す。

要素プロセッサは、3 種類のプロセッサをもっている。fleng の実行を行なう専用のプロセッサ UNIRED (Unifier / Reducer)、通信 / 同期を専門に行なう NIP(Network Interface Processor)[62]、並列計算にともなう雑多な仕事を請け負う MP (Management Processor) である。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。この要素プロセッサを Inference Unit (IU) と呼ぶ。Inference Unit の構成を図 3.2に示す。メモリは各要素プロセッサに分散して実装されているが、アドレス空間はグローバルになっており、分散共有メモリとなっている。

UNIRED[60] は、fleng の実行を念頭においたタグアーキテクチャと、命令セットを持つプロセッサである。UNIRED は他の IU のアドレスに対するアクセスを行なう場合に

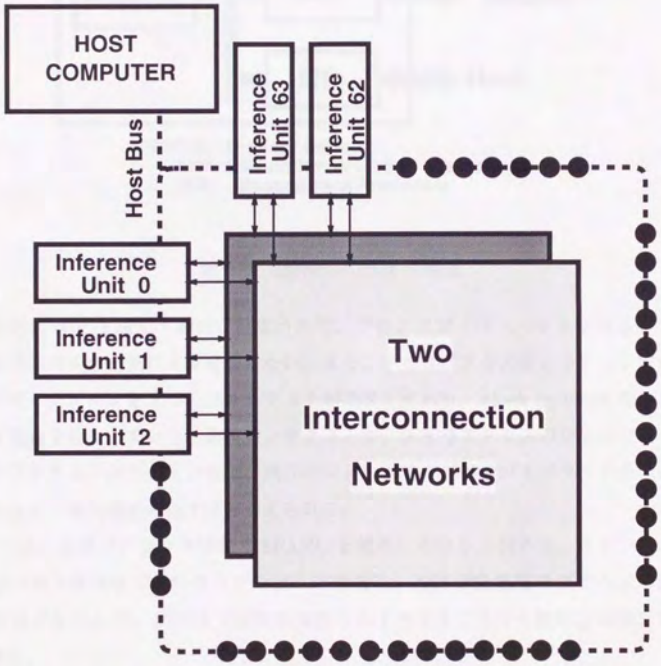


図 3.1: PIE64 の構成

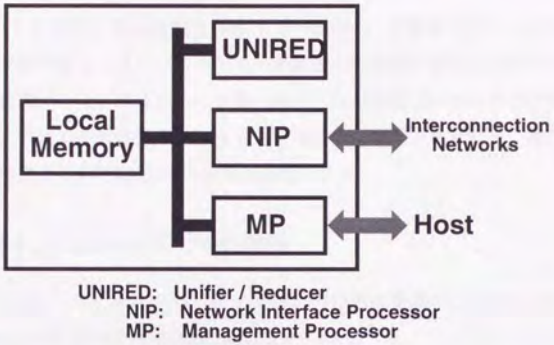


図 3.2: Inference Unit の構成

は、自動的に NIP を通じて通信を行なうので、プログラマ / コンパイラはあるメモリがリモートかローカルかを気にする必要はない。また、メモリアクセスのレイテンシ隠蔽のために一つのパイプラインを 4 つのコンテクストが共有しており、clock by clock のマルチコンテクスト実行を行なっている。あるコンテクストが、メモリアクセスのためにブロックしても他のコンテクストが生きていれば、残りのコンテクストだけでパイプラインを埋めて実行を続けるため、実行速度の低下はおさえられる。

MP には、汎用プロセッサである SPARC を使用している。MP は、スケジューリングの管理やメモリ管理などを行なうプロセッサである。MP が計算用のプロセッサ以外に独立して存在することで、PIE64 では計算速度を低下させることなく動的な制御を行なうことができる。

NIP は UNIRED と協力して、分散共有メモリを実現する。NIP は記号処理に適した、抽象度の高い通信インターフェイスをもっており、計算用プロセッサに負荷をかけずに高速度な通信を実現している。また単一代入変数による同期を直接サポートしている。すなわち、ある単一代入変数に対して値の書き込みを待つスレッドのリストの管理を行ない、変数に対する書き込みがあると、そのスレッドのリストを MP に渡す機能を持っている。このように高度なサポートをがあるため、UNIRED 上のコードはリダクションそのもののみに

専念することができる。

ネットワークは自動負荷分散機能を持っている [63]。各要素プロセッサがネットワークに自分の負荷情報を流し、各クロスバスイッチが最小の負荷を選択してフォワードすることによって、各要素プロセッサはもっとも軽い負荷をもつ要素プロセッサとその負荷値を知ることができる。自動負荷分散機能によって負荷は均等に分配されるので、最小負荷の要素プロセッサの負荷値から PIE64 全体の負荷が類推できる。

3.2 PIE64 上の fleng 実行時処理系

PIE64 の fleng システムは、コンパイラ系と実行時処理系の二つからなる。本節では、PIE64 上の fleng 実行時処理系である piexf について述べる。

3.2.1 実行時処理系 piexf

PIE64 上の fleng 処理系 piexf は、大きく分けて次の 3 種類のプログラムからなる。

- ホストマシン上で動くホストプログラム
- MP 上で動く ランタイム・カーネル
- UNIRED 上で動く Dispatcher ルーチン

図 3.3に piexf の構成を示す。

ホストプログラム

piexf はホストマシン上の unix の一つのプロセスとして起動する。このプロセスは、まず、PIE64 を初期化し MP にランタイム・カーネルを、UNIRED に Dispatcher ルーチンをロードする。このプログラムによって、コンソールに対する入出力、ファイルに対する入出力、X-Window を用いた入出力インターフェイスが与えられる。また、シンボルとシンボルの印字文字列の対応などの一元的に管理する必要がある資源の管理も行なう。

ランタイム・カーネル

ランタイム・カーネルの役割は、大別すると以下の三つにわかれる。

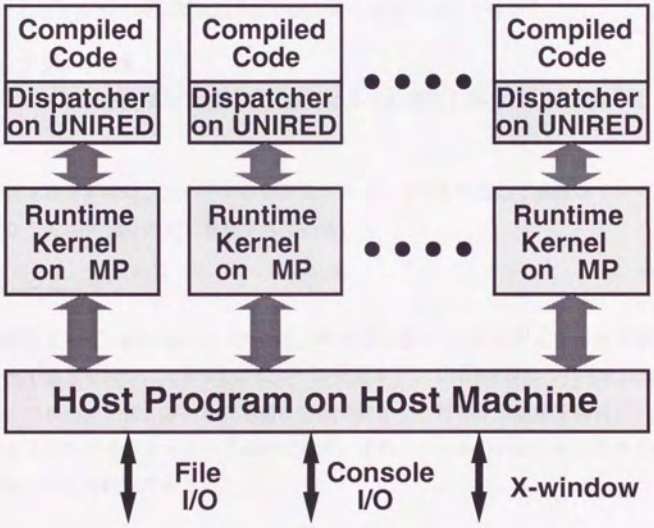


図 3.3: piexf の構成

- **ゴールの管理**
  fleng の実行単位であるゴールをスタック状のゴールプールに蓄え、UNIRED に送って実行させる。また、MP 同士で通信を行ないゴールがすべての IU に均等に行き渡るようにする。

- **メモリの管理**
  UNIRED や NIP に対して、メモリをページ単位で供給する。メモリが不足すれば同期をとってすべての UNIRED を停止させ大域的な GC を行なう。

- **プログラムの管理**
  プログラムを述語単位で管理し、必要になった述語を動的に UNIRED にロードする。

ランタイム・カーネルは、スパークのレジスタウインドウを利用して高速なマルチコンテクストを行なうモニタ [31] の上に構築されている。

Dispatcher ルーチン

UNIRED 上の Dispatcher ルーチンは、その名の通りゴールに対してそれを実行する述語を呼び出す働きを持つ。UNIRED にゴールが渡ると、UNIRED はこの Dispatcher を起動する。このルーチンは、ゴールの名前と引数の数から、対応する述語を特定し、その述語の定義であるコンパイルド・コードを呼び出す。また、ゴールの引数をレジスタに展開するのもこのルーチンの仕事である。

3.2.2 IU 内の協調実行

各 IU での fleng の実行は UNIRED、NIP、MP の三つのプロセッサの協調によって行なわれる [95]。NIP は分散共有メモリを実現と、サスペンド / アクティベイトの処理を行なう。MP はゴールプール、メモリなどの管理を行なうランタイム・カーネルを実行する。NIP/MP がこれらの、いわば余分な仕事を行なうので、UNIRED で実行されるプログラムはゴールの書き換えに専念できる。図 3.4に IU 内での 3 プロセッサの協調の様子を示す。

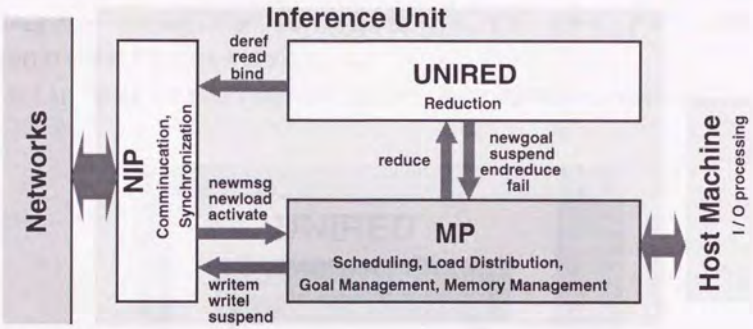


図 3.4: IU 内協調実行モデル

MP 上のランタイム・カーネルは、UNIRED のコンテクストがすべて埋まるかゴールがなくなるまで、ゴールプール中のゴールを選択して UNIRED に送る。UNIRED は、ゴールをリダクションし新たなゴールを生成して、これを MP のランタイム・カーネルに送り返す。 ランタイム・カーネルは、他の IU の負荷と自分の負荷を比較して、他の IU の負荷が自分の負荷よりも少なく、他の IU に送った方がいいと判断した場合には、UNIRED から受けとったゴールを NIP を介して他の IU に送る。そうでない場合には、ゴールをゴールプールに蓄える。

UNIRED でリダクションを行なおうとしたさいに、変数が十分束縛されていず、リダクションができなかった場合には、サスペンドが生じる。UNIRED はそのゴールを NIP に送る。NIP は、サスペンドの原因となった変数に対して、サスペンドリストと呼ぶリストを作り、そこにゴールを登録する。

UNIRED で変数に束縛を行なう場合にも、NIP に束縛を依頼する。ある変数が束縛されると、その変数のサスペンドリストに登録されているゴールがアクティベイトされる。NIP はこれらのゴールを MP に渡す。MP は NIP から受けとったゴールを、ゴールプールに蓄える。

また piexf には、インタプリタが MP 上にインプリメントされており、コンパイルされ

ていない fleng のプログラムの実行も可能である。コンパイルされたオブジェクトコードと fleng のコードが混在して動作することが可能である。また、将来インタプリタの実行も UNIRED に移行することも計画されている。

図 3.5に PIE64 での fleng の実行の様子を示す。 図中の矢印はゴールの流れを示す。

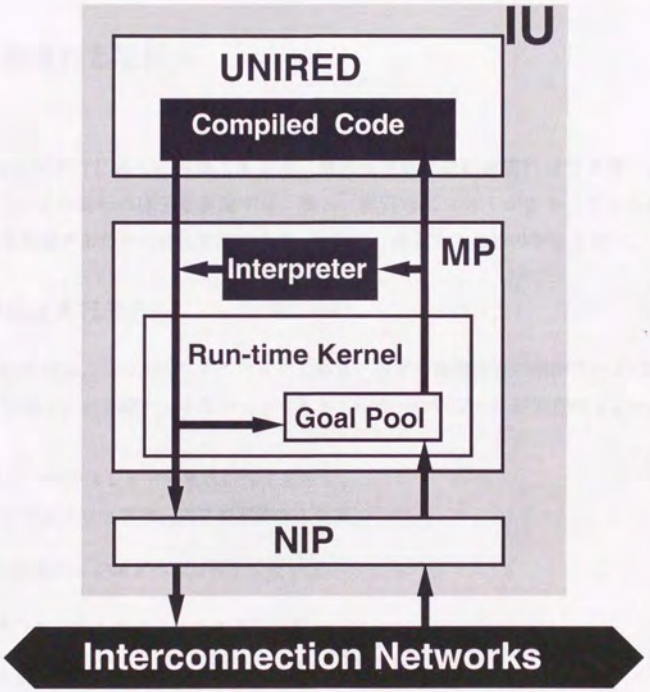


図 3.5: PIE64 での fleng の実行

第 4 章

fleng の実行と問題点

fleng は高並列実行に適した言語であるが、素朴な実装では高速実行はできない。本章では、まず fleng の実行の様子を詳説する。次に、実行時にコストがかかっている点を解析し、これを削減するための要件を検討する。最後に、削減するための手法を示す。

4.1 fleng 実行モデル

fleng の実行は、ゴールのリダクションである。通常の言語実装の用語でいえば、一回のゴールのリダクションが一つのスレッドであり、一つ一つのゴールが実行のフレームに相当する。

図 4.1に一つのスレッドの実行の様子を示す。

fleng の実行スレッドは、以下の要素からなる。

- 実行環境のレジスタへのロード / セーブ
- サスペンドチェック / パターンマッチ
- システム述語の実行 / 変数の束縛
- サブゴールのゴールフレームの作成

ゴールフレームは、通常はヒープに確保されている。ゴールをリダクションするにはまず、このヒープ上のゴールの情報をレジスタに取り出す必要がある。サブゴールのゴールフレームを作成する際には、ちょうど逆にレジスタ上に存在するサブゴールの環境をヒープ

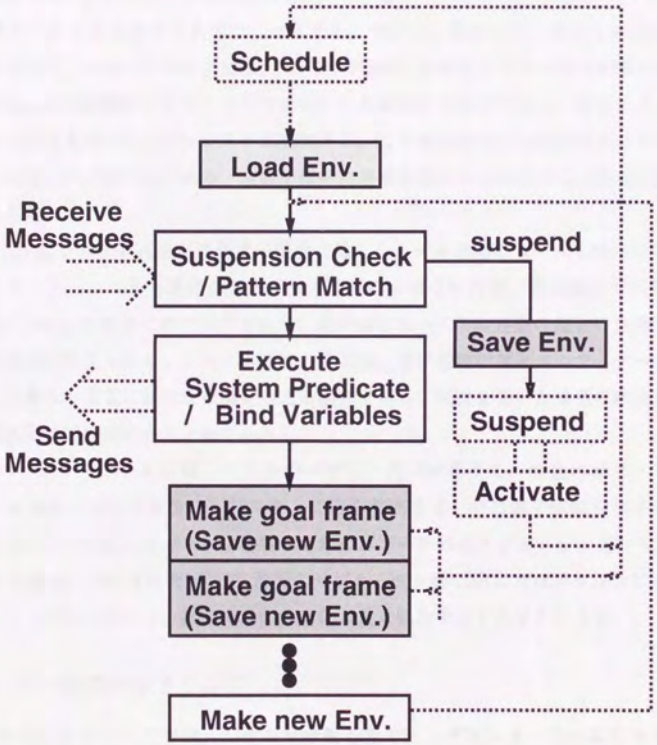

Schedule
Load Env.
Receive
Messages
Suspension Check
/ Pattern Match
suspend
Save Env.
Execute
System Predicate
/ Bind Variables
Suspend
Send
Messages
Activate
Make goal frame
(Save new Env.)
Make goal frame
(Save new Env.)
Make new Env.

図 4.1: スレッドの実行

上のゴールフレームに書き込む。通常の逐次言語では、サブルーチンの際に呼び出し元のコンティニュエーションをスタックにセーブして、実行をサブルーチンに移す。これに対して fleng では逆に呼びだし先のコンティニュエーションをセーブしていると考えられる。

fleng のサスペンドチェックの部分では、ゴールとクローズのヘッド部のパターンを照合して、コミットできるかどうかをチェックする。できない場合には、実行を中止してサスペンドを行なう。パターンマッチ部は、ゴールの実行に必要なデータをすべて揃える役割も担っている。ある変数が、リストセルであることを確認するだけでなく、そのリストセルの car 部 や cdr 部を取り出してレジスタに展開する。この様に変数への参照が生じるのは、このパターンマッチの時のみである。変数を用いた通信の観点からいえばこの部分が受信部であるといえる。

ボディーにシステム述語が存在する場合には、システム述語のゴールを作ることなく直接実行を行なう。システム述語はサスペンドすることはないため、直接実行することが可能である。fleng のすべてのプログラムは、最終的にはシステム述語の集合に展開されて実行されていると考えられる。システム述語の役割は、主に数値計算とユニフィケーションであるが、どちらも変数に新たな束縛を与える動作である。変数を用いた通信の観点からいえば、この部分は送信部であると考えられる。

最後のボディーゴールに関しては他のボディと扱いが異なる。最後のボディーゴールは、ゴールフレームにではなくレジスタ上に値を整列させ、その場で制御を移す。これはゴールフレームを作成するコストを省略するためとゴールのスケジューリングコストを削減するためである。これは逐次言語の 最後のサブルーチン呼び出しではスタックにコンティニュエーションをつまない、終端再帰の最適化と良く似た手法であるといえる。

4.1.1 ゴールのハンドリング

ゴールのハンドリングとは、ゴールを分配したり、リダクションの対象にするために UNIRED に渡したりすることである。このようなハンドリング操作は逐次言語には全く存在しないことであり、最大の相違点である。特に fleng には、サスペンドとアクティベイトという機構が存在するため、ハンドリングは他の並列言語と比較しても複雑になっている。

ゴールのハンドリングの主な操作は、ゴールプールからゴールを取り出し UNIRED にわたすことと、UNIRED が作成した新しいゴールをゴールプールに戻すことである。

PIE64 上の piexf では各 IU 上に、ゴールプールを実装している。piexf での負荷分散は基本的にゴールの生成時に行なう。すなわち、ある IU 上で新たにゴールが作成された時点で、その IU 上の負荷、各 IU 上の負荷とそのゴールに対するスケジューリングポリシから導かれる IU に対してそのゴールを送信する。この点が、要求駆動型の PIM 上の KL1 の実装と異なる。

PIE64 上の fleng では、ゴールのハンドリングは図 4.2のように行なわれる。

UNIRED で作成されたゴールは、MP に渡される。実行時カーネルは、UNIRED に空いたコンテクストがある場合には、即座にそれを UNIRED に送り返す。この場合処理は FIFO 的に行なわれる。他の IU の負荷が低く、ゴールを渡したほうがよいと判断される場合には、NIP を通してその IU に渡す。

空きコンテクストがなく、他の IU の負荷も低くない場合には、ゴールはゴールプールに入れられる。このプールはスタックになっているのでここでは制御は LIFO 的に行なわれる。UNIRED に空きコンテクストが新たに生じると、実行時カーネルはスタックのトップのゴールを UNIRED に渡す。

UNIRED では、ゴールのヘッド部のパターンマッチを行なう。ここでサスペンドしたゴールは、NIP に渡される。NIP はこのゴールを対象の変数に対してサスペンドさせる。他のゴールのリダクションによって、対象の変数に値が代入されると、サスペンドが解ける。変数への値の代入は NIP が行なうので、NIP はサスペンドが解けるのを知ることができる。NIP はサスペンドの解けたゴールを MP に渡して、再び実行の対象となるようにする。これがアクティベイトである。MP は、このゴールを UNIRED から来たゴールと同様に扱う。

4.2 fleng の実行のコストと最適化手法

fleng の実行のコストは、ゴールリダクションのコストとハンドリングのコストに大別できる。

4.2.1 ゴールリダクションのコスト

ゴールリダクションのコストは以下のものからなる。

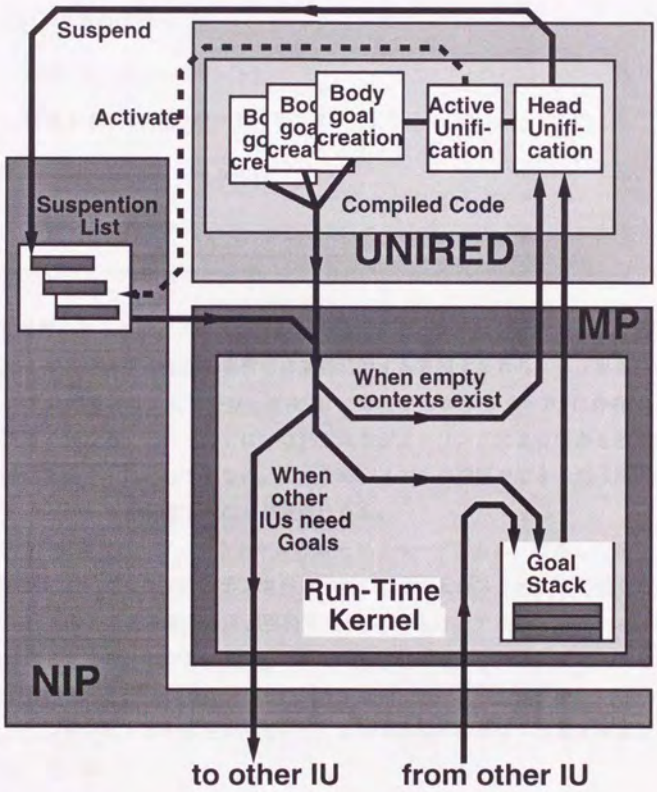


図 4.2: ゴールのハンドリング

- サスペンドチェック
  - メモリ参照 (ローカル / リモート)
- システム述語の実行
  - 束縛 (ローカル / リモート)
  - 構造メモリの確保 (ローカル / リモート)
- サブゴールのゴールフレーム作成
  - フレーム領域の確保
  - フレームへの環境のセーブ

ここで問題になるのは、メモリ参照の局所性である。サスペンドチェックの実行ではメモリ読み出しが、システム述語の実行では変数に対する束縛すなわちメモリ書き込みが行なわれる。いずれの場合も、変数がローカル IU に存在するかリモート IU に存在するかで、コストは大きく異なる。このコストは、コードスケジューリングによってある程度隠蔽できるが、完全に隠蔽することはできない。このため、メモリの参照ができるだけ局所的になるようにゴールとデータを配置することが重要になる。

もう一つの問題点は、ゴールフレームのロード / セーブである。前述した通り、この操作は逐次言語においてもサブルーチンコールで生じる操作とコスト的には等価であり、同じ計算木を実行すると考える場合、逐次言語でも fleng でもコンティニュエーションをセーブする回数は同じである。しかし、fleng では大域変数が存在しないことによってセーブする変数の量が多いことと、一つのスレッドの長さが短いため計算木自体が細かくなってしまうことから、コンティニュエーションのセーブ、ロードにかかるコストは逐次言語よりはるかに大きくなっている。

このコストを低減するには一度の実行で走る長さを長くすることが必要になる。

4.2.2 ハンドリングのコスト

ハンドリングのコストは、以下のものからなる。

- サスペンド / アクティベイト のコスト
  - 実行環境のロード / セーブ
  - 複数回のサスペンドチェック
- MP でゴールを扱うコスト

PIE64 では、NIP がサスペンド / アクティベイトをサポートしており、そのコストのほとんどは NIP が引き受けている。しかし、それでもサスペンドは実行効率に多大な影響を与える。サスペンドが生じると、ゴールの実行環境のセーブを行なわなければならない。また、アクティベイトする時にも実行環境をロードし、サスペンドチェックをするという重い処理を再び行なわなければならない。これらの処理は、計算には全く貢献しない無駄な動作である。また、サスペンドが生じると、UNIRED のコンテクストが解放される。UNIRED は空きコンテクストができたことを MP に通知する。MP はすぐに新しいゴールを送り返すが、新しいゴールを受けとるまでには、しばらく時間がかかりその間空きコンテクストが存在してしまう。このため UNIRED のコンテクストの使用率が低下し、実行効率が低下する。

もうひとつのコストは MP のランタイム・カーネルにかかるコストである。ランタイム・カーネルのもっとも重要な役割は、UNIRED のコンテクストを常に埋めておくことである。すなわち、UNIRED からの リダクション終了 メッセージに対して、なるべく早く新しいゴールを送ることである。スケジュールの対象になるゴールがたくさん発生すると、これをキューに収めたり、他の IU に送ったりする処理で、MP での処理が滞り、新しいゴールを送るまでに長い時間がかかるようになる。すると UNIRED のコンテクストの使用率が低下し、実行効率が低下する。一つのゴールの実行長が長くなれば、MP に渡されるゴールの数が減少してこのコストを低減できる。

4.3 fleng の高効率な実行に必要となる制御

前節でみたように fleng のプログラムの実行を高速化するためには、

- 局所参照率の向上

- サスペンド率の低減
- スレッド長の増大

の三つが重要である。このため具体的なの手法として、静的負荷分割による負荷分散の抑制、静的スケジューリング、粒度制御が考えられる。

4.3.1 負荷分散の抑制

PIE64 の fleng 処理系では、デフォルトですべてのゴールを均等にプロセッサに割り当てる。これは、PIE64 では負荷平滑がハードウェアでサポートされていること、PIE64 のリモートメモリ参照が比較的高速であることと、UNIRED がマルチコンテクストを持ちある程度のメモリ参照のレイテンシがある程度隠蔽できることから、プロセッサの稼働率を第一に考えて選択した戦略である。

しかし、UNIRED のマルチコンテクストによるレイテンシ隠蔽も完全ではない。コンテクスト数が少ない時にはもちろん無効であるし、レイテンシが隠蔽できたとしてもクリティカルパスが伸びることによって処理時間がのびることは防ぐことができない。

これを防ぐためには、負荷分散を適切に抑制する必要がある。局所性をもつゴール、変数に関しては負荷分散を抑制し、局所的に配置することによってメモリ参照 / 書き込みの局所性を上げることができる。

4.3.2 静的スケジューリング

前述の通りサスペンド / アクティベイトのコストは非常に大きく、何らかの方法でこれを削減する必要がある。このためには、ゴールがリダクションの対象になる順序を制御する必要がある。具体的には、あるデータを必要とするゴールはそのデータを作成するゴールよりも後にリダクションされるようにしなければならない。このような、理想的な実行順序はある程度静的に求めることができる。しかし問題は、どのように実際に制御するかである。複数の IU で実行されるゴールの順番を制御することは、オーバーヘッドが非常に大きいため現実的ではない。従って、静的にスケジューリングするためにはスケジュールの対象となるゴールを一つの IU 上におくことが不可欠であり、ここでも負荷分散の抑制が重要になる。

4.3.3 粒度制御

fleng のゴールフレームの作成とゴールフレームからの展開のコストは大きい。また、ゴールフレーム自体の確保もヒープを圧迫し、数多くの GC を引き起こし、実行効率の低下を引き起こす。これらのコストを低減するためには、実行の粒度を大きくする、すなわち一つのゴールフレームから生じるスレッドの長さを長くすれば良い。これについては、節を改めて議論する。

4.3.4 コスト削減とプロセッサ稼働率

上記のような最適化を行なうことで実行コストを削減することが可能であると思われるが、ここで注意しなければならないのは、プロセッサの稼働率である。コストを削減しても並列度が下がり、稼働率が低下しては高速実行は期待できない。上述の最適化はいずれも負荷バランスとトレードオフの関係にある。静的スケジューリングを確実に行なうには、スケジュールされるゴールは同一の IU 上になければならず、負荷分散を抑制しなければ実現できない。負荷分散を抑制することは負荷の不均衡を生む。また粒度を粗くすれば、各 IU の負荷がばらつきやすくなる。

したがってこれらの最適化を行なうにはある程度負荷のバランスを犠牲にしなければならない。これは、プログラムのその時点での並列度が充分にない場合には、プロセッサの稼働率を減らすことに直結する。したがってこれらのコストを削減を考える際にはプログラムのその時点での並列度をつねに考慮に入れなければならない。

しかし並列度は動的に定まるため静的に予想することは難しく、したがって負荷バランスとトレードオフの関係にあるような最適化は通常の方法では難しい。

そこで、並列度に応じてこれらの制御を行なうことが考えられる。つまり並列度が高い時には、負荷バランスをくずすようなオプティマイズを行ない、低い時には行なわなければ良い。本研究では、複数の異なる最適化を行なったコードを用意し、これらを動的に切替えることによって並列度に応じた最適化を行なう。

4.4 fleng の実行粒度制御

実行粒度の制御には、二つの手法が考えられる。一つは、動的な手法である。これは、ゴールの実行をゴールキューを介さずに行なう。UNIRED 上にスタックフレームを作成して、サブゴールをゴールフレームを作成せずに実行するようにすれば、高速な実行が期待できる。これはひとつのゴールハンドリングのコストに対して、複数のリダクションをおこなうことで、ゴールハンドリングコストを少なくする手法である。

もう一つは、静的な手法である。プログラムを静的に構成し直して、1つのリダクションで行なわれる仕事量を大きくする。1リダクション辺りの実行スレッドの長さが、倍になれば計算全体から見れば、ゴールハンドリングのコストは半減したことになる。このための手法として、プログラム変換を行なっていくつかの述語をまとめることが考えられる。本研究ではプログラム変換 [77] の一つである述語の展開を用いて行なう。展開とはサブゴールの定義をその場に展開することである。展開により、一つのゴールの実行長は一般に長くなり、粒度を向上させたことになる。

これらの手法は一長一短である。前者には以下のような特徴がある。

○ すべてのクローズに適応可能

× 低並列時には負荷の集中を引き起こすため使用できない。

× スタックフレームを作成するコストはかかる

後者には以下のような特徴がある。

× プログラム変換できる場所は限られる

○ 高並列時、低並列時ともに可能

○ ゴールハンドルのコストが完全になくなる。

これらの手法は完全に直行しており、併用が可能であり、本プロジェクトではこれらの双方を用いる。

4.4.1 PIE64 における動的粒度制御

ここで、PIE64 における動的粒度制御について簡単に説明する。

PIE64 では、動的な粒度制御手法としてスタックを用いた実行を行なう。この手法ではサブゴールを call で実行する。この際に、もとのゴールの コンティニュエーション を スタック上にセーブする。これは逐次言語のサブルーチンコールとほぼ同じである。ゴールの実行時にサスペンドが生じるとオンデマンドで、ゴールフレームをアロケートし、サスペンドを行なう。

負荷状態が変化して、この手法を使うのがふさわしくない状態になった場合には、call をトラップして強制的にヒープ上にゴールフレームを作成させ、スタックを用いる実行を中止する。

この動的粒度制御に関する詳細は [93] にゆずる。

4.5 Multiple Code

本研究では、4.3.3 で述べたように複数の異なる最適化を行なったコードを用意し、並列度に応じた最適化を行なう場合、いくつコードを用意するのが妥当であろうか。コードの数を増やすとそれだけきめ細かい制御が可能になるが、コードの数をふやすとコード量が増えてメモリを圧迫する上、制御へのオーバーヘッドが増大するといった問題も生じる。

4.5.1 fleng の実行パターン

コードの数を決定するために、ここで、fleng プログラムの実行時の並列度について調べてみる。

2.2.2で述べた通り、fleng のプログラムは粗粒度並列 をストリーム + プロセスで、細粒度並列 単一代入変数 + ゴールで表現する。

ここで問題になるのは、粗粒度並列 の部分である。多くのプログラムでは、まずストリームのセットアップを行なう。この時ストリームのセットアップ自体には並列性が期待できず、多くの場合この部分の並列度は低い。しかし、一旦セットアップが行なわれてしまえば、粗粒度 / 細粒度双方の並列性が期待できるため充分な並列度が得られる。図 4.3に bestpath を求めるプログラムの実行例を示す。当初低迷する並列度が、ある時を境にして

急激に上昇することがわかる。また終盤においても、急激に並列度が低下し、そのあとはしばらく低いままで実行したあと停止する。

このように fleng のプログラムの実行は 2 つのフェイズからなるとみなすことができる。

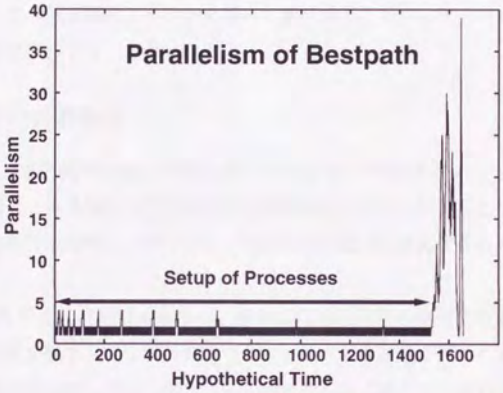


図 4.3: fleng プログラム実行のフェイズ

fleng プログラムの実行は、極端な低負荷と高負荷のどちらかで行なわれているケースが支配的である。この性質を利用して低負荷時向けのコードと高負荷時向けのコードを用意して切替えれば、多くの時間において適切な制御が行なえると考えられる。

4.5.2 高負荷時の最適化

高負荷時には、並列度が充分に存在することを仮定した最適化を行なう。並列度が充分にあるため、ゴールを他の IU に渡した場合にそれが実行されるまでには、ある程度の実行時間がかかることが予想される。また、ある程度負荷の不均衡が生じてもそれが直接プロセッサ稼働率を低下させることにならない。このため、負荷の不均衡を引き起こす可能性のある最適化が行なえる。

高負荷時の最適化はスタックによる粒度の調整と述語の展開による粒度の調整を中心に行なう。負荷分散の観点から見ると、粒度を調整することは、負荷分割を停止したことに相当する。本来他の IU に分散することが可能であったタスクを、親ゴールを実行した IU が抱え込むことになるからである。このようにローカルにゴールを抱えることによってより大域的な静的なスケジューリングが可能になる。展開したクローズに対して、静的にデータ依存解析を行ない、その依存順にスタックを用いて実行することで、サスペンドを起こさないスケジューリングで実行することができる。

4.5.3 低負荷時の最適化

低負荷時には、並列度が不足しており空いている IU が存在することを仮定した最適化を行なう。この場合、ある暇な IU に投げた仕事は即座に実行されることを仮定することができる。また、負荷の不均衡は、即プロセッサ稼働率の低下に直結するため、極力避けなければならない。

このため動的な粒度の調整はできない。静的な粒度の調整も並列度の低下が起きない場合、もしくは並列度が低下しても実行のクリティカルパスの長さが短くなる場合のみに限定して行なう。負荷分割に関しては、各クローズ内のゴールに関して、静的にグルーピングを行ないその単位で IU への割り当てを行なう。同時に静的にスケジューリングを行なう。

4.5.4 動的コード切替え

コードの切替えは、ランタイム・カーネルが行なう。ランタイム・カーネルは適当なインターバルで負荷を監視しており、負荷状態が変化するとローカルメモリ上のフラグを書き換える。するとそれ以降に、UNIRED に渡されたゴールに対しては、Dispatcher がフラグを見て実行するコンパイルドコードを切替える。

この様子を図 4.4に示す。

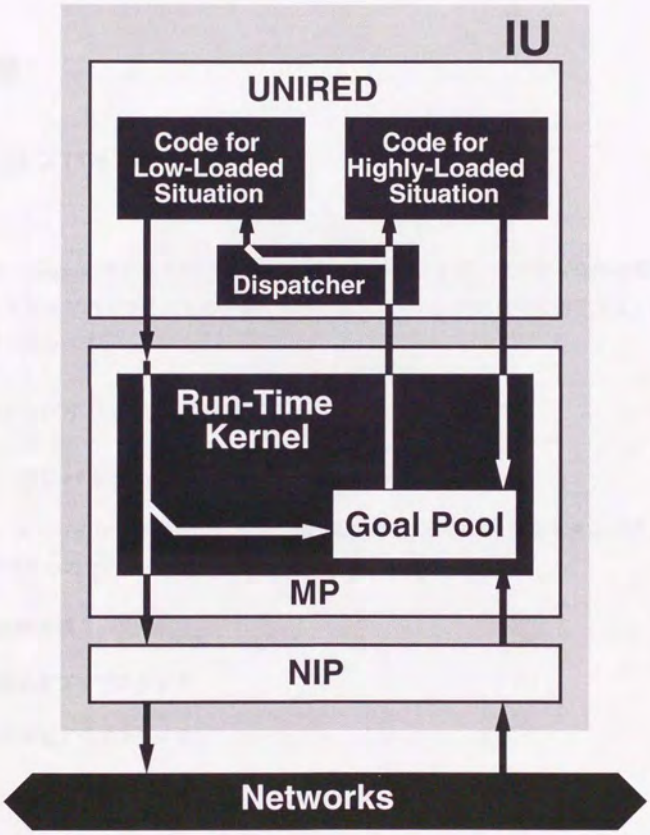


図 4.4: Dual Code

第 5 章

fleng コンパイラシステム

本章では、fleng のコンパイラシステムに関して述べる。まず、システム全体の構成について述べ、次にコンパイラそのものに関して述べる。さらに、静的負荷分割手法と、静的粒度調整についてのべる。

5.1 コンパイラシステムの構成

5.1.1 コンパイラシステムの概要

fleng コンパイラシステムの構成を、図 5.1に示す。コンパイラシステムは大きく分けて、以下の 6 つのプログラムからなる。

- 静的解析機
- 粒度調整プリプロセッサ
- 負荷分割プリプロセッサ
- コンパイラ
- マージャ
- アセンブラ

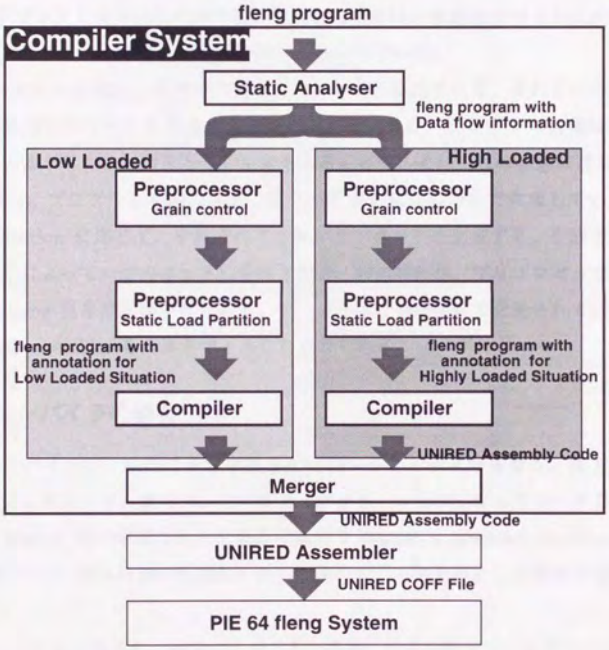


図 5.1: fleng コンパイラシステム

静的解析機は、モード解析、データ依存解析、プログラムの意味解析を行なう。プリプロセッサは、静的解析機の出力するデータと、与えられた最適化ポリシに従って、それぞれ、プログラム変換による粒度調整と負荷分割 annotation の付加を行なう。コンパイラは、annotation を解釈し、指定された最適化を実現したアセンブラ・コードを出力する。このコードをアセンブラによってアセンブルすることでオブジェクトコードを得る。

プリプロセッサとコンパイラを分離して実装したのは、最適化のポリシと、最適化の実装を分離して実装することで、実験を容易にするためである。

このシステムの特徴は、途中でパスが二つにわかれる点である。それぞれのパスは、低負荷時、高負荷時のコードを生成する。プリプロセッサは、プログラム自体は双方のパスで共有しているが与える最適化ポリシを変更することで、それぞれの状態に適した annotation つき fleng プログラムを出力する。コンパイラも双方のパスで共有しているが、それぞれの annotation に応じて、それぞれアセンブラ・コードを生成する。生成されたコードは、マージャによって一つのコードに合成される。静的解析器、プリプロセッサ、コンパイラはすべて fleng 自身で記述されている。マージャは、perl[90] で記述されている。アセンブラは、gnu assembler をカスタマイズしたものである。

5.1.2 コンパイラの概要

fleng コンパイラは、中間コードをはさんで二つのフェイズからなる。第 1 フェイズは中間コードジェネレータ、第 2 フェイズはターゲット・コード・ジェネレータである。中間コードは、Prolog 用の中間コードである WAM (Warren's Abstract Machine)[73] や、KL1 の中間コード KL1-B [38] と類似した、無限のレジスタを仮定した仮想マシンコードとなっている。

中間コードのレジスタの、実際のレジスタへの割り付けは第 2 フェイズで行なわれる。

このような構成をとることで、ターゲット・コード・ジェネレータを書き換えるだけで、さまざまなマシンに向けたコードが作成できる。実際、本コンパイラはもともとワークステーション上の fleng 処理系向けに、C 言語を出力するために作成されたものである。現在ターゲット・コード・ジェネレータは、PIE64 向けの UNIRED コードの他、C 言語を出力するものと 88000 アセンブラ・コードを出力するものが存在する。C 言語を出力するものは、通常の逐次 UNIX ワークステーションおよび並列計算機 AP1000[61] 上の fleng 処

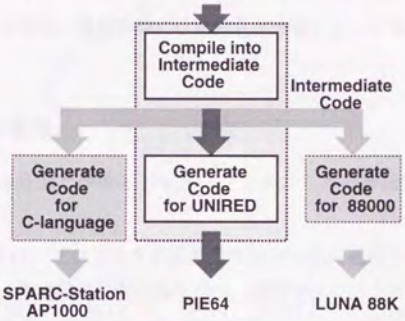


図 5.2: fleng コンパイラ

理系 [87] をターゲットにしたものである。88000 アセンブラ・コードを出力するものは、ワークステーション LUNA-88K をターゲットとしている。

5.2 負荷分散と静的スケジューリング

本節では、本システムの負荷分散、静的スケジュールに関して述べる。

本システムでは負荷分散を以下の三段階によって行なう。

1. **コンパイラによる静的負荷分割**
   データ依存関係によって実際には並列実行できないグループにゴールを分割する。

2. **ランタイム・カーネルによる動的負荷分割**
   実行時の負荷状態に応じて、1. で分割したグループを他の IU に投げるかどうかを決定する。

3. **相互結合網による自動負荷平滑**
   投げると決定したゴールグループを、最小負荷の IU に投げる。

これらの階層を 4.5で述べた複数のコードを用いて実現する。低負荷時用に静的負荷分割を行なったコードを、高負荷時用に負荷分散を全く行なわないコードを用意する。低負荷

時には、静的負荷分割で分割されたゴールのグループを相互結合網によって平滑化を行ないながら負荷分散が行なわれる。高負荷時には動的負荷分割によって負荷分散が完全に抑制される。

5.2.1 静的解析の概要

本項では、静的負荷分割と静的スケジューリングのための静的解析手法について述べる。

静的負荷分割の目的は、プログラムの並列度を減少しない範囲でメモリ参照の局所性を向上させることである。静的負荷分割に関しては、プロファイリングを行ないその結果を用いる方法が日高により提案されている [94, 32]。この手法は、プロファイリングを行なった結果を ATMS (Assumption based Truth Maintenance System) に似た機構で多重世界的に管理することで、最適な負荷分割を行なうものである。しかしこの手法は、プロファイルの実行に時間がかかる上、プロファイラの出力の解析にも膨大な時間がかかるため、負荷分割の手法としては適していない。

本論文では、データ依存関係を静的に解析することで、最適な分割を行なう手法をとる。この手法では、データ依存関係があり並列に実行できないゴールをまとめてひとつのプロセッサに割り当てる。このようにすると、まとめられたゴール間にはもともと並列度はないので、プログラム全体の並列度は減少しない。さらに、ゴール間でのデータ転送をになう変数を同じプロセッサに確保すれば、ゴール間でのメモリ参照を省くことができる。

静的スケジューリングの目的はゴールの実行順を制御して、サスペンドを避けることである。サスペンドを避けるには、データ依存の順番に従って、ゴールを実行すれば良い。

ここで、データ依存関係を示すグラフが得られたとしよう。このグラフは、データとゴールをノード、依存関係をアークとする、有向グラフである。負荷の分割とスケジューリングはデータ依存関係グラフの分割問題に帰着する。

負荷分割は、同時に実行できず、データを共有しているゴールを一まとめにする作業である。データ依存グラフで、2 つのゴールを通るグラフのウォークがある場合、それらのゴール間にはデータ依存関係があるため同時に実行できない。したがって負荷分割は、グラフのウォークに沿ってグラフを分割することで実現できる。また、静的スケジューリングは、データ依存関係の順番に実行を行なうことで実現される。したがって、グラフのウォー

クに沿って実行すれば良いことになる。

本システムは、まずプログラムのモードの解析を行なう。このモードに基づいて、データ依存グラフを作成する。最後に、データ依存グラフを適切に分割する。この分割がそのままゴールとデータの分割になり、分割されたグラフ内の順番が静的な実行順となる。このようにして得られた分割とスケジュールを annotation としてプログラムに付加する。

5.2.2 モード解析

解析は、モード解析とデータ依存関係の解析の二つのフェイズからなる。モード解析では、述語の入出力モードを解析する。データ依存関係の解析では、モードの情報に基づいて、それぞれのクローズ間のデータ依存関係を解析する。

Committed-Choice 型言語に関しては、モード解析の手法が確立している [70, 29]。この手法では、ボトムアップにモードに対する束縛条件をプログラムから抽出し、トップダウンにこの束縛条件を組合せ、束縛充足問題として、述語のモード大域的に解析している。これは、計算時間のかかる作業である。また従来のモード解析では、モードを入力と出力のみとしている。本解析で必要としているのはデータとゴールの間の依存関係であり、入出力そのものでなくそのデータがいつ必要になるか、いつ作成されるか、という情報が必要なのである。このため、従来の入出力モードではわれわれの目的には不十分であると同時にオーバースペックでもある。[86] では、入力モードを単純な入力モードと構造データの内部がそのゴールで束縛されないばあいの入力とに分類してこのモード体系を拡張しているが、これでも不十分である。

われわれはこのため新たに "強入力"、"強出力" と呼ぶモードを追加したモードシステムを採用した。我々のモードシステムは以下の 5 つのモードからなる。述語のモードはモード記号のリストで表される。

例えば、下のプログラムを考えてみよう

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

この述語は、実行のためには第一引数を必要とし、実行後にはかならず第二引数を束縛する。したがってこの述語のモードは`[++, --]` となる。

表 5.1: モードシステム

記号	モード名	意味
++	強入力	そのゴールの実行に必要
+	(弱) 入力	そのゴールのサブゴールのどこかで必要
−−	強出力	そのゴールの実行で束縛される
−	(弱) 出力	そのゴールのサブゴールのどこかで束縛される
?	**unknown**	

(弱) 入力モード、(弱) 出力モードを完全に求めるためには、プログラムの全域解析が必要になるが、ここで必要なのは、強入力、強出力だけであるため、モード解析は局所的にしたがって高速に行なうことができる。

強入出力のモード解析は以下のルールで行なう。

1. ヘッドでその引数が変数でなければ、その引数は強入力
2. ヘッドでその引数に Non-Variable Annotation が付加されていれば、その引数は強入力
3. ヘッドでその引数が変数で、その引数にたいして値を束縛するシステム述語が存在すれば、その引数は強出力
4. ヘッドでその引数が変数で、その引数と、Non-Variable Annotation が付加されている変数とユニフィケーションするシステム述語が存在すれば、その引数は強出力
5. 上記以外であれば unknown

5.2.3 データ依存グラフの導出

次に、述語のモードからデータ依存関係を示す有向グラフを求める。あるゴールが強入力モードのデータを持つ場合、そのゴールがそのデータに依存することを意味し、あるゴールが強出力モードのデータを持つ場合、そのデータがそのゴールに依存することを意味する。

データ依存グラフは以下のステップで導出できる。

1. 変数をデータノードとする。
2. ボディゴールをゴールノードとする。
3. ボディゴールが強入力のモードの変数を持つ場合にはその変数のデータノードから、ボディゴールのゴールノードへアークを作る
4. ボディゴールが強出力のモードの変数を持つ場合にはボディゴールのゴールノードから、その変数のデータノードへアークを作る

5.2.4 データ依存関係グラフの分割

先に述べたように、負荷分割とスケジューリングは、グラフの分割に帰着する。分割されたグラフが、ひとつのプロセッサに割り当てられるため、グラフ上のゴールには並列度があってはいけない。したがって、グラフからゴールノードのみを取り出すとシーケンシャルになっていなければならない。

このために、グラフの分割は以下のように行なう:

1. 任意の最長のウォークを選ぶ。
2. 1. で選択したウォーク上のゴール / データを取り除きひとつのプロセッサに配置する。
3. 2. によって孤立したデータノードがあれば、そのノードも取り除き 2. と同じプロセッサに配置する。
4. 1-3 をすべてのデータ / ゴールノードがなくなるまで繰り返す。

図 5.3にこの様子を示す。このグラフは、3 つの最長のウォークをもっているが、ここではその内の B を選んでいる。つぎに孤立したデータノードを B と同じグループにいれ、これらをグラフから取り除く。この作業を繰り返すことで、最後の分割したグラフを得る。

5.2.5 アノテーション

グラフを分割した結果得られた負荷分割、静的スケジューリングはアノテーションを用いて表現し、コンパイラに指定する。

アノテーションは、変数、構造データ、ボディゴールに付加される。アノテーションは以下の書式で記述される。

Item @[*annotation1, annotation2,....*]

負荷分割に用いるアノテーションは下の 4 つである。

- `local`
  データ / ゴールをローカル IU に配置する。
- `on(` *label* `)`
  データがある IU を *label* が指すようにする。
- `to(` *label* `)`
  データ / ゴールを *label* が指す IU に配置する。
- `any(` *label* `)`
  データ / ゴールを最小負荷の IU に配置し、その IU を *label* が指すようにする。

静的スケジューリングに用いるアノテーションは下の一つである。

- `sequence(` *number* `)`
  同じ IU に配置されたゴールの中での実行の順番を示す。

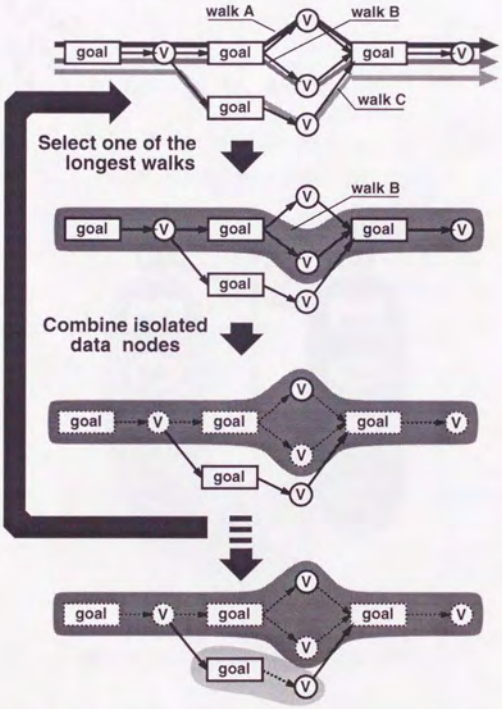


図 5.3: データ依存グラフの分割

5.2.6 負荷分割 / 静的スケジューリングの例

上述の方法を例を挙げて示す。下のプログラムは、n-queen のプログラムの一部である。

```
check(P, D, L, [Q|Lp0], Lp, A0, A):-
    add(Q, D, Sum),
    equal(Sum, P, R1),
    sub(Q, D, Dif),
    equal(Dif, P, R2),
    chk(R1, R2, P, D, L, Lp0, Lp, A0, A).
```

ここで add, sub, equal, chk のモードはそれぞれ [++, ++, --], [++, ++, --], [++, ++, --], [++, ++, ?, ?, ?, ?, ?, ?, ?] とする。

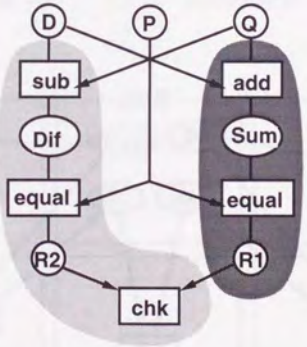


図 5.4: データ依存関係と負荷の分割例

図 5.4 はこのプログラム中のデータ依存グラフと、その分割を示している。この分割に従ってアノテーションを付加したのが下のプログラムである。

```
check(P, D, L, [Q|Lp0], Lp, A0, A):-
    add(Q, D, Sum @ [any(1)])
                  @ [to(1), sequence(1)],
    equal(Sum, P, R1 @ [to(1)])@
                   [to(1), sequence(2)],
    sub(Q, D, Dif @ [any(2)])
                  @ [to(2), sequence(1)],
```

```
equal(Dif, P, R2 @ [to(2)])
            @ [to(2), sequence(2)],
chk(R1, R2, P, D, L, Lp0, Lp, A0, A)
            @ [to(2), sequence(3)].
```

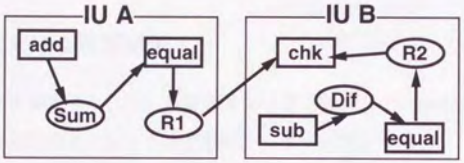


図 5.5: データ / ゴールの最適配置

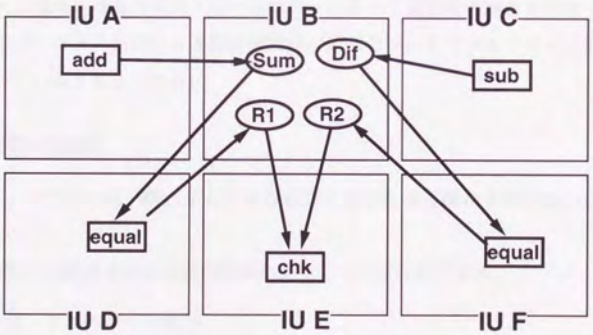


図 5.6: データ / ゴールのナイーブな配置

図 5.5,5.6は、ゴール / データの配置を示している。大きい四角は、IU を示し、ゴールとデータの間のアークはデータの参照を意味する。図 5.5は、このアノテーションの結果の最適な配置を、図 5.6 は、このアノテーションがない場合の配置を示している。最適配置では、リモートメモリ参照は一つしかない。しかし、アノテーションなしの場合はほとんどす

べての参照がリモートになる。また、実行の順序が制御されないため、頻繁にサスペンドが起こることが予想される。

このように、データ依存グラフの分割によって静的な負荷の分割とスケジューリングを行なうことができる。

5.3 PIE64 における粒度制御

fleng の実行粒度を調整するには、複数のスレッドを融合して一つのスレッドとすることでひとつスレッドの実行のランレングスを伸ばすことが必要となる。

スレッドを融合させる方法にはクローズのボディ部の述語同士を融合する方法と、クローズとその子述語を融合する方法が考えられる。本稿では、後者である**展開** (unfolding) を用いてスレッドの融合を行なう。

2.3.3節で述べた上田らの手法が、Committed-Choice 型言語特有のストリームというデータフローに埋め込まれたコントロールフローに沿って実行粒の融合を行なっているのに対して、このプログラム変換による粒度制御は、通常のコントロールフローに沿った実行粒の融合を行っていると考えられる。

5.3.1 述語の展開

展開とは、ボディー部で呼び出している述語を親である述語のなかに埋め込むことである (図 5.7)。

述語の展開には展開される述語の性質によって 2 つに分類できる。

- 述語にガード条件がない場合
  展開される述語にガードが存在しないまたは、ガードがあってもその時点でサスペンドすることがあり得ないので、ないとみなして良い場合。

  ```
  foo:- bar(a), baz ..
  bar(a):- bar1,bar2.
  ```

  bar は、ガードを持つが、このガードは foo によって常に解決されるのでないとみなして良く、下のように展開できる。

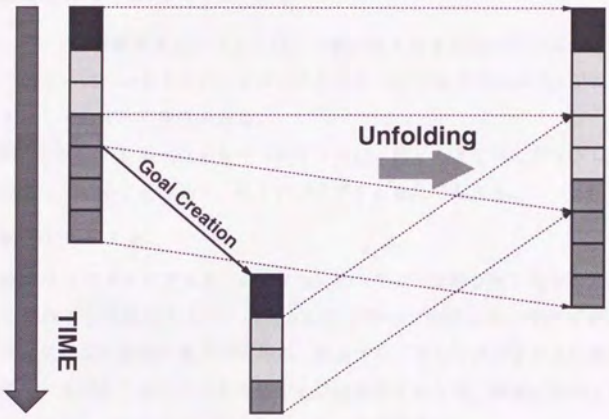


図 5.7: 述語の展開

```
foo:- bar1,bar2,baz ..
```

- 述語にガード条件がある場合
  展開される述語にガードがあり、サスペンドする可能性がある場合、そのガードを展開する親の述語のガードに追加する必要がある。この場合、同期をずらしてしまうことになるので、潜在的にデッドロックが生じる可能性があるので、注意しなければならない。

```
foo(A):- bar(A), baz ..
bar(a):- bar1,bar2.
```

bar のガードをfoo にずらして下のように展開できる。

```
foo(a):- bar1,bar2,baz ..
```

5.3.2 展開とデッドロックの可能性

展開時にガードを移動するということは、一般には入力と出力の依存関係を変更することである。このさいに、いままでデッドロックしなかったプログラムがデッドロックしてしまうように変化してしまう可能性がある。

ここで問題になるのは 並列性をもつプログラムは、ほとんどすべてデッドロックの可能性を潜在的にもっていることである。以下のプログラムを見てみよう。

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

このような簡単なプログラムでさえ、foo(X,X) というような呼び出し方をした場合には、デッドロックを生じる可能性をもつ。入力と出力を持つプログラムのすべてが、潜在的にデッドロックを生じる可能性があるのである。あるプログラムフラグメントに関して、その内部でデッドロックが生じないことを保証するのは容易であるが、外部にそのプログラムを含むようなデッドロックが生じないのを保証することはできない。

一般に、プログラム変換においては、変換後のプログラムが正常であることを保証する必要がある。しかしデッドロックに関しては生じないことを保証することは不可能である。

従って、われわれはもとのプログラムがデッドロックしない時には、変換後のプログラムもデッドロックしないことを保証する変換をおこなう。これを保証するには、変換前と変換後のプログラムがまったく同一の入出力依存関係をもつことが必要になる。

5.3.3 低負荷時の最適実行粒度

実行の最適な粒度は、プログラムのその時点での並列度に依存する。とくに低負荷時の静的粒度調整に関しては、並列度を抑制することにつながるため慎重に行なわなければならない。本項では、並列時と最適実行粒度の相関に関して考察する。

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クリティカルパスであると考えて、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。また、ゴールは左から右の順に生成すると考える。

ボディーゴール B_j の生成 / 送出にかかる時間を T_{Gj} とす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。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。

ここで、低負荷時すなわち空いている IU にゴールを送った時にすぐにそれが実行されると仮定することができる場合に、どの程度の粒度融合が可能かを考察する。この時の T_L

は、ランタイム・カーネルの実行状況にも依存するが、100 クロック程度である。T_G は、ゴールの引数の数にも依存するが、10 クロック程度である。したがって、低負荷時にあっても、クローズの中のボディゴールの数が 10 程度までは、展開してよいということになる。

5.3.4 並列度と最適実行粒度

前項での粒度と負荷との関係の考察を確認するために、実行粒度と実行スピードの相関を調べた。

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、解析結果に基づき展開を行なって粒度を調整し 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。

実験環境

実験は PIE64 上で 50 台のプロセッサを用いて行なった。

結果

実験の結果を図 5.8に示す。

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上昇するに従って、最適な粒度が上昇することがわかる。

ここで着目するべきは、プログラムの並列性が低い場合にも、最も速度が出る場合の粒度が 1 でないことである。すなわち、プログラムの並列度が低く、プロセッサの稼働率が下がってしまうような場合でも、ある程度の粒度調整を行なうことが望ましい。これは、前項の考察の結果と合致する。

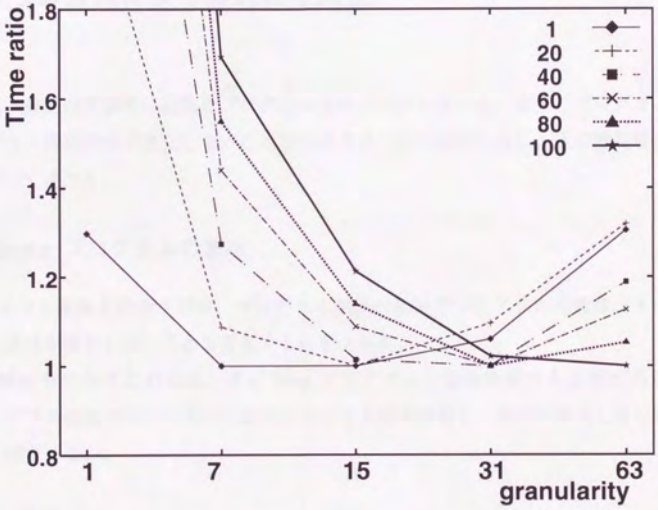

Time ratio
1.8
1.6
1.4
1.2
1
0.8
1
7
15
31
63
granularity
1
20
40
60
80
100

図 5.8: 並列度と粒度

第 6 章

プログラム変換による静的粒度調整

本章では、静的粒度調整に必要なプログラム変換について述べる。まず、プログラム変換の前後でデータ依存関係が変化しないことを保証するための枠組を示し、次に静的粒度調整の実装について述べる。

6.1 fleng プログラムの意味

プログラム変換を行なうには、プログラム変換の前後でプログラムの意味、すなわち入出力依存関係が変化しないことを保証する必要がある。

この解析を行なうためには、まず fleng プログラムの意味を定める必要がある。さらに、プログラム変換がプログラムの意味に与える影響を特定し、それが変化しない範囲を求めなければならない。

6.1.1 準備

本節では、[72, 70] に従って、fleng の表示的意味を定義する。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。

項と束縛

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。$T \in Term$ に対して $var(T)$ は T 中の変数の集合を与える。

束縛は Var から $Term$ へのマッピングとする。束縛全体の集合を Θ とする。

$$\forall v \in Var \forall \theta \in \Theta \exists t \in Term(V\theta \equiv t)$$

束縛は変数に対する項の代入の集合であると考えることもできる。

$$\theta = \bigcup_{i=1}^{n} (v_i \lhd t_i)$$

すべての代入の集合を S とする。

束縛上に関係矛盾 $\not\asymp$ を以下のように定義する。σ_1、$\sigma 2$ に対して

$$\exists X \in Var((X \lhd t_1) \in \sigma_1 \wedge (X \lhd t_2) \in \sigma_2)$$

を満たす X が存在し、t_1 と t_2 がユニフィケーション可能でないなとき σ_1、$\sigma 2$ が矛盾であるとし、$\sigma_1 \not\asymp \sigma 2$ と書く。

また、この条件を見たさない場合は、σ_1、$\sigma 2$ が無矛盾であるとし、$\sigma_1 \asymp \sigma 2$ と書く。

束縛上に関係 $\prec$ を以下のように定義する。σ_1、$\sigma 2$ に対して

$$\forall T \in Term(\exists \theta (T\sigma_1\theta \equiv T\sigma_2))$$

であるとき、$\sigma_1 \prec \sigma_2$ であるとする。

さらに、束縛上に関係 $\gg$ を定義する。束縛 σ_1、$\sigma 2$、E に対して、

$$(E \cup \sigma_1 \succ \sigma_2) \wedge (E \setminus \sigma_1 \not\succ \sigma_2)$$

が成り立つ時、E のもとで σ_1 は $\sigma 2$ に必要であるとし、$\sigma_1 \underset{E}{\gg} \sigma_2$ と書く。

ガード付ユニフィケーション

ガード付ユニフィケーションとは、束縛の対で、ある束縛と、その束縛が行なわれるために必要となる束縛の集合をならべたものである。例えば、$< \sigma \mid \theta >$ のように書く。この場合、σ が行なわれた後、θ が行なわれることを示す。すべてのガード付ユニフィケーションの集合を GU とする。

ガード付ストリーム

あるガード付ユニフィケーションの集合 Gs が以下の条件を満たすときガード付ストリームと呼ぶ。

1. $\forall < \sigma_1 \mid \theta_1 > \forall < \sigma_2 \mid \theta_2 > \in Gs((\sigma_1 \asymp \sigma_2) \wedge (\theta_1 \asymp \theta_2))$

2. $E = \{\sigma \cup \theta | \forall < \sigma \mid \theta > \in Gs\}$ の時、

$$\forall < \sigma \mid \theta > \in Gs((E \backslash \sigma) \not\asymp \sigma)$$

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。

例えば、次のプログラムを見てみよう。

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

このプログラムの実行結果として可能性のある入出力履歴は、`test` の第一引数を`H1` とすると、

$$\{\{< H1 = a \mid H2 = b >\}, \{< H1 = b \mid H2 = c >\}\}$$

の二つである。

6.1.2 同期付マージ

次に、ガード付ストリームの合成演算同期付マージを導入する。同期付マージは基本的にはガード付ストリーム同士の集合和を求める演算であるが、単なる集合和では、ガード付ストリームとしての条件を満たさなくなる。同期付マージはこの条件を満たすことを保証するための演算である。

同期付マージは、 $\|$ で表され、ガード付ストリーム $GS_1, ..., GS_n$ に対して、新たなガード付ストリーム $GS = (GS_1 \| ... \| GS_n)$ を与える。GU を $\bigcup_{i=1}^{n} GS_i$ とする。また、$E_g = \{\sigma | \forall < \sigma | \theta > \in GU\}$、$E_u = \{\theta | \forall < \sigma | \theta > \in GU\}$、$E = E_g \cup E_u$ とする。

$$GS = \{< \lim_{j=0}^{\infty} \sigma_j \mid \theta > | \forall < \sigma_0 \mid \theta > \in GU\}$$

$$\sigma_{i+1} \;=\; \exists s \in \sigma_i \left((\sigma_i \backslash s) \cup \alpha \middle| \forall < \alpha \mid \beta > \in GU \left((\beta \underset{E\backslash\beta}{\gg} \sigma_i) \right) \right)$$

同期付マージには交換則と結合則がなりたつ。すなわち、任意の $Gs_a, Gs_b, Gs_c \in GS$ に対して、

$$Gs_a \mid\mid Gs_b \equiv Gs_b \mid\mid Gs_a$$

$$(Gs_a \mid\mid Gs_b) \mid\mid Gs_c \equiv Gs_a \mid\mid (Gs_b \mid\mid Gs_c)$$

また、

$$Gs_a \mid\mid Gs_b \equiv Gs_a \mid\mid Gs_a \mid\mid Gs_b$$

であるから、

$$Gs_a \mid\mid Gs_b \mid\mid Gs_c \equiv (Gs_a \mid\mid Gs_c) \mid\mid (Gs_b \mid\mid Gs_c)$$

が成り立つ。

隠蔽

ここで、隠蔽という概念を導入する。ある束縛 θ に対して、ある束縛環境 E のもとである変数集合 V を通して観測が可能な束縛 θ' を与える演算 $\theta' = [\theta]_{V,E}$ を定義する。

$$[\theta]_{V,E} = \lim_{i \to \infty} \theta_i.$$

ここで、

$$\theta_0 \;=\; \theta.$$

$$\theta_{i+1} \;=\; ((\theta_i \backslash s)) | \exists s \in \theta_i (V(E\backslash\theta)\theta_i \equiv V(E\backslash\theta)(\theta_i \backslash s)))$$

例をあげて説明しよう。束縛 $\{X \lhd x\}$ は 束縛 $\{\}$ のもとでは、変数集合 $\{Y\}$ を通しては観測できない。

$$[\{X \lhd x\}]_{\{Y\},\{\}} = \{\}$$

しかし、束縛 $\{Y \lhd X\}$ の元では、観測可能である。

$$[\{X \lhd x\}]_{\{Y\},\{Y \lhd X\}} = \{X \lhd x\}$$

同様に、ガード付きストリームにも隠蔽を導入する。

$$\left[(\bigcup_{i=1}^{n} < \sigma_i \mid \theta_i >\right]_{V,E} = \bigcup_{i=1}^{n} < [\sigma_i]_{V,E} \mid [\theta_i]_{V,E}) >$$

隠蔽と同期つきマージの間には、以下の関係が成り立つ。

$$[gs_1]_{V,E} \mid\mid [gs_2]_{V,E} \Rightarrow [gs_1 \mid\mid gs_2]_{V,E}$$

また、$var(gs_1) \cap var(gs_2) \equiv V_1$ のとき、以下の関係が成立する。

$$[gs_1 \mid\mid [gs_2]_{V_1,E_1}]_{V_2,E_2} \Longrightarrow [gs_1 \mid\mid gs_2]_{V_2,E_2 \cup E_1}$$

プログラムの意味

クローズ $H : -B_1, .., B_n.$ に対して、このクローズのガードを σ_0, ユニフィケーションを θ_0 とし、述語 B_i の意味を $Gs_i = \{< \sigma_{ij} | \theta_{ij} >\}$ とすると、このクローズの意味 Gs は以下で定義される。

$$[(\{< \sigma_0 \mid \theta_0 >\} \mid\mid Gs'_1 \mid\mid \ .. \ \mid\mid Gs'_n)]_{var(H),E}$$

ただし、

$$Gs'_i = \forall j(\{< (\sigma_0 \cup \sigma_{ij}) | \theta_{ij} >\})$$
$$E = \left(\bigcup_{i=1}^{n} (\sigma_{ij} \cup \theta_{ij})\right) \cup \sigma_0 \cup \theta 0$$

σ_0 をすべての、ガードつきユニフィケーションの右項に追加するのは、コントロール依存、すなわち、σ_0 が満たされクローズが実行されない限り、ボディー部のクローズの動作は生じないことを表現している。

このように、サブゴールの意味からクローズの意味を合成することが可能である。

6.1.3 ボディ部展開の意味

クローズ $H : -B_1, B_2, .., B_n.$ の n 番目のボディを展開することを考える。fleng ではボディの順番には意味がないので、n 番目にすることで、一般性は失われない。親クローズのガード条件、ユニフィケーションを σ_0, θ_0, ボディーゴールの B_i の意味を $gs_i = \{< \sigma_{ij} \mid \theta ij >\}$ とする。また、ボディゴール B_h に対応するクローズは $BH : -BB_1, .., BB_m.$ であるとし、ガード条件を意味する束縛を ψ、システム述語の実行による束縛を ϕ とする。BB_p の意味を $gs_p = \{< \sigma_{pq} \mid \theta pq >\}$ とする。BE とする。

B_n に対応するクローズの意味は、

$$\left[< \psi \mid \phi > \ || \left(\mathop{||}_{p=1}^{m} < \psi \cup \sigma_{pq} \mid \theta_{pq} > \right) \right]_{var(BH),BE}$$

であるから、

展開前のゴールの意味は

$$Gs = [Gs_1 \ || \ Gs_2 \ || \ Gs_3]_{var(H),E}$$

ただし、

$$\begin{aligned} Gs_1 &= \{< \sigma_0 \mid \theta_0 >\} \\ Gs_2 &= (\mathop{||}_{i=1}^{n-1} < \sigma_0 \cup \sigma_{ij} \mid \theta_{ij} >) \\ Gs_3 &= \left[< \sigma_0 \cup \psi \mid \phi > \ || \left(\mathop{||}_{p=1}^{m} < \sigma_0 \cup \psi \cup \sigma_{pq} \mid \theta_{pq} > \right) \right]_{var(BH),BE} \end{aligned}$$

Gs_3 の隠蔽は省略できて、

$$Gs_3 = < \sigma_0 \cup \psi \mid \phi > \ || \left(\mathop{||}_{p=1}^{m} < \sigma_0 \cup \psi \cup \sigma_{pq} \mid \theta_{pq} > \right)$$

となる。

ガードを移行しない展開

5.7で述べたように、展開はガードを移行して展開する場合と、ガードを移行しない場合の 2 種に大別できる。ガードを移行せずに展開することは、ボディーゴールのガード条件を省略して、ボディークローズの内容を展開することであるから、この場合の展開後のセマンティクスは下のようになる。

$$Gs' = [Gs'_1 \mid\mid Gs'_2 \mid\mid Gs'_3 \mid\mid Gs'_4]_{var(H),E}$$

$$\begin{aligned}
Gs'_1 &= \{< \sigma_0 \mid \theta_0 >\} \\
Gs'_2 &= \left(\mathop{\Big\|}_{i=1}^{n-1} < \sigma_0 \cup \sigma_{ij} \mid \theta_{ij} > \right) \\
Gs'_3 &= < \sigma_0 \mid \phi > \\
Gs'_4 &= \left(\mathop{\Big\|}_{p=1}^{m} < \sigma_0 \cup \psi \cup \sigma_{pq} \mid \theta_{pq} > \right)
\end{aligned}$$

この場合に展開の前後でセマンティクスが変化しない、すなわち $Gs = Gs'$ になるための条件は

$$\theta_0 \succ \psi$$

である。

直観的にいえば、親ゴールが子ゴールの実行環境を保証しているわけである。

ガードを移行する展開

ガードを移動する展開の意味は以下のようになる。クローズ内の一つのボディーゴールを展開することは、そのボディーゴールのガード条件をクローズのガード条件に追加することを意味する。

$$Gs'' = [Gs''_1 \mid\mid Gs''_2 \mid\mid Gs''_3 \mid\mid Gs''_4]_{var(H),E}$$

$$
\begin{aligned}
Gs_1'' &= \{< \sigma_0 \cup \psi \mid \theta_0 >\} \\
Gs_2'' &= \left(\mathop{\|}_{i=1}^{n-1} < \sigma_0 \cup \psi \cup \sigma_{ij} \mid \theta_{ij} > \right) \\
Gs_3'' &= < \sigma_0 \cup \psi \mid \phi > \\
Gs_4'' &= \left(\mathop{\|}_{p=1}^{m} < \sigma_0 \cup \psi \cup \sigma_{pq} \mid \theta_{pq} > \right)
\end{aligned}
$$

この場合の条件は、以下の二つである。まず、

$$[\theta_0]_{var(H),E} \equiv \{\}$$

がなりたち、かつ、すべての、$n \in [1 : n-1]$、m に対して、

1. $(\phi \underset{E}{\gg} \sigma_{nm})$
2. $([\theta_{nm}]_{var(H),E} \equiv \{\})$

のいずれかが成り立つ。

直観的には、前者の条件は、ゴールのシステム述語が同様に外部に出力をもたないので影響を与えないことを意味する。後者の条件は、1 はそのボディゴールは結局展開するゴールの実行結果を待たないと実行できないので、依存関係が変化しないことを、2. はそのボディゴールが外部に出力を持たないので、結果としてセマンティクスに影響を与えないことを意味する。

6.2 プログラム展開の例

本節では、前節で示した条件を実際の fleng プログラムの例をあげて示す。本節では、説明を容易にするために、fleng プログラムのガード部のユニフィケーションを GHC のガードゴールのようにコミット・バーを用いて示しているが、ガードゴールに相当する部分にはユニフィケーションしか用いていないので、fleng で記述するのと等価である。

6.2.1 ガードの移行なしの展開の例

下のプログラムで foo 内に bar を展開するケースを考えてみよう。

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

$\sigma_0 = \{\}, \theta_0 = \{B \lhd a\}, \psi = \{B \lhd a\}$ であり、

$$\sigma_0 \cup \theta_0 = \{B \lhd a\} \succ \psi$$

であるから、明らかに条件 1 を満たす。したがって、ガードなしで移行することが許される。

6.2.2 ガードの移行を行なう展開の例

例 1: 展開可能な例

下のプログラムで foo 内に bar を展開するケースを考えてみよう。

```
foo(X, A, B, C):- X = x |
                  bar(A, B), baz(B, C)
bar(A, B):- A = a | B = b.
baz(B, C):- B = b | C = c.
```

展開した結果は

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

このとき、

$\sigma_0 = \{X \lhd x\}, \theta_0 = \{\}, \psi = \{A \lhd a\}, \phi = \{B \lhd b\}$ $\sigma_{11} = \{B \lhd b\}, \theta_{11} = \{C \lhd c\}$ であり、

この場合は、

$$(\phi \underset{E}{\gtrdot} \sigma_{11}) \wedge [\theta_0]_{Var(H),\{\sigma_0,\theta_0,\sigma_{11},\theta_{11}\}} \equiv \{\}$$

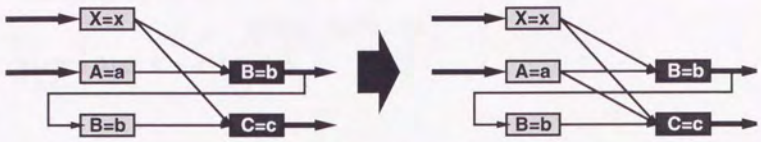


図 6.1: 例 1: 展開可能な例

であるから展開の条件を満たす。

展開しない場合のセマンティクスは、

$$\left[\begin{array}{l} < \{X \lhd x, A \lhd a\} \mid \{B \lhd b\} > \\ \parallel < \{X \lhd x, B \lhd b\} \mid \{C \lhd c\} > \end{array}\right]_{\{X,A,C\},E}$$

であり、展開した場合には、

$$\left[\begin{array}{l} < \{X \lhd x, A \lhd a\} \mid \{B \lhd b\} > \\ \parallel < \{X \lhd x, A \lhd a, B \lhd b\} \mid \{C \lhd c\} > \end{array}\right]_{\{X,A,C\},E}$$

となり、一見異なるが、マージした結果はどちらも

$$\left\{\begin{array}{l} < \{X \lhd x, A \lhd a\} \mid \{B \lhd b\} >, \\ < \{X \lhd x, A \lhd a, B \lhd b\} \mid \{C \lhd c\} > \end{array}\right\}$$

となり、セマンティクスは変化しない。

例 2: 展開できない例

前節のプログラムに良く似た下のプログラムで、bar を展開することを考えてみよう。

```
foo(X, A, B, C, D):- X = x |
                      bar(A, B), baz(C, D).
bar(A, B):- A = a | B = b.
baz(C, D):- C = c | D = d.
```

展開した結果は以下のようになる。

```
foo(X, A, B, C, D):- X = x, A = a|
                     B = b, baz(C, D).
baz(C, D):- C = c | D = d.
```

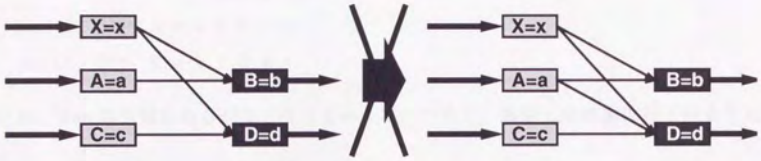


図 6.2: 例 2: 展開できない例

このとき、$\sigma_0 = \{X \lhd x\}, \theta_0 = \{\}, \psi = \{A \lhd a\}, \phi = \{B \lhd b\}, \sigma_{11} = \{C \lhd c\}, \theta_{11} = \{D \lhd d\}$ である。

この場合は、

$$(\phi \not\geq_E \sigma_{11})$$

であるため、条件を満たさない。

セマンティクスを比較すると、展開前が

$$\left\{ \begin{array}{l} < \{X \lhd x, A \lhd a\} \mid \{B \lhd b\} >, \\ < \{X \lhd x, C \lhd c\} \mid \{D \lhd d\} > \end{array} \right\}$$

であるのに対して展開後は、

$$\left\{ \begin{array}{l} < \{X \lhd x, A \lhd a\} \mid \{B \lhd b\} >, \\ < \{X \lhd x, A \lhd a, C \lhd c\} \mid \{D \lhd d\} > \end{array} \right\}$$

となり、変化しているのがわかる。これによって、例えば、`foo(x, A, P, P, D).` というような呼び出しを行なった場合に、もとのプログラムでは生じなかったデッドロックを生じる。

例 3: 隠蔽によって展開が可能になる例

前節の例とほとんど同じだが、隠蔽によって展開が可能になる例を示す。

```
foo(X, A, B, C):- X = x |
                    bar(A, B), baz(C, D).
bar(A, B):- A = a | B = b.
baz(C, D):- C = c | D = d.
```

変化は、foo の引数から D がなくなっているだけである。展開した結果は以下のようになる。

```
foo(X, A, B, C):- X = x, A = a|
                    B = b, baz(C, D)
baz(C, D):- C = c | D = d.
```

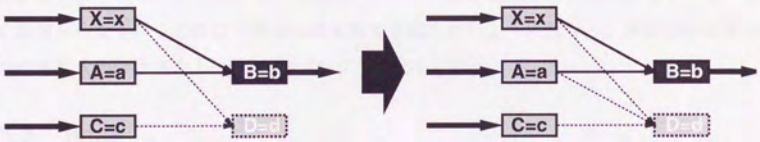


図 6.3: 例 3: 隠蔽によって展開が可能になる例

この場合も同様に、

$$(\phi \not\gg \sigma_{11})_{/E}$$

であるが、

$$[\theta_{11}]_{Var(H),E} \equiv \{\}$$

であるために、展開の条件を満たす。

セマンティクスも同期付マージまで前節の例と同じであり、一見展開の前後で異なるかに見えるが、束縛 $\{D \lhd d\}$ が外部から見えないためセマンティクスは両方とも

$$\{< \{X \triangleleft x, A \triangleleft a\} \mid \{B \triangleleft b\} >\}$$

になる。

6.3 粒度調整システム

本節では、粒度調整システムの実装について述べる。

6.3.1 粒度調整システムの概要

図 6.4に粒度調整プログラムの構成を示す。本プログラムは、まず対象プログラムを内部形式に変換する。そして、内部形式に対して意味解析を行ない、展開する場所を決定する。展開場所は、高負荷時用、低負荷時用それぞれに対して適切な最大粒度を外部から与え、その粒度を越えないように決定する。展開も内部形式上で行なわれる。ボトムアップの展開が終了したあと、ボトムアップの展開によって状態が確定した部分に関してトップダウンの展開を行なう。この際にも展開の最大粒度を越えないように行なう。展開が終了するとその内部形式から後述の fleng−− のプログラムとして出力する。

6.3.2 fleng−−

調整プログラムが出力する fleng のプログラムには、システム述語の実行順序や同期をともなわない分岐が必要になる。このため fleng のセマンティクスをより限定し、決定性をもたせた言語 fleng−− を設定し、プログラム変換の対象言語とした。

fleng−− は、システム述語の実行順序を左から右であることを保証する。fleng ではこの仮定は成り立たないので、例えば以下のようなプログラムとしては正常に動作する保証はない。しかし、fleng−− では正常な動作を保証する。

```
foo(#A, #B, C):-
   compute('+', A, B, D),
   compute('+', D, 1, C).
```

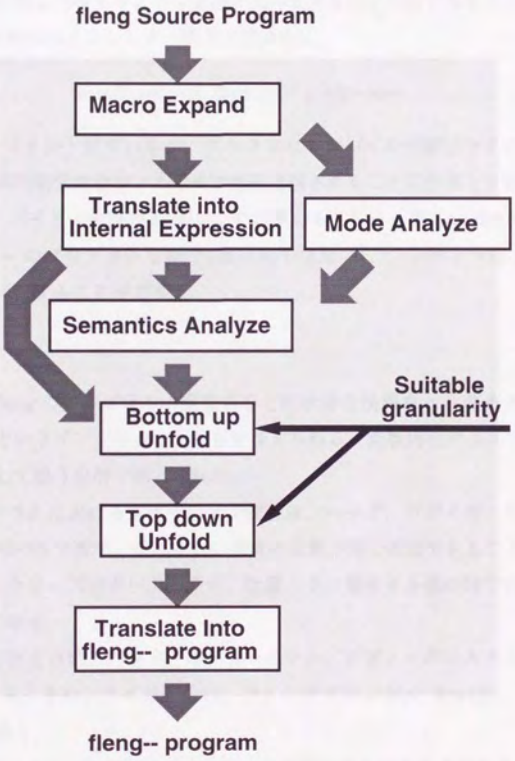


図 6.4: 粒度調整システムの構成

fleng−− は、同期をともなわない分岐を表現することができる。fleng では分岐はすべて述語のヘッド部で行なわれるため、分岐を行なうには、ゴール作成をおこなう必要がある。このオーバーヘッドをさけるためには、一つのゴールの内部で非同期な分岐を実現できる必要がある。非同期な分岐とシステム述語の実行順序も左から右であることを保証する。

この非同期な分岐は以下のシンタックスで表される。

condition --> *then-part* ; *else-part*

これは、通常の if マクロと似ているが、if マクロはコンパイルの前にマクロ展開されて処理されるが、この非同期分岐はコンパイラで直接処理されることに注意して欲しい。

われわれのコンパイラは実際は fleng−− を対象としている。正しい fleng のプログラムは、正しい fleng−− のプログラムなので (逆は真ではない)、コンパイラは fleng プログラムを正常にコンパイルすることができる。

内部形式

内部形式は、fleng のプログラムの変数名などの余分な情報をとりのぞき、プログラムでは変数名の一致というインプリシットな形で与えられる、変数同士のユニフィケーションを明示的な情報として扱う目的で導入された。

変数は、プログラムにおける位置で表す。位置は、ヘッド、ボディゴール、システム述語のいずれかからのパスで表す。それぞれの位置の変数が同じ名前であることは、明示的に unify であらわす。クローズのガード条件も、位置とその要求する値の対で示す。変数パスの記法を、表 6.1に示す。

変数ルートのそれぞれは、ヘッド、ボディーゴール、ボディーのシステム述語、適当な内部変数に対応する。また、リストセルは、ファンクタシンボル `$cell` 、アリティ 2 のファンクタとして扱う。

実際の例を見てみよう。下のプログラムは、2 引数の大きい方を求めるものである。

このプログラムを内部形式に展開したものを図 6.6に示す。

表 6.1: 変数パス

変数パス	:=	変数ルート
	|	変数パス, ファンクタ名, ファンクタナンバ
変数ルート	:=	$head($N$)
	|	$body($N$,$M$)
	|	$sys($N$,$M$)
	|	$iv($N$)
ファンクタ名	:=	$vec(ファンクタシンボル、アリティ)

```
max(A,B,C) :-
    gt(A,B,D),
    $1$max(D,C,A,B).
$1$max(true,C,A,B) :-
    C = A.
$1$max(false,C,A,B) :-
    C = B.
gt(# A,# B,R) :-
    compute(>,A,B,R).
```

図 6.5: プログラム例

```
predicate(func(max,3),[
      gbu(nil,
           [func(gt,3),func($1$max,4)],
           nil,
           [unify([$body(1,1)],[$head(1)]),
            unify([$body(1,2)],[$head(2)]),
            unify([$body(1,3)],[$iv(4)]),
            unify([$body(2,1)],[$iv(4)]),
            unify([$body(2,2)],[$head(3)]),
            unify([$body(2,3)],[$head(1)]),
            unify([$body(2,4)],[$head(2)])
           ])
      ]).
predicate(func($1$max,4),[
      gbu([guard([$head(1)],$sym(true))],
           nil,
           [func(=,2)],
           [unify([$sys(1,1)],[$head(2)]),
            unify([$sys(1,2)],[$head(3)]),
            unify([$sys(1,1)],[$sys(1,2)])
           ]),
      gbu([guard([$head(1)],$sym(false))],
           nil,
           [func(=,2)],
           [unify([$sys(1,1)],[$head(2)]),
            unify([$sys(1,2)],[$head(4)]),
            unify([$sys(1,1)],[$sys(1,2)])
           ])
      ]).
predicate(func(gt,3),[
      gbu([guard([$head(1)],$any),
           guard([$head(2)],$any)],
           nil,
           [func(compute,4)],
           [unify([$sys(1,1)],$sym(>)),
            unify([$sys(1,2)],[$head(1)]),
            unify([$sys(1,3)],[$head(2)]),
            unify([$sys(1,4)],[$head(3)]),
            unify([$sys(1,4)],$atom)
           ])
      ]).
```

図 6.6: 内部形式の例

6.3.3 意味解析プログラムの概要

意味解析プログラムは、プログラムの意味をボトムアップに求めるものである。クローズの意味は、そのクローズのサブゴールが実際にどのクローズにマッチするかによって変化する。したがって、クローズの意味は、ボディーゴールにマッチするクローズの、すべての組合せを合成したものになる。このようにサブゴールの意味をを限定したクローズをクローズインスタンスという。マージは、前節でのべた同期付マージを用いて行なう。この際同時に、それぞれのボディゴールが、展開可能であるかどうかを判断する。また、同時にどこまで深く展開が可能かを同時に調べる。

それぞれのクローズで unfold が可能な条件だけがわかれば、unfold を繰り返すことによって、可能な限りの粗粒度化が可能である。しかしこれには、unfold をするたびに再び unfold が可能かどうかのチェックを行なう必要がある。これをさけるために、一度の解析で、あるクローズの下がどれだけ展開が可能かを調べている。理論的には、それぞれのクローズに対して、すべてのレベルで展開の可能性を考えることが可能である。しかし、この手法は大量の計算時間を必要とすることが予測されるうえ、実用上必要性があまりないと考えられるので、下に述べるように展開を 2 段階のみ考えることにした。

本プログラムでは、それぞれのクローズインスタンスに対して、極大展開という概念を導入した。極大展開は、意味が変化しない範囲で最大限の展開を行なうものである。極大展開では、各ボディーゴールのインスタンスに対して

- 極大に展開する
- 一段階だけ展開する
- そのまま

のいずれかを選択する。この際に各ボディーゴールのインスタンスの依存関係などを考慮し、最大限になるようにする。

本プログラムの解析手法は本質的にボトムアップであり、ボディーゴールに対する完全な情報を期待している。したがって、ループが存在したり、解析対象のファイルに存在しないボディーゴールが呼び出されると、解析がストップしてしまう。そこで、本プログラムで

は補助的にモード解析の結果を用いる。静的負荷分割の際に用いた、局所モード解析の結果を用いて、疑似的に不明なボディーゴールの挙動を表現して、解析を進める。

6.3.4 意味解析プログラム

意味解析プログラムでは、クローズの意味を、次の三つの組で表現する。

- ガードグループ
  複数のガードからなる組。1 つ目のガードグループ (ガードグループ番号 0 番) が、そのクローズそのもののガードを表す。2 つ目以降は、ボディゴールによるガードを表す。
- ユニフィケーション
  そのクローズ及び、ボディーゴールによって外部に提供されるユニフィケーション。
- 依存
  ユニフィケーションとガードグループの間の依存関係。ガードグループ 0 にのみ依存するユニフィケーションは、そのクローズのみによって提供されるユニフィケーションである。

図 6.5の gt を例にとろう。ガードは、第一引数と第二引数がなにかに束縛されている、という条件である。本プログラムでは、$any でこの条件を表現している。ユニフィケーションとして、第三引数になにかを束縛する。このガードと、ユニフィケーションの間に依存関係がある。従ってこのクローズの三つ組は、下のようになる。

```
gb(func(gt,3,1),
    [[
        guard([$head(1)],$any),
        guard([$head(2)],$any)
    ]],
    [
        unify([$head(3)],$atom)
    ],
    [[0]])
```

このような三つ組をボディごとにもとめ、これを同期つきマージで合成する。
合成の手順は以下のようになる。

1. おのおののガードに関して、その条件が内部のユニフィケーションで満たすことができるかを調べる。できるものを、内部ガード、できないものを外部ガードと呼ぶ。内部ガードに関してはその条件を満たすために必要なユニフィケーションの組を求める。

2. 1 の結果とユニフィケーションからガードへの依存を用いて、あるユニフィケーションが生じるのに必要になる外部ガードの集合を求める。

3. 外部に露出するユニフィケーション (外部ユニフィケーション) を求める。

4. 外部に露出するユニフィケーションに必要な外部ガードを求める。

この様子を 6.7に示す。

以下それぞれ詳しく見ていく。

ガード条件を満たすユニフィケーションの組を求める ここで問題になるのは、ガードを満たすユニフィケーションを求める方法である。通常の言語では、変数に対する代入とその値の使用の関係は明白である。しかし、単一代入変数には変数同士の束縛が存在するため、ある変数に値を束縛する方法は無数に存在する。このため、ガードを満たすユニフィケーションは、プログラムの字面からはわからない。

この問題を解決するため、本プログラムでは、変数ネットワークを導入した。変数ネットワークは、変数のそれぞれの位置での出現をノード、それらの名前のが同じことによる同値関係、または明示的なユニフィケーションによる同値関係をアークとするグラフである。ガード条件を満たす、ユニフィケーションの組は、ガードの変数を示すノードと、ガードが要求する値のノードを結ぶパスを構成するアークの組である。

変数ネットの例を図 6.8に示す。この例は、次のプログラムの変数 A に関するネットである。

```
test([A], C):-
    A = C,
    C = B,
    test2(A, B).
```

この時、内部のユニフィケーションでは満たすことのできないガードが検出されることがある。このようなガードを外部ガードとして登録する。また、満たすことができるかど

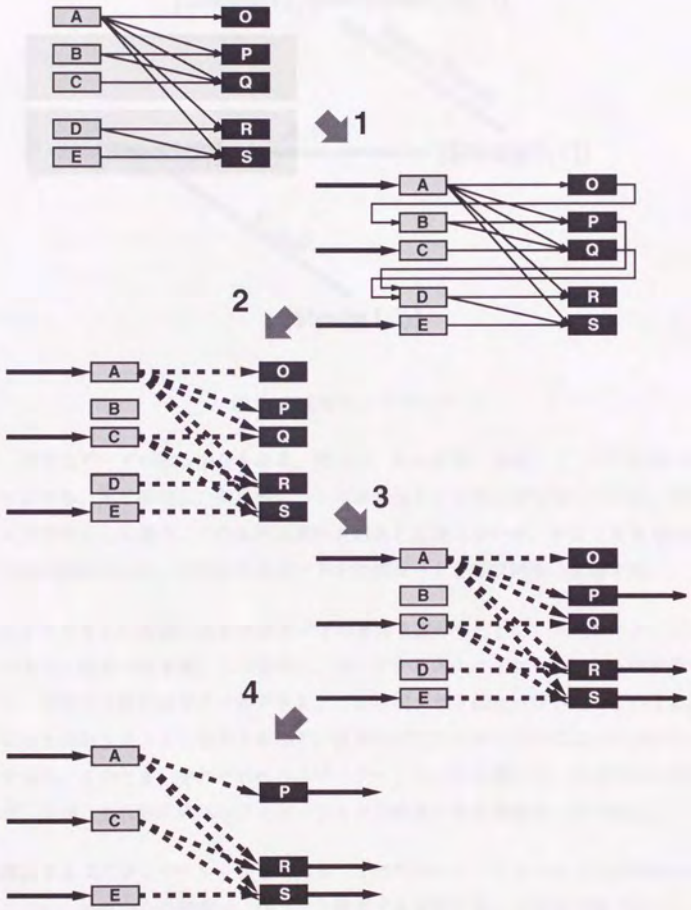


図 6.7: 同期つきマージによる合成

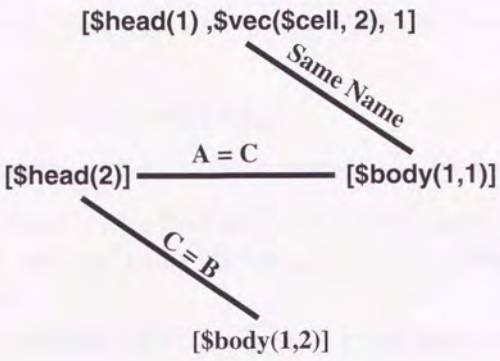


図 6.8: 変数ネットワーク

うかが、不明なガードがある場合もある。例えば、ある変数に特定のシンボルを待つガードがあったとする。同じ変数に、何らかのシンボルを返すことだけがわかっている、ユニフィケーションが存在した場合、この条件は満たされるとは限らないが、少なくとも block しないことだけは保証される。このようなガードを内部ガードと呼び同様に登録する。

ユニフィケーションの前提になる外部ガードの集合を求める 1で、ユニフィケーションから、ガードへの依存が求まる。この依存と、ガードからユニフィケーションへの依存によって、クローズ内での依存のグラフができる。この有向グラフにループがあるということは、デッドロックがおきることに相当するので、正常なプログラムに対してはループがないことが仮定できる。このとき、それぞれのユニフィケーションから遡って、到達することのできる外部ガードが、それぞれのユニフィケーションの前提になる外部ガードである。

外部に露出するユニフィケーションを求める すべてのユニフィケーションが外部から観測可能ではない。外部からの参照は、ヘッドを構成する変数を通してのみ可能であるからである。しかし、ヘッドの変数に直接束縛をするような束縛以外にも、外部から観測可能なユニフィケーションは存在する。また、変数間のユニフィケーションなども外部に露出する可能

性がある。

本プログラムでは、これを変数ネットワークを用いて実現した。アルゴリズムを下に示す。

1. 変数パスをヘッドの n 番目の引数にする。
2. パスから到達可能な、実体オブジェクトまたは変数パスマークを探す。
3. どちらもなければ、そこに変数パスをマークとしてつける。実体オブジェクトがアトムであれば、そのパスにアトムを束縛するユニフィケーションが外部に露出していることがわかる。

   変数パスマークがついていれば、そのマークの示すパスと現在のパスの間の変数同士のユニフィケーションがあることがわかる。

   実体オブジェクトが、構造体であれば、それぞれの要素のパスを作成し、それぞれに対して、2 以降を行なう。

このアルゴリズムをヘッドの引数すべてに対して行なうことによって、外部に露出するユニフィケーションを求める。

図 6.9 に変数ネットを用いた変数同士のユニフィケーションの検出の様子を示す。

外部に露出するユニフィケーションに必要な外部ガードを求める 3 で、変数パスノードから、実体オブジェクトを示すノードを結ぶパス上のユニフィケーションが、外部ユニフィケーションを構成するユニフィケーションである。従って、これらの要素ユニフィケーションに前提となる外部ガードの和集合が外部ユニフィケーションの前提となるガードということになる。

以上のようにして外部ガードと外部ユニフィケーションとそれらの間の依存関係がわかる。これらが、あらたにこのクローズの意味を構成する。

6.3.5 展開可能性の判断

前項で述べたマージの結果を用いて、極大展開の際の各ボディゴールの展開の深さを決定する必要がある。このためには、それぞれのボディゴール間の依存関係を解析する必要が

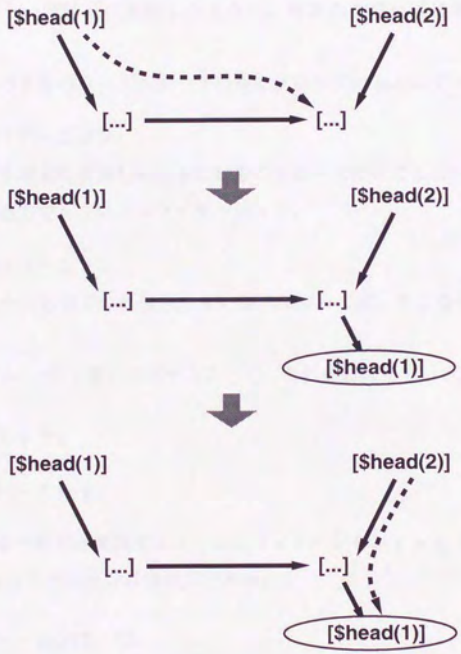


図 6.9: 変数ネットを用いた変数同士のユニフィケーションの検出

ある。

このためにはまず、ボディゴールの提供するユニフィケーションの分類する必要がある。ユニフィケーションを以下の三つに分類する。

- 強出力ユニフィケーション
  ボディゴールを一段階だけ展開したときに、対象のクローズに生じるユニフィケーション。
  ボディゴールの 0 番のガードグループのみに依存しているユニフィケーション
- 中出力ユニフィケーション
  ボディゴールを極大に展開したときに対象のクローズに生じるユニフィケーション
  強出力でも弱出力でもないユニフィケーション。
- 弱出力ユニフィケーション
  デッドロックを引き起こす危険性のない展開によっては、生じないユニフィケーション
  疑似ガードグループ -1 番に依存することで、これを表現している。

例をあげて説明しよう。

```
foo(x, Y):- Y = y.
```

このクローズ foo を一段だけ展開すると、ユニフィケーション Y = y が与えられる。したがって、このユニフィケーションは強出力である。

```
foo(X, Y):- bar(X, Y).
bar(x, Y):- Y = y.
```

このクローズ foo でも同様に ユニフィケーション Y = y が与えられるが、これには 2 段階の展開が必要となる。したがって、このユニフィケーションは中出力である。

```
foo(x, Y):- bar(A, B), baz(A, B, Y).
bar(a, B):- B = b.
baz(A, B, Y):- A = a, baz1(B, Y).
baz1(b, Y):- Y = y.
```

このクローズ foo でも最終的には、 ユニフィケーション Y = y が出力される。しかし、これは展開では不可能である。これは、依存関係が、 baz → bar → baz1 というようになっているためで、展開のみでこの依存関係にそったスケジューリングができないためである。このようなユニフィケーションを弱出力という。

前項で述べた手法の 1 の段階すなわち、ガード条件を満たすユニフィケーションの組を求める際に、それぞれのガードが依存する、強出力、弱出力ユニフィケーションがわかる。また、それぞれのユニフィケーションが依存するガードは、わかっているので、これらから、ユニフィケーション間の依存が導ける。これを利用して、それぞれのボディゴールの展開レベル間の依存関係がわかる。

例えば、あるボディ B_n の強出力が、 B_m の強出力にのみ依存している場合は、 B_m は、B_n を極大展開したあとでないと極大展開できない。ボディ B_n の強出力が、 B_m の弱出力にのみ依存している場合には、 B_m は、 B_n を 1 段展開したあとであれば、極大展開が可能である。このように、各ボディゴールの極大展開、1 段展開をノードとしてそれらのあいだの依存グラフが作成できる。これを展開フローと呼ぶ。展開フローのそれぞれのノードには、展開する際に必要となる外部ガード、内部ガードも同時に記述する。

図 6.10に、 6.5の展開フローを示す。

ここには、 max の二つのクローズインスタンスの二つの展開フローを示している。どちらも、 gt を展開した後でなら、 1max を展開できるという意味である。

6.3.6 展開プログラム

展開プログラムは、意味解析プログラムの結果を用いて、内部形式の上で展開の上限にみちるまで展開を行なう。展開の上限値は、引数としてこのプログラムに与える。高並列時、低並列時それぞれに応じて、この上限値を切替えることで、適切なコードを得る。

展開はトップダウンに行なう。一つのクローズの意味は、一般には複数の意味の論理和になる。したがって、展開を行なう際にこれらの合流をおこなわなければならない。

例えば、図 6.5の最大値を求めるプログラムを考えてみよう。このプログラムの max の意味は、ボディゴール gt が第三引数に束縛する値によって二つにわかれたものの和になる。 fleng のプログラムとしてはこの二つを合成したプログラムを作成しなければならない。

```
[
    [
        cond(unfold(1),func(gt,3,1),
                [guard([$body(1,1)],$any,g(1,0,1)),
                 guard([$body(1,2)],$any,g(1,0,2))],
                nil)
    ],[
        cond(unfold(2),func($1$max,4,1),
                nil,
                [guard([$body(2,1)],$sym(true),g(2,0,1))
    ]
],[
    [
        cond(unfold(1),func(gt,3,1),
                [guard([$body(1,1)],$any,g(1,0,1)),
                 guard([$body(1,2)],$any,g(1,0,2))],
                nil)
    ],[
        cond(unfold(2),func($1$max,4,2),
                 nil,
                 [guard([$body(2,1)],$sym(false),g(2,0,1))])
    ]
]
```

図 6.10: 極大展開の指定

このためにまず、前項で作成した展開フローの合成を行なう。展開フローを前方から比較していき、一致しない点で分岐をするような展開フローを作成する (図 6.11)。

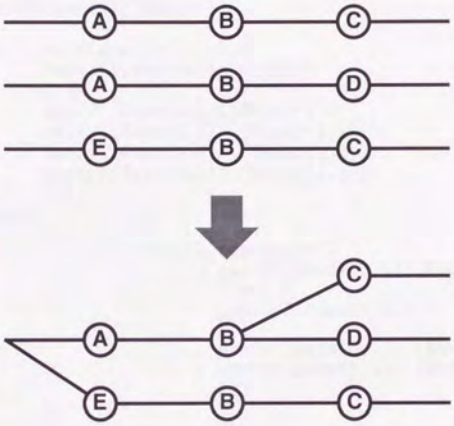


図 6.11: 展開フローの合成

展開は、展開フローに従っておこなう。このとき、パラメータとしてあたえられた展開の上限値と、展開した際の粒度の大きさを比較して、上限値を越えるのがわかると、そこで展開を中止する。

展開した結果の内部形式を図 6.12に示す。

6.3.7 fleng−− への変換

展開が終了した内部形式からコンパイラが処理することのできる fleng−− のプログラムへの変換を行なう。

前項の例を変換したものを図 6.13に示す。

展開フローの分岐が生じている点では、前述の non-blocking if 文をもちいて、分岐を行なう。分岐条件になるのは、展開フローの内部ガード条件である。展開フローの外部ガー

```
clause(func(max,3,1),gbu([
        guard([$head(1)],$any),
        guard([$head(2)],$any)
        ],[
        label(1,subclause(nil,nil,[
                label(4,func(compute,4))
                ],[
                unify([$sys(4,1)],$sym(>)),
                unify([$sys(4,2)],[$body(1,1)]),
                unify([$sys(4,3)],[$body(1,2)]),
                unify([$sys(4,4)],[$body(1,3)])
                ])),
        blanch([
                [
                        label(2,subclause([
                                guard([$body(2,1)],$sym(false))
                                ],nil,[
                                label(5,func(=,2))
                                ],[
                                unify([$sys(5,1)],[$body(2,2)]),
                                unify([$sys(5,2)],[$body(2,4)])
                                ]))
                ],
                [
                        label(2,subclause([
                                guard([$body(2,1)],$sym(true))
                                ],nil,[
                                label(6,func(=,2))
                                ],[
                                unify([$sys(6,1)],[$body(2,2)]),
                                unify([$sys(6,2)],[$body(2,3)])
                                ]))
                ]
                ])
        ],nil,[
        unify([$body(1,1)],[$head(1)]),
        unify([$body(1,2)],[$head(2)]),
        unify([$body(1,3)],[$iv(3)]),
        unify([$body(2,1)],[$iv(3)]),
        unify([$body(2,2)],[$head(3)]),
        unify([$body(2,3)],[$head(1)]),
        unify([$body(2,4)],[$head(2)])
        ])).
```

図 6.12: 展開後の内部形式例

```
max(#(Var0),#(Var1),Var2):-
        compute(>,Var0,Var1,Var3),
        ((Var3 =: false)-->
                Var2 = Var1),
        ((Var3 =: true)-->
                Var2 = Var0).
```

図 6.13: 展開後の fleng−− プログラム

ド条件は、すべてヘッド部に移行する。図 6.13では、ヘッド部に Non-Variable Annotation が付加されているが、これが外部ガードがヘッド部に移行したものである。

6.3.8 トップダウンの展開

前項までで説明したボトムアップの展開のみでは、不十分な場合がある。以下のプログラムは、1 ずつ増える整数列を生成する述語である。

```
gen(N,Max,Ns) :-
    le(N,Max,F),
    $1$gen(F,Ns,N,Max).
$1$gen(true,Ns,N,Max) :-
    Ns = [N|Ns1],
    add(N,1,N1),
    gen(N1,Max,Ns1).
$1$gen(false,Ns,N,Max) :-
    Ns = nil.
```

図 6.14: 整数列ジェネレータ

これを全項までの手法で展開を行なうと、図 6.15のように展開される。ここで、述語呼び出しadd に着目してみよう。このadd の第 1 引数は、Non-Variable Annotation により、値が存在することが保証されている。また、第 2 引数は定数 1 であるから、この述語add は、同期なしで実行できることがわかる。しかし、ボトムアップの展開では、この部分をどのように展開するかは、gen の展開時にではなく、そのサブ述語1gen を展開する時点で決定されるため、この余分な同期を取り除くことができない。

このため、トップダウンの述語展開を組み合わせることで、この種の余分な同期の除去

```
gen(#(Var0),#(Var1),Var2):-
        compute(=<,Var0,Var1,Var3),
        ((Var3 =: false)-->
                Var2 = nil),
        ((Var3 =: true)-->
                add(Var0,1,Var4),        % 余分な同期
                gen(Var4,Var1,Var5),
                Var2 = [Var0|Var5]).
```

図 6.15: 整数列ジェネレータのボトムアップ展開

を行なう。ボトムアップ展開が終了した後に、それぞれのクローズの内部形式に対して、各ボディゴール呼び出し時点での引数の状態をトップダウンに求める。引数の状態がボディゴールに対応するクローズを確定できる場合には、そのボディーゴールを展開することができる。図 6.16にトップダウン展開において使用できる情報の流れと、ボトムアップ展開において使用できる情報の流れを示す。ボトムアップ展開時には、一階層の間の情報のみが使えるので、最適化にも限界がある。トップダウン展開では、より大域的な情報を用いることができる。

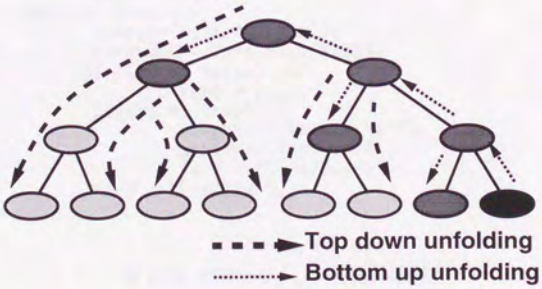


図 6.16: トップダウン展開とボトムアップ展開

本プログラムでは、この過程においても変数ネットワークを用いている。図 6.17に、図 6.15のプログラムに対して、トップダウン展開を行なったものを示す。

```
gen(#(Var0),#(Var1),Var2):-
        compute(=<,Var0,Var1,Var3),
        ((Var3 =: false)-->
                Var2 = nil),
        ((Var3 =: true)-->
                compute(+,Var0,1,Var4),
                gen(Var4,Var1,Var5),
                Var2 = [Var0|Var5]).
```

図 6.17: 整数列ジェネレータのボトムアップ＋トップダウン展開

トップダウン展開では、繰り返して展開を行なうことができる。6.17での gen の呼び出しに着目してみよう。一度目のトップダウン展開で、add が展開されてcompute になったことによって、gen の第一引数である Var4 に値が束縛されていることが保証された。第二引数の Var1 は Non-Variable Annotation によってすでに保証されているので、gen を展開することができるようになる。これを行なうと、図 6.18のプログラムを得る。

```
gen(#(Var0),#(Var1),Var2):-
        compute(=<,Var0,Var1,Var3),
        ((Var3 =: false)-->
                Var2 = nil),
        ((Var3 =: true)-->
                compute(+,Var0,1,Var4),
                compute(=<,Var4,Var1,Var5),
                ((Var5 =: false)-->
                        Var6 = nil),
                ((Var5 =: true)-->
                        compute(+,Var4,1,Var7),
                        gen(Var7,Var1,Var8),
                        Var6 = [Var4|Var8]),
                Var2 = [Var0|Var6]).
```

図 6.18: 複数回のトップダウン展開

この場合には topdown 展開はいくらでも行なうことができる。このような展開を行なうことで、ゴールフレームのセットアップにかかるコストが削減できるが、コード量が増大してしまうため、注意が必要である。

第 7 章

fleng 処理系の評価

本章では、前章までに述べた、粒度制御と負荷分割に関する実機での評価を行なう。

7.1 負荷分散手法の評価

本節では、静的負荷分割と動的負荷分割の効果と、負荷量との相関を調べる。

7.1.1 実験環境

実験は PIE64 のシステムを用いて行なった。

測定条件は以下の通りである。

- 使用 IU 台数: 2 4 8 12 16 24 32 48 64
- 動的負荷分割: ON, OFF
- 静的負荷分割: ON, OFF

使用 IU 台数を変化させることで、相対的にプログラムの負荷量とプロセッサ台数の比率が変化する。すなわち、台数が少ない場合には高負荷に、多い場合には低負荷になる。

動的負荷分散は 2 つのコードを用いて実現している。高負荷時のコードは負荷分散を行なわない。低負荷時のコードは負荷分散を行なう。

7.1.2 対象プログラム

対象としては、エラトステネスの篩を用いて 3000 までの素数を求めるプログラム "primes" を用いた。このプログラムは、それぞれの素数に対する篩を fleng のプロセスとして実現し、このプロセス間に自然数のストリームを通して、素数の数列を得るものである。

このプログラムは、一般に想像されるほどには並列性がない。プロセス自体は素数の数だけ生成されるが、実際に稼働するのは小さい値の素数に対応する篩のプロセスばかりで、大きい値の素数に対応する篩のプロセスはほとんど動かないからである。図 7.1に primes を 3000 まで求めた際の世代ごとの並列度を示す。ある世代のリダクションによって生じるゴールは次の世代でリダクションされると仮定している。これは、1 リダクションにかかる時間がすべて同じで計算機の台数が無限大であることを仮定することに相当する。

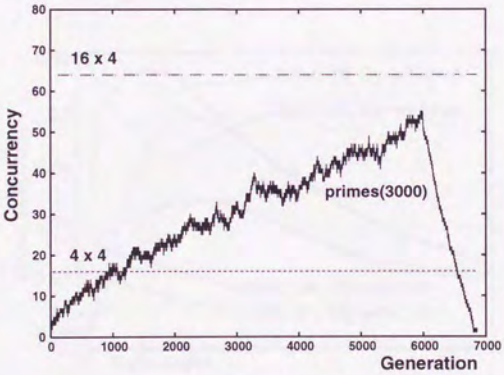


図 7.1: primes の並列度

このグラフからわかるように、primes の並列度は 4 台の IU の 16 のコンテクストを埋めるには十分であるが、16 台の 64 のコンテクストには届かず、64 台の IU に対しては全く不足している。したがって、4 台近辺をを高負荷時、16 台近辺を中負荷時、それ以上を低負荷時と考えることができる。

7.1.3 実験結果

それぞれの場合について以下の項目を測定した。

- UNIRED ストール率
  UNIRED の実行時間に対する、ストール時間の割合。通信のレイテンシを隠蔽できないと生ずる。このため、有効なコンテクスト数が少ないと増加する。
- NIP 実行時間率
  NIP が実行している時間の割合。
- サスペンド回数

図 7.2に UNIRED のストール時間を、示す。図 7.3に NIP の実行時間を示す。

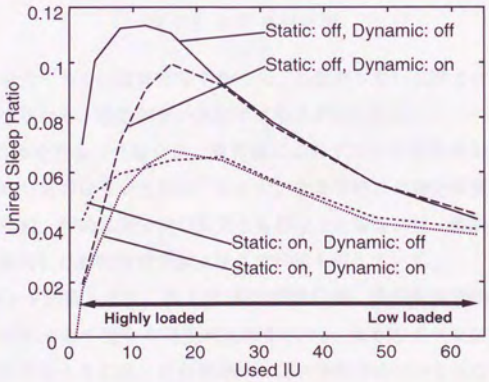


図 7.2: UNIRED ストール時間

これらのグラフは基本的に同じ傾向を示しているのがわかる。一番上のラインが、動的負荷分割、静的負荷分割の双方とも行なわない場合である。その下のラインが、動的負荷分

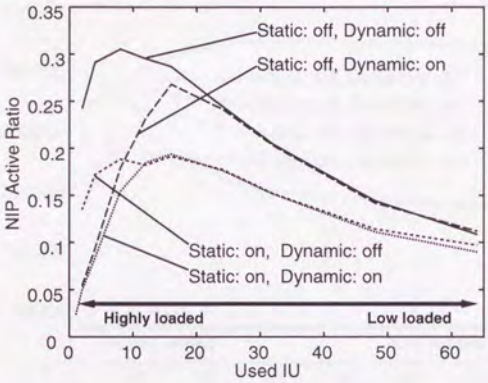


図 7.3: NIP 実行時間

割のみを行なった場合である。高負荷時すなわち、台数が少ない状態では良い成績を示しているが、低負荷時すなわち、台数が多い状態では効果が見られない。

静的負荷分割のみを行なった場合は、負荷値によらずストール時間が減少している。高負荷時には動的負荷分割のほうが効果的であるが、低負荷時には静的負荷分割のほうが有効である。動的負荷分割、静的負荷分割の双方とも行なった場合には、低負荷時には静的負荷分割に準じ、高負荷時には動的負荷分割に準じた特性を示している。

図 7.4にサスペンド回数を示す。図 7.5に動的負荷分割、静的負荷分割の双方とも行なわない場合に対するそれぞれの場合の速度向上率を示す。 双方のグラフに共通するのは、動的 / 静的負荷分割を行なうものは、低負荷時には動的負荷分割のみを行なうものに準じ、高負荷時には静的負荷分割のみを行なうものに準ずる、という点である。

静的負荷分割のみを行なったケースでは、高並列部分で実行速度が行なわないものよりも遅くなっている。この原因は、静的負荷分割のオーバーヘッドによるものであると考えられる。動的負荷分割を行なわない場合は静的負荷分割したコードを、最低負荷の IU に投げようとする。この時、どこにも負荷の軽い IU がないと、結局他の IU には投げずにローカ

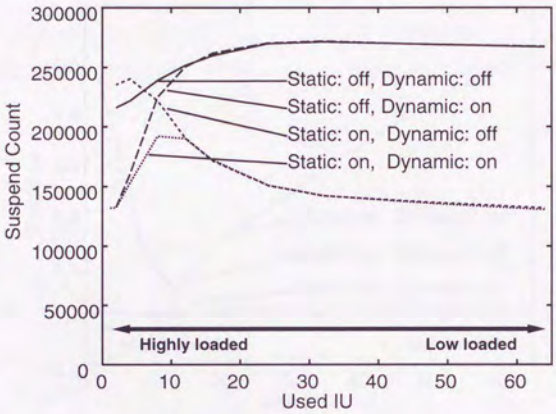


図 7.4: サスペンド回数

ルに実行する。この場合は参照の局所性は確保されるが、この投げようとする試行のコストがオーバーヘッドとなって、速度の低下を引き起こしている。動的負荷分割を併用すると、高負荷時にはこの試行を行なわなくなるので、このオーバーヘッドは解消される。

7.1.4 まとめ

動的負荷分割の効果は、高並列時に顕著である。しかし、低並列時には全く効果がない。これに対して、静的負荷分割は低並列時には有効であるが、高並列時には速度低下を引き起こす。これらのことから、全領域で安定した高性能を得るためには、双方の負荷分割を併用することが必要であると結論できる。

7.2 粒度調整の静的評価

本節では、本研究で述べた粒度調整に関して静的な評価を行なう。粒度調整の前後でコードを構成する命令の分布がどのように変化したかを評価する。

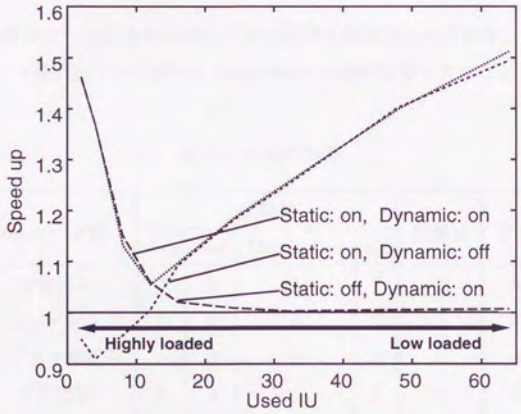


図 7.5: 速度向上率

7.2.1 展開の効果

本項では、展開によって実行されるコード量がどのように変化するかを調べる。

対象プログラム

例として下のプログラムを考えてみよう。

```
foo(A, B, C):- bar(A, B, C).
bar(A, B, C):- baz(A, B, C).
```

このプログラムに対して、展開を行なうと下のプログラムを得る。

```
foo(A, B, C):- baz(A, B, C).
```

これによってコード量がどのように変化したかを見てみよう。展開のみの評価を行なうため、4.1で述べた終端ゴールの最適化は行なわないとする。

結果

表 7.1に展開前のコードと展開後のコードにおける命令の分布を示す。展開前のコードでは foo から bar を呼び出すのに用いる Dispatcher 内部の命令も含める。

表 7.1: 展開の効果

op コード種	展開前				展開後	差分
	foo	bar	Dispatcher	計		
即値代入	1	1	5	7	1	6
デリファレンス	0	0	0	0	0	0
レジスタ間転送	0	0	0	0	0	0
算術演算	0	0	6	6	0	6
分岐	0	0	5	5	0	5
ロード	0	0	8	8	0	8
ストア	5	5	0	10	5	5
その他	3	3	0	6	3	3
計	9	9	24	42	9	33

展開による命令数の減少は、33 命令である。このうち 24 命令は Dispatcher のコードである。展開後のゴール `foo` のリダクションにかかるコストが 9 命令であることを考えると Dispatcher を省略することは大きな意味を持つと考えられる。分類から見ると、ロードとストアの減少が著しい。これは、ゴールフレームへのセーブとロードが省略されることによるものである。

また、命令数のみでは評価できないが、実際には新たに生成されたゴールは MP を経由して戻ってきており、そのレイテンシを考慮すると、さらに差が拡大すると考えられる。

7.2.2 実際のプログラムでの評価

本項では、実際のプログラム中でもっとも多く実行されるループに関して、命令数の変化と命令の構成の変化を調べる。本項では、実際のプログラム中での効果を見るため終端ゴールの最適化を行なう。前項の例と異なりこの例では、調整前も後も Dispatcher ルーチンを用いていない。このような場合でも一定の効果があることを示す。

対象プログラム

対象プログラムとして次節で、評価に用いる N-queens の一部を用いる。N-queens の中でもっとも多く実行される、check というループを構成する述語を例にとり、ループの一順に用いられる命令数を評価する。

下にオリジナルのプログラムを示す。

```
check(P,D,L,[Q|Lp0],Lp,A0,A) :-
    add(Q,D,J),
    eq(J,P,K),
    sub(Q,D,V),
    eq(V,P,M),
    or(K,M,N),
    $2$check(N,A0,A,D,P,L,Lp0,Lp).
$2$check(true,A0,A,D,P,L,Lp0,Lp) :-
    A0 = A.
$2$check(false,A0,A,D,P,L,Lp0,Lp) :-
    add(D,1,D1),
    check(P,D1,L,Lp0,Lp,A0,A).
```

これに対して粒度調整を行なったものを下に示す。

```
check(#(Var0),#(Var1),Var2,[#(Var3)|Var4],Var5,Var6,Var7):-
        compute(-,Var3,Var1,Var8),
        compute(+,Var3,Var1,Var9),
        compute(=,Var8,Var0,Var10),
        compute(=,Var9,Var0,Var11),
        compute(lor,Var11,Var10,Var12),
        ((Var12 =: false)-->
                compute(+,Var1,1,Var13),
                check(Var0,Var13,Var2,Var4,Var5,Var6,Var7)),
```

```
        ((Var12 =: true)-->
                Var6 = Var7).
```

このプログラムは、展開によって数値演算の述語も展開されている。2check の展開による変化のみを計測するため、比較対象のプログラムも数値演算の述語は展開した。下にそのコードを示す。

```
check(# P,# D,L,[# Q|Lp0],Lp,A0,A) :-
    compute(+,Q,D,J),
    compute(=,J,P,K),
    compute(-,Q,D,V),
    compute(=,V,P,M),
    compute(lor,K,M,N),
    $2$check(N,A0,A,D,P,L,Lp0,Lp).
$2$check(true,A0,A,D,P,L,Lp0,Lp) :-
    A0 = A.
$2$check(false,A0,A,D,P,L,Lp0,Lp) :-
    compute(+,D,1,D1),
    check(P,D1,L,Lp0,Lp,A0,A).
```

なお、双方のプログラムに対して終端ゴールの最適化をおこなった。このため両方とも、ゴールフレームに値を書き込むことなく、ループを行なう。オリジナルのプログラムが間接的にループを形成しているのに対して、調整後のプログラムは、直接ループを形成している。

結果

表 7.2にループ内部の命令の分布を示す。

命令数全体では、3 分の 2 程度に減少している。UNIRED は、RISC 的なアーキテクチャであるので、基本的にはすべての命令は 1 クロックで実行される。ブランチのペナルティやメモリアクセスのレイテンシは存在するが、十分コンテクストが供給されている場合にはコンテクストスイッチによって隠蔽されるため、十分な負荷が存在すると仮定すれば、すべての命令が 1 クロックで実行されると考えられる。したがって、この命令数の減少はそのまま実行時間の減少につながる。

表 7.2: ループ内の命令の分類

op コード種	調整前			調整後	差分
	check	2check	計		
即値代入	9	3	12	10	2
デリファレンス	7	2	9	9	0
レジスタ間転送	20	7	27	13	14
算術演算	6	1	7	7	0
分岐	10	2	12	11	1
ロード	1	0	1	1	0
ストア	4	2	6	2	4
その他	1	0	1	0	1
計	58	17	75	53	22

この減少の大部分はレジスタ間転送命令の減少によるものである。これは、間接的なループが直接的なループになったことによる。述語を直接呼び出す場合には、引数をレジスタ r1 以降に並べることになっている。調整前のコードでは間接的にループを行なうため、一回のループにつきレジスタへの値の整列を 2 度行なわなければならない。しかし調整後のコードではこれが 1 度になるため、レジスタ間転送命令が大きく減少している。

7.2.3 まとめ

静的粒度調整の結果、どのようにコードが変化するかを調べた。展開による命令数の省略は、ロード / ストア 命令が主であり、特に Dispatcher の動作が省略されることによって命令数が減少している。

また、終端ゴールを最適化しているため Dispatcher を用いないようなプログラムにおいても、コード量が減少する。この場合のコード量の減少のほとんどは、レジスタ間転送命令の減少によるものである。これは終端ゴールを最適化した場合には、ゴールフレームをレジスタ上に作成するが、展開によってこれが省略されるからである。

7.3 静的粒度調整の評価

本節では、本論文で述べた静的粒度調整の効果について調べる。

7.3.1 実験環境

実験は PIE64 上のシステムを用いて行なった。

測定条件は以下の通りである。

- 使用 IU 台数: 1, 2, 3, 4, 6, 8, 12, 16, 24, 32, 40, 48, 56, 64

7.3.2 対象プログラム

対象プログラムとしては、前述した primes の他に n-queens を用いた。n-queens は基本的に全解探索問題であり、計算木は幅が広く浅いため、非常に高い並列度をもつことで知られている。この点で、primes とは対象的な性質を持つプログラムであるといえる。

測定には、それぞれのプログラムを通常にコンパイルしたものと、展開によって粒度を最大限まで拡張したものを用いた。

7.3.3 実験結果

それぞれの場合について以下の項目を測定した。

- スピードアップ
  実行時間 の 逆数の比
- UNIRED 稼働率
  全時間に対して、UNIRED が有効に実行を行なっている時間。ストールしている時間は除く。
- サスペンド率

サスペンド率は、サスペンドの回数をリダクションの回数で割ったものである。しかし、粒度調整を行なうとリダクションの回数は、展開によって変化してしまう。このため、基数となるリダクションの回数としてオリジナルのコードを用いた場合の回数を用いた。

図 7.6に Primes のスピードアップを示す。全領域で、1.5 から 2 倍の速度向上が見られる。

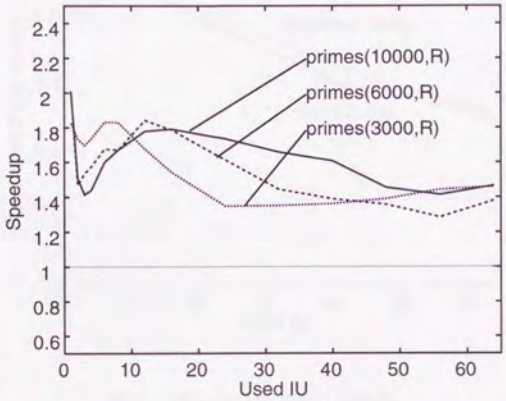


図 7.6: Primes スピードアップ

図 7.7は primes 10000 の場合の UNIRED の稼働率である。使用 IU 台数が増えるに従って、稼働率が低下しているがこれは相対的にプログラムの並列度が減少してくるためである。IU 台数が多い領域で、調整を行なったコードのほうが高い稼働率を示している。

図 7.8に primes 10000 の場合のサスペンド率を示す。調整を行なったコードは、サスペンド率が著しく低下しているのがわかる。これは、粒度調整の静的スケジューリング効果であると考えられる。

図 7.9に N-queen のスピードアップを示す。全領域で、1.8 倍程度の速度向上が見られる。

図 7.10は 11-queen の場合の UNIRED の稼働率である。調整を行なったコードは非常に高い稼働率を示している。

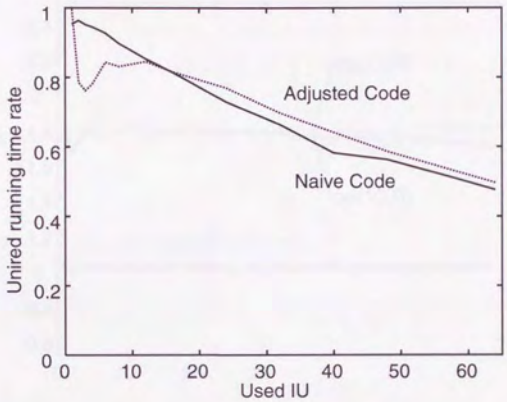


図 7.7: Primes UNIRED 稼働率

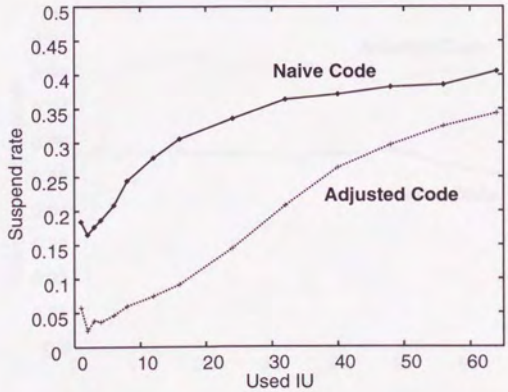


図 7.8: Primes サスペンド率

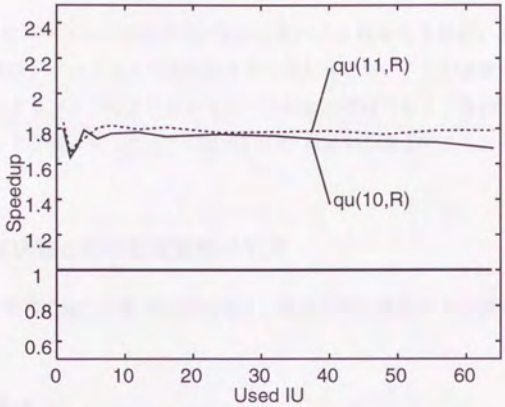


図 7.9: N-queen スピードアップ

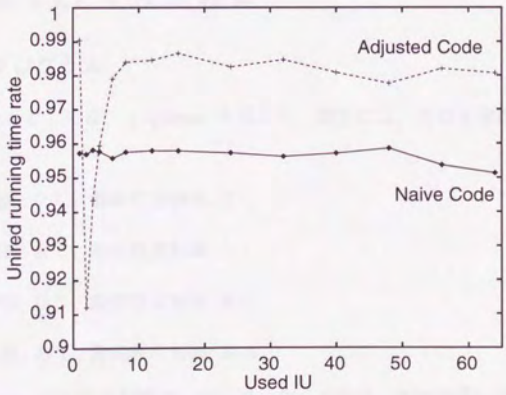


図 7.10: N-queen UNIRED 稼働率

7.3.4 まとめ

調整を行なったコードは、行なわないものに比べ 1.5 倍から 2 倍近い速度の向上を示している。この傾向は、プログラムの並列度などには依存しない。この速度向上の原因は、前節で述べた展開によるコードのクリティカルパスの縮小だけでなく、静的スケジューリングによってサスペンドが減少すること、UNIRED の 稼働率が向上することの影響であると考えられる。

7.4 静的粒度調整と動的粒度調整の効果

本節では、本論文で述べた静的粒度調整と、動的な粒度調整がどの領域で効果的であるかを評価する。

7.4.1 実験環境

実験は PIE64 上のシステムを用いて行なった。

測定条件は以下の通りである。

- 使用 IU 台数: 1, 4, 8, 16, 32, 40, 48, 64

7.4.2 対象プログラム

対象プログラムとしては、n-queens を用いた。測定には、次の 4 種類のプログラムを用いた。

1. 静的粒度制御: なし、動的粒度制御: なし
2. 静的粒度制御: あり、動的粒度制御: なし
3. 静的粒度制御: なし、動的粒度制御: あり
4. 静的粒度制御: あり、動的粒度制御: あり

また、n-queens に与える引数は、8 を用いた。これは、比較的低い並列度のプログラムを用いることで、台数との相対関係で、高負荷状態と低負荷状態をつくり出すためである。十分な測定時間を得るために、8-queens を 10 回繰り返して、測定を行なった。

7.4.3 実験結果

それぞれの場合について以下の項目を測定した。

- スピードアップ
  それぞれの場合について、静的粒度制御、動的粒度制御ともに行なわない場合を 1 とした、速度の向上率を測定した。
- 高負荷モード／低負荷モード／アイドルの時間比率

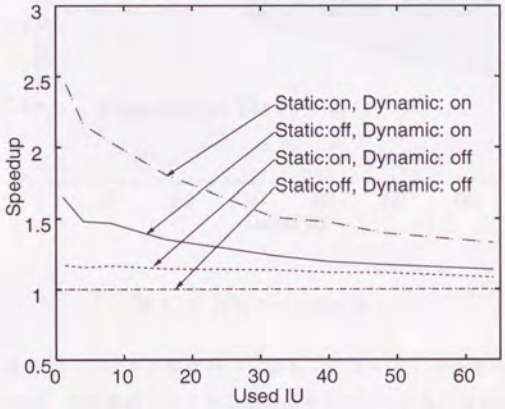


図 7.11: 静的 / 動的粒度調整の効果

図 7.11に静的 / 動的粒度調整を行なった場合の速度向上を示す。図の左側は、プロセッサ台数が少ないので相対的に高負荷状態に、右側は低負荷状態になっている。一番下のグラフが、粒度調整を全く行なわない場合を示している。静的粒度制御のみを行なった場合のグラフが下から 2 番目のグラフである。負荷状態に関わらず、一定の効果をあげている。動的粒度制御のみを行なった場合のグラフが下から 3 番目のグラフである。低負荷状態では高い効果をあげているが、負荷が低くなると効果が減少する。1 番上の両方を行なった場合のグ

ラフにも同様の傾向がみられるが、静的粒度制御の効果によって、低負荷状態でも比較的高い性能を示している。

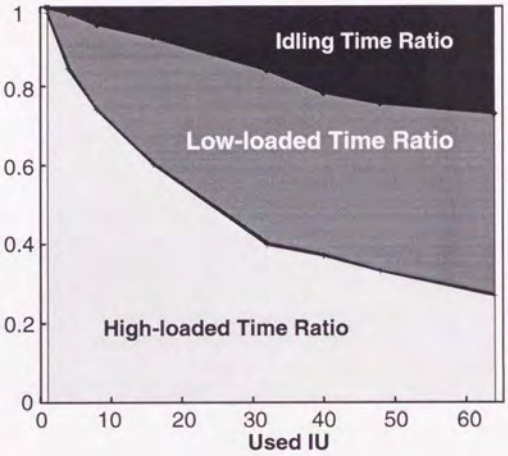


図 7.12: 負荷モードの比率

図 7.12に、低負荷バージョンと高負荷バージョンのコードのそれぞれの実行比率を示す。下から、高負荷時、低負荷時、アイドル時の比率を示している。負荷が減るにつれアイドル時間が増え、それにともなって低負荷時の状態が増えていくのがわかる。8-queens の実行では、最低の並列度の場合 (すなわち最大プロセッサの場合) でも 30% 程度の時間は高負荷モードで実行することができているが、より低い並列度の場合には、高負荷モードでの実行はできなくなり、動的粒度調整は機能しなくなることが予想される。

7.4.4 まとめ

動的粒度制御は、高負荷時には非常に効果的であり、細粒度言語の実行には必須であるといえる。しかし、低負荷時にはあまり効果はない。静的粒度制御は、負荷状態によって効

果が変化しないという特徴を持つ。これら二つの手法の双方を相補的に用いることで、より高速な実行が可能になる。

第 8 章

考察

本章では、本研究で述べた手法に関して検討を加える。まず、粒度制御の手法に関して他の言語における手法、とくに関数型言語の粒度制御の手法との比較を行なう。次に、プログラム変換に関して検討する。最後に、粒度制御に必要になる言語機能について、検討する。

8.1 従来の粒度制御手法との比較

粒度の制御は、並列実行の一つのキーポイントであり、さまざまな言語で試みられている。粒度の制御には大きく分けて二つのアプローチがある。一つは、粗粒度または逐次の言語の実行粒を分割して最適な粒度を得る方法。もう一つは、本研究と同様に細粒度の言語に対して実行粒を合成して最適粒度を得る方法である。ここでは、Committed-Choice 型言語と並んで後者の代表である関数型言語の並列実装に関して、本研究との比較を行なう。

8.1.1 関数型言語

関数型言語はもともと並列でも逐次でもない。しかし、その参照透明性によって並列実行に対する適正があることが知られている。特に、その実行機構がデータフロー計算機と相性が良いことから、データフロー計算機向けの言語として盛んに研究がされている。代表的なデータフロー向け関数型言語としては、Id[45]、pH[46]、SISAL, Valid[41] などがある。

データフロー向け関数型言語は、大別して strict なものと、non-strict (lenient) なものに分けられる。strict な関数型言語は、関数を実行する際に引数の評価の後に実行を行な

うが non-strict な関数型言語は引数の評価と関数本体の評価を並行して行なう。後者は、パイプライン的な実行が可能になるため、前者よりも高い並列度が得られる。SISAL は、strict な関数型言語であり、Id, pH, Valid などは non-strict である。

関数型言語と Committed-Choice 型言語

Committed-Choice 型言語と比較すると関数型言語には以下の利点がある。

- データの依存関係が caller と callee に限定されるので、実行モデルが単純
  Committed-Choice 型言語では、データの依存関係には、束縛はないため実行モデルが複雑になる。
- 意味論がシンプルであるため、プログラムの変換などが容易
  関数型言語の意味論は λ 計算であたえられる。λ 計算ではプログラムの 畳み込み (folding)、 展開 (unfolding) などの操作には制約がない。
- 言語機能 (型、高階関数など) が整備されている
  Committed-Choice 型言語はいまだに、言語機能としては非常に原始的なレベルにとどまっている。これに対して、関数型言語では言語機能や型システムに関する研究が非常に進んでいる。これは、関数型言語の簡潔な意味論に依存する点が大きいと思われる。Committed-Choice 型言語に対しても、関数型言語などを参考にして拡張を行なうべきであると思われる。

逆にデメリットとしては、以下があげられる。

- 複雑なデータ依存関係を持つプログラムの記述が困難
  純粋な関数型言語では、制御の依存関係とデータの依存関係が同じ構造になるため、ツリー状のデータ依存関係しか記述できない。このため、複雑なデータ依存関係を記述するためには、それをツリー状に直す必要がある。この問題を解決するために導入されたのが I-structure[22, 21] である。これは、代入を一度だけにかぎった変数であり、代入以前の読み出しは、代入までブロックされる構造であり、Committed-Choice 型言語の論理変数と良く似たものである。この構造を導入することで、複雑なデータ依存関係をそのまま記述することが可能にはなったが、I-structure へのアクセ

スを明示的に記述しなければならず、プログラマに負担がかかる点で、Committed-Choice 型言語におとる。

- 非決定性のあるプログラムの記述が困難
  関数型言語には非決定性が存在しないため、どうしても非決定性があるプログラムの記述は不可能であるという点である。例えば、ユーザーの入力に対してインタラクティブに動作するプログラムは、ユーザーの入力が非決定的であることから、トリッキーなプログラミングをしなければ、記述するのは難しい。

後者の問題の解決のために、M-structure[45]、Q-structure[80] などの構造記憶を用いる方法が提案されている。M-structure は、複数のプロセスが資源を共有するのに用いる。M-structure は I-structure 同様に、作成時点では空であり、put オペレーションによって fill される。I-structure と異なるのは take オペレーションによって再び空になり、再び put オペレーションを受けられるようになることである。複数の take オペレーションが put オペレーションを待ってブロックしている場合、どの take オペレーションがアクティベイトされるかが非決定的に定まることで、プログラムに非決定性を導入する。Q-structure も I-structure 同様に、作成時点では空である。Q-structure は 複数の writer を許しており、複数の put されたデータは FIFO 状に管理され、writer はブロックしない。take オペレーションが生じると、FIFO 状に管理しているデータを順番に返す。これらの構造は、非常にパワフルであり非決定性のある問題も非常に容易に記述できる。しかし、非決定性を導入していることから容易に予想できるように、これらの構造は、関数型言語の美点である、性質の良い意味論をスポイルすることになる。

I-structure を持つ non-strict 関数型言語と Committed-Choice 型言語を比較してみよう。言語の基本機能としては、この二つはほとんど等価であると考えられるが、いくつかの点で Committed-Choice 型言語が勝っている。

一つは単一代入変数 + ユニフィケーションと I-structure + 代入の差である。

- 単一代入変数は変数同士のユニファイが可能であるが、I-structure ではできない。
- ユニフィケーションは双方向性を持つが、I-structure への代入にはない。

これらの差によって、Committed-Choice 型言語は I-structure つき関数型言語よりも柔軟な記述が可能になっている。例えば、複数のプロセスにたいしてバリア同期をとる Short-circuit と呼ばれる手法は、変数同士のユニファイを利用したものであり、I-structure で実現するのは難しい。反面単一代入変数の持つこれらの特徴により、一般に Committed-Choice 型言語では、変数に値を束縛する際にどのクローズがその値を使用するのかを、静的に特定することが難しい。

ユニフィケーションの双方向性は、ある種のプログラムには非常に有効である。この機能は Committed-Choice 型言語の祖先である論理型言語から引き継いだ機能であり、宣言的なプログラミングを行なう際には有効である。しかし実際のプログラムの多くは双方向性を必要としていないという指摘もある。上田ら [68, 71, 69] はメッセージ指向実装の際に、すべての変数のモードを矛盾なく定めることのできる、すなわち変数が双方向性を持たない FGHC に対象を限定しており、それでほとんどのプログラムを記述でき、できないプログラムに関しても簡単な変換で、記述が可能であるとしている。このような制約を満たすプログラムを well-moded であるという。

双方向性がないと仮定すれば、前者の変数同士の束縛も比較的容易に関数型言語に埋め込むことができる。すなわち、ふたつの I-structure A, B があり、A から B にデータが流れる場合には、A,B をユニファイするという動作は、A に値が来るのを待ってその値を B に流すというスレッドを一つ生成することで実現が可能である。

もう一つの関数型言語と Committed-Choice 型言語の差は非決定性である。Committed-Choice 型言語には、以下の二つの非決定性がある。

1. 変数束縛の非決定性
   変数に対する束縛が複数あった場合、一番早いものが成功する。

2. クローズ選択の非決定性
   複数のクローズが実行可能である場合、そのうちの任意のものが実行される。

対象プログラムを well-moded に限ると、一つの変数に対する代入は一度しか行なわれないことが保証されるので、前者の非決定性はなくなる。後者の非決定性は、M-structure で表現が可能である。

以上述べたように、構造メモリを導入した non-strict な関数型言語の言語モデルは、Committed-Choice 型言語とほぼ等価であり、関数型言語における研究は、Committed-Choice 型言語に応用できる可能性がある。

8.1.2 関数型言語の動的粒度制御

関数型言語に関する粒度制御の研究は大きく分けて、動的な粒度制御と、静的な粒度制御にわかれる。動的な粒度制御の手法には、Lazy Task Creation[42] や、StackThreads[66] がある。これらはいずれも、スレッドのフォーク時のコストを低減するための手法である。

Lazy Task Creation

Lazy Task Creation [42] は並列 LISP のインプリメントのために提案された手法で、並列実行のためのタスク作成、スケジューリングのオーバーヘッドを避けるためのものである。

この手法では、新たなスレッドの作成が指示された場合に、そのスレッドの生成を実際に他のプロセッサが使用可能になり実行ができるようになるまで遅延させ、他のプロセッサがスチールすることが可能になる情報をフレームとして、スタックに積んで実行を続行する。他のプロセッサが空いていず、元のプロセッサがそのフレームまで戻ってくるまで並列実行が不可能だった場合には、もとのプロセッサが実行を行なう。フレームを作成するコストは、サブルーチン呼び出しとほぼ同等なので、タスクを実際に作成するよりはるかに軽くすむ。

StackThreads

StackThreads[66] は、future をベースとした並行オブジェクト言語 ABCL/f [65] の実装に用いられている動的粒度制御手法である。

future を用いた並列言語を素朴に実装すると、future の帰す値を参照してブロックする可能性のある部分では、スレッドを新たに生成して、ブロックに備える必要がある。しかし実際の実行時には、ブロックしない場合の方が多く、この場合にはスレッドを作成したコストは無駄になる。StackThreads[66] は、このコストを避けるために、通常の逐次言語と同様にスタックを用いた実行を行なう。実際にブロックが生じるとそこではじめて、そのコ

ンティニュエーションをスレッドのフレームとして、ヒープに吐き出す。この手法を用いると、ブロックが生じない限りは逐次言語の手続き呼び出しと同等のコストで実行ができる。

もう一つの特徴は、スレッド間の値の受渡しを最適化していることである。通常の実装ではスレッド間の値の受渡しは、メモリを用いるしかない。しかし、スタックを用いて逐次実行しているときには、レジスタで値を返すことが可能である。StackThreads では、値を返す場所を reply box と呼んでいる。reply box には、boxed と unboxed という状態がある。unboxed の状態では、reply box 自身に値が存在し、boxed の状態では、reply box は、値のアドレスを指すポインタとなる。あるスレッドの実行がブロックしないで終った場合、スレッドは返り値を unboxed な reply box に代入しリターンする。この場合、reply box はレジスタ上にとられるので、メモリを介さない通信が行なわれる。ブロックした場合には、スレッドは reply box を boxed に変更し、リターンする。返り値はメモリを介して渡されることになる。

関数型言語の動的粒度制御のまとめ

StackThreads、Lazy Task Creation は双方ともスタック上にスレッドを作成するという点で、PIE64 におけるスタックを用いた実装の手法と共通している。しかしこれらは実装の動機と重点を置くポイントにおいて多少異なる。Lazy Task Creation はプログラムの負荷の変化への対応に重点をおいている。しかし、対象の言語が細粒度ではない future をベースとした言語なので、高負荷時の高効率な実装に関してはあまり留意されていない。一方、StackThreads の対象である ABCL/f は同じ future ベースの言語ではあるが、はるかに細粒度を指向しており、高負荷時の高効率な実装に重点を置いている。しかし、負荷の変動に対する対応は視野に入っていない。PIE64 における動的粒度制御手法はこれらの手法の複合であるともいえる。

8.1.3 関数型言語の静的粒度調整

関数型言語はデータフロー計算機の言語として、多くの研究がなされてきた。しかし、純粋なデータフロー計算機の性能の限界が認識されるにつれ、データフロー計算機はノイマン型計算機との Hybrid 型計算機へと変化していった。この計算機の変化にともなって、関数型言語を Hybrid 型計算機または、通常のノイマン型計算機のの実行モデルに適合するよ

うにコンパイルする手法が開発された。それが、静的粒度調整である。

Hybrid Architecture

データ駆動計算は、いわゆるノイマン型の計算がプログラムカウンタを用いて計算を起動するのに対し、データの到着を契機にして計算を起動する計算法である。

データ駆動計算を行なうには、プログラムをデータフローグラフで表現する。グラフのノードがそれぞれの演算を行なう部分となり、グラフのアークがデータが移動するパスを表現することになる。計算は、必要なデータすべてがノードに到着すると自動的に発火し、結果は出力のアークを伝わって次のノードに送られる。実行の順序がデータの依存関係のみに依存するため、問題に内在する並列性を自然な形で引き出すことができる。

この単純なモデルではループや再帰が実現できないため、データにタグを付加して、実行環境を識別することが行なわれる [20]。複数の入力アークがある場合に、同一のタグを持つ入力データ同士のみをマッチングし、出力データにもそのタグを付加するのである。このようなモデルを実装したデータ駆動型計算機としては、SIGMA-1 [33] がある。

データ駆動計算のモデルは、関数型言語のモデルと非常に近く、関数型言語は容易にデータ駆動計算機に実装することが可能である。すなわち、関数を引数の数だけ入力アークをもつノードと考えれば、引数が揃うと実行が行なわれる関数型言語のモデルに合致する。

純粋なデータ駆動型計算機はトークンの待ち合わせの遅延があるため、演算機の稼働率がどうしても限定される。これに対してノイマン型の計算機では分岐が生じない限りパイプライン上のの命令が実行され続けるため稼働率が高い。この点に着目して、両者の長所を合わせ持つアーキテクチャが考えられた [19, 34]。このモデルでは、計算を局所性のある部分とない部分に分離する。そして、局所性のある部分では、パイプラインを用いた実行を行ない、局所性のない部分は、データ駆動計算機的な待ち合わせ機構による同期を用いる。別の言い方をすれば、局所性のある部分の計算をスタティックにスケジューリングすることにより、その部分のデータを伝搬するアークをレジスタ上にとるのことで、高速化をはかるのである。この局所性を持つ計算部分を強連結ブロック [52] などと呼ぶ。このような混合型のアーキテクチャをもつ計算機として EM-4[83],Monsoon[49],P-RISC[47], RWC-1[82] などがある。

このようなアーキテクチャは関数型言語の素朴な実装では、スレッドの長さが短過ぎる

ため有効に活用できない。長いスレッドを得るために、二つのアプローチがある。ひとつは、スレッドを明示的に扱う言語を用いることである。EM-C[56]、や MPC++[89] などはこれであると考えられる。もう一つのアプローチは、関数型言語を静的に解析することで、本来は短いスレッドを合成して長いスレッドを構成するものである [57, 51, 50]。

グラフ分割による粒度調整

関数型言語の実行において粒度を調整することは、古くから行なわれている。[55] では、関数型言語 SISAL に対して粒度の調整を行なっている。この手法では、まず、ターゲットマシンをいくつかのパラメータで表現して、実行時間をこれらのパラメータと粒度の関数で表してこれを解くことで最適な粒度を求める。さらに、SISAL から変換した IF1 と呼ぶデータフローグラフの上で、変換を行なうことで最適な粒度のプログラムを得ている。まず、データフローグラフを最小単位に分割する。次に、この単位を可能な限り大きくなるまで融合させる。最後に融合した単位を最適な粒度に分割する。

しかし、この手法は対象が SISAL が strict な言語であるため、変換して得られるデータフローグラフの構造が単純であることを前提としているため他の non-strict な関数型言語や Committed-Choice 型言語に用いることはできない。

Merge up と merge down

[57] では、non-strict な関数型言語 Id を後に述べる TAM にコンパイルする手法を述べている。この手法では、プログラムをデータフローグラフに展開した上で、クラスタと呼ぶプログラムブロックに分割する。この 1 クラスタは、TAM の実行単位 である quantum にコンパイルされる。クラスタへの分割は、デッドロックを生じる可能性があるので、慎重に行なわなければならない。分割は以下の基本分割を行なったあとに、merge up、merge down と呼ぶ処理を行なうことで実現される。

基本分割には以下の 3 つがある。このいずれかを行なうことで、基本分割がなされる。

1. **Dataflow partitioning**
   一本の制御の流れをそれぞれのクラスタにする。

2. **Dependence sets partitioning**

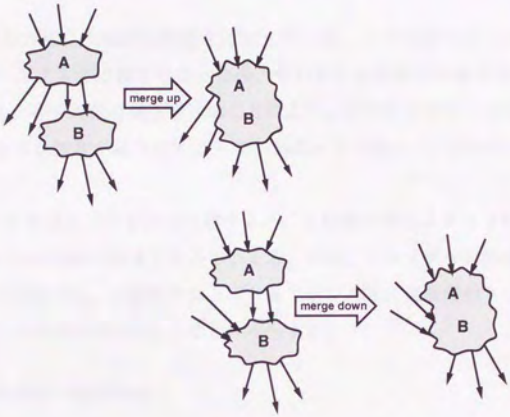


図 8.1: merge up と merge down

依存する入力の集合が同じノードをひとつのクラスタにする。

3. **Dominance sets partitioning**

同じ出力集合に依存されるノードをひとつのクラスタにする。

Merge up、merge down は基本分割で生成されたクラスタをより大きくするための処理である。ブロックを大きくしていくことでクラスタを大きくする。Merge up は、依存関係の下流のブロックが、上流からの入力だけを受けている場合に、下流のブロックを上流のブロックに組み込む操作である。Merge down は、上流のブロックの出力が、一つの下流ブロックにのみ入力されている場合に、上流のブロックを下流のブロックに組み込む操作である。Merge up は Dependence sets partitioning と、merge down は Dominance sets partitioning と、それぞれ同じことを実現するものである。

大域的解析

[51] では、この方法に大域的な解析を付加している。この手法では、caller 側と callee 側のそれぞれで、クラスタ分割を行なった後、それぞれの内部での依存関係を annotation として互いに送る。相手の依存関係を知ることにより、クラスタのマージを進めることができる。このプロセスを何度か繰り返すことで、caller と callee にまたがった大域的な解析が可能になる。

本研究における手法とこの手法を比較すると、本研究のボトムアップの解析と展開は、callee から caller への情報の伝達であるといえる。同様にトップダウン解析が、caller から callee への伝達に相当する。本研究では、ボトムアップの後に複数回のトップダウンを行なうだけであるが、これを交互に行なうことに相当する。

Threaded Abstruct Machine

Threaded Abstruct Machine(TAM)[25] は、関数型言語をストックハードウェアに並列実装するためのモデルである。

TAM の最大の特徴は、スケジューリングを実行単位を Thread と Quantum の 2 段階に分けていることである。Thread は Non-blocking で 分岐なしに走るコードブロックであり、Quantum は、密接に関係する thread と inlet の集合である。inlet はデータの待ち合わせに特化した非常に小さい thread である。Quantum は、内部の thread をスケジュールするためのローカルなスケジュールキューを持っている。

TAM でのスケジューリングは、グローバルには Quantum 単位で行なわれる。ある Quantum がスケジュールされると、その Quantum 内部の動ける thread がなくなるまでローカルにスケジュールを行なう。なくなると、再びグローバルなスケジュールを行ない、他の quantum に制御を移す。

この手法のメリットは、

1. ローカルなスケジュールは、実行フレームを共有するためスイッチングのコストが低くできる。

2. I-structure などのリードをスプリットフェイズで行なえる。リードのリクエストを出した後、同じ quantum の中の他の thread にコンテクストをスイッチするので、その

間にリプライが戻ってくれば、リードのレイテンシを隠蔽することができる。

ことである。

前者は、一種の静的粒度制御である。本研究では静的な粒度制御は non-blocking な領域に限り、block する可能性のある部分の制御は動的な粒度制御に譲っているが、TAM ではすべてを静的に行なうためにこの block する可能性のある部分をふくむ Quantum を導入していると考えられる。PIE64 の処理系の場合、ハードウェアがコンテクストスイッチを直接サポートしているため、後者のメリットは関係ない。

Concert

Concert[37] は並行オブジェクト指向言語を通常のハードウェアで実現するため実行時システムである。並行オブジェクト指向言語のメソッド呼び出しは、対象のオブジェクトがローカルに存在するかどうか、メソッド呼び出しがブロックするかどうか、などのメソッド呼び出しが起きる状況に応じてさまざまな最適化が可能である。Concert ではコンパイラがこのコストの階層を認識しており、Assertion によってメソッド呼び出しが起きる状況をユーザーが特定することによって、静的に定まる部分に関してはコストの安い手法でコンパイルすることで効率のよい実行を行なう。メソッド呼び出しは、ローカルでノンブロックであることがわかっている時には、通常のサブルーチンコールとして実現される。これも一種の静的な粒度制御として考えることができる。

non-blocking で実行できる部分を静的に最適化するという点で、Concert は本研究のプログラム変換による最適化に類似している。しかし、本研究が静的解析による自動的な最適化を行なっているのに対して、Concert では Assertion をプログラマが設定しなければならない。これは、プログラマに負担となる上、Assertion が正しい保証がないという問題を引き起こす。Concert では Assertion が間違っている場合にも、正常な実行を保証するようにリカバーを行なうが、そのコストは非常に高い。

関数型言語の静的粒度制御のまとめ

本研究の粒度調整と、関数型言語のスレッドへのコンパイルは、コンテクストスイッチのコストを削減するために、静的解析をおこなってスレッド長を長くする操作をおこなっているという点で本質的に同じである。本研究ではプログラム変換を用いて実現しているのに

対し、関数型言語ではデータフローグラフのクラスタへの分割によって実現している点が異なる。この相違は主に関数型言語と Committed-Choice 型言語の違いから生じるものである。Committed-Choice 型言語に関しては、グローバルなデータフローグラフを直接求めることが難しいため、本研究ではクローズ単位での変換によって実現した。Committed-Choice 型言語においても well-moded なプログラムのみに対象を限定すれば、データフローグラフが得られると考えられる。データフローグラフに帰着させたプログラムに関してこれらのクラスタリングによる実装を行なうことも考えられる。

本研究では、プログラム変換の手法を展開のみに限っているので、本質的には可能なプログラム変換のうちの一部がサポートできていない (8.2.2を参照)。この点は、本研究の今後の課題の一つである。

8.2 プログラム変換に関する考察

8.2.1 変換条件に関する考察

ここでは、本稿で述べたプログラム変換と [77, 70] で述べられたプログラム変換の比較を行なう。[77, 70] では、束縛と束縛の依存関係を単純な包含関係のみで定義している。しかし、束縛の重要な特徴は変数間の束縛によって、直接の包含関係のない束縛と束縛の間に依存関係が生じるという点である。本稿では、この点に留意して形式化を行なっている。

また、[70] では展開の一種で、サブゴールのガードを上部のクローズのガードに持ちあげる Case Splitting の変換ルールが不必要に厳密である。[70] によれば、 Case Splitting を行なうためにはボディにアクティブなユニフィケーションが存在していないことが必要となっている。本研究では、アクティブなユニフィケーションが存在していてもそのユニフィケーションの結果が外部に露出していなければよいとしている。このような差が生じるのは、[70] では、隠蔽を明示的に取り扱っていないためである。本研究では、隠蔽を明示的に扱うことでこの条件を緩和している。

8.2.2 変換手法に関する考察

本研究では、プログラム変換の手法として展開のみを用いた。しかし、解析の結果シリアライズが可能であることが判明しても、展開のみではシリアライズが実現できないケースもある。

```
foo(A,B,C,D,W,X,Y,Z):-
   add(A, B, B1), add(B1, C, D),
   add(W, X, X1), add(X1, Y, Z).
```

このプログラムは、$D = A + B + C, Z = W + X + Y$ を計算するものである。このクローズの中には、2 組のデータフローが存在するので、展開のみではシリアライズすることはできない。しかし、プログラム変換の手法の一つである連結をゆるせば、以下の様に変形できる

```
foo(A,B,C,D,W,X,Y,Z):-
    bar(A,B,C,D), bar(W,X,Y,Z).
bar(A,B,C,D):-
    add(A, B, B1), add(B1, C, D).
```

これはさらに以下のように展開できる。

```
foo(A,B,C,D,W,X,Y,Z):-
    bar(A,B,C,D), bar(W,X,Y,Z).
bar(#A,#B,#C,D):-
    compute(+, A, B, B1), compute(+ B1, C, D).
```

このように連結を用いればさらにスレッド長を伸ばすことが可能になる。これは今後の課題の一つである。

8.2.3 確定的実行に関する考察

Prolog に代表されるバックトラックを用いて実行制御を行なう論理型言語では、不必要なバックトラックが実行効率を低下させる最大の原因となる。プログラムが確定的であれば、バックトラックを行なわずに実行することができ、バックトラックで取り消される計算をはぶくだけでなく、バックトラック自身のコストを省略することができる。また、完全に確定的に動作させることはできなくても、失敗する計算をより早い段階で失敗させることによって、無駄な計算をより少なくすることができる。

[26] では、Prolog を対象に、各クローズが成功する条件をボトムアップに求め、その条件をクローズの先頭に記述することで、より早い段階でのバックトラックを実現している。もし各クローズの成功条件が排他的であれば、確定的に実行することができ、バックトラックを省くことができる。同時に、トップダウンに述語を実行する環境を解析し、この環境を加味することでより多くの排他性を引き出している。

[39] では、大規模並列計算機を対象にした制約論理型言語 PARCS に対して、同様の手法を試みている。PARCS は、OR 並列実行を行なうが、この際ゴールの環境を完全にコピーする。この作業は非常に重いため、失敗を予想して確定的な実行をおこなうことは非常に重要である。PARCS では、ヘッド部に存在する変数のみを参照する、ユニフィケーションに関して、ゴールの環境を複製する前にチェックを行なう。いわば、Committed-Choice 型言語のガードのように扱うわけである。これによって、ゴール環境の複製前に失敗することができる。

Committed-Choice 型言語では、確定的な実行はこれらの論理型言語ほどには重要ではない。なぜなら、Committed-Choice 型言語はバックトラックも OR 並列実行も行なわないため、非確定であるために余分に動作する可能性があるのは、ヘッド部のパターンマッチのみであるからである。しかし、Committed-Choice 型言語においてもパターンマッチの対象になるクローズを減少させることができれば、この部分を高速化することができる。

本研究で述べた展開は、トップダウンな情報伝搬による確定的な実行ととらえることもできる。例えば下のプログラムを見てみよう。

```
foo(A,B):- bar(A,C), baz(C,B).
bar(x,C):- C = y.
bar(p,C):- C = q.
baz(y,B):- B = z.
baz(q,B):- B = r.
```

このプログラムの実行には、bar が 2 通り、baz が 2 通りで 4 通りの可能性がある。しかし実際には、引数の値によって 2 通りの実行以外は生じない。
展開を行なうと下のプログラムを得る。

```
foo(x,Var0):- Var1 = y, Var0 = z.
foo(p,Var0):- Var1 = q, Var0 = r.
```

実際に生じる 2 通りに対応したプログラムを出力しているのがわかる。このようにプログラム展開は、確定性を増す実行方式としてとらえることもできる。

8.3 言語機能に関する考察

8.3.1 module

プログラム変換によって、プログラムの効率化をはかる際に問題になるのは、プログラムの部分に対して性質を保存しなければならないことである。性質を保存する単位を大きくすればするほど、また、期待する性質をより緩く定義できればできるほど、内部での最適化の自由度が増大する。

例えば、次の例を見てみよう。

```
test([A1|Next], BN):-
     test1(A1, B1),
     BN = [B1|BNext],
     test2(Next, BNext).
test1(a, B):- B = b.
test2([A2], BN):-
     test1(A2, B2),
     BN = [B2].
```

このプログラムを下のように変形することは、一般にはできない。

```
test([A1,A2], BN):-
     BN = [B1, B2],
     test1(A1, B1), test2(A2, B2).
```

これは、第 2 引数に出力されるリストの第 1 要素に、第 1 引数への入力の第 2 要素が依存していると、デッドロックが生じる可能性があるからである。実際には、このような呼び出し方をすることはなくても、それが保証できない限り、このような変形は行なえない。つまり、あるルーチンの持つべき性質を知るには、そのルーチンを呼び出すプログラムの部分を限定しないかぎり不可能である。

このように、呼び出される部分を限定するための機能として、module 機能がある。module は、本来はプログラムの名前空間を分割するために導入されるものであるが、一部の言語では module が 外部に対して提供するルーチンを定義する機能がある。この機能によって、外部に提供しないルーチンに関しては、呼び出しを行なう場所が、その module の内部に限定されるので、最適化が可能になる。このような module 機能は Common Lisp[27] などで実現されている。

しかし、従来の言語で実現されている module 機能では上述の最適化を支援するものとしては十分とはいえない。なぜなら、その module の呼び出し口を特定するだけでは、最適化ができない場合も多いからである。この解決のために、module の宣言に入出力モードの定義を加えることも考えられる。しかし入出力モードだけでは不十分で、最適化が限定される。上記の例で考えてみよう。仮にtest のモードが構造体の内部まですっかりわかっていたとしても、同様にデッドロックの可能性が存在するため最適化はできない。構造体の内部に関してまでモードの宣言を行なえば、解決できる場合もあるが、そのような宣言を行なうことは、プログラマへの負荷が大きく、現実的ではない。

これは、従来のモード宣言には依存関係の記述が全く抜けているからである。本研究で述べた、強入力 / 強出力を持つモードシステムでは、強出力が強入力に依存していることを示すことが可能で、非常に限定された範囲ではあるが依存関係を表現できる。より一般的な手法で、依存関係の宣言を行なうことができれば、最適化できる範囲も広がる。

このための宣言の手法を考察してみる。モジュールの宣言には、依存の宣言よりもむしろ、依存していないことの宣言が望ましいと考えられる。最適化で要求されるのは、依存関係ではなく、依存していないことの保証であるからである。例えば下のような宣言を行なえれば、最適化が可能になる。

```
:- public test([mode(+),independent(2)], [mode(-)])
```

この宣言は、第 1 引数が入力でかつ第 2 引数に依存していない引数に対してのみ動作を保証する、述語 test を宣言している。このような宣言があれば、上述のような変換を安全に行なうことができる。

このような宣言を行なった場合、宣言と実際に使う部分の整合性の保証が問題になる。静的に検証することも可能ではあるかも知れないが、計算量を考えると、あまり現実的ではない。この種の情報は、プログラマには自明である場合が多いので、プログラマの責任において保証してもらうので十分であると考えられる。

8.3.2 データ並列実行構造

前述の問題のもう一つの解決策として、データ並列実行を行なう実行構造を導入する方法も考えられる。前述の例は、二つのデータに同じ処理を加えることのみを目的としてい

る。しかし、適切なデータ表現構造と実行構造がないため、リストで表現しリストを手繰るかたちで実行を行なっている。このことが、データ間の依存関係を不明確にし、最適化に不必要な制約を与えていると考えられる。これに対して明示的なデータ並列実行の構造を導入すると、並列実行対象のデータ間の依存関係はないことが保証されるため、逆に逐次に実行することが可能になる。

関数型言語 Id[45] には、array とよぶデータ構造とその上の map 関数による並列実行構造を持っている。これは、高階関数を用いた構造なので、fleng に導入することは容易ではないが、このようなデータ構造と並列実行構造を導入することも、今後の検討課題である。

第 9 章

結論

9.1 本論文のまとめ

本論文では、高並列計算機上での高効率な言語システムの実装手法の確立を目指し、高並列推論エンジン PIE64 上に Committed-Choice 型言語 fleng の処理系を実装を通して、高並列計算機上の高性能な処理系の構築法に関して議論した。

細粒度高並列言語を高効率に実装するポイントは負荷分散とスケジューリングおよび粒度の制御である。これらの最適化が困難なのは、これらの最適化が本質的に負荷の不均衡をもたらす可能性を持っているためである。このため最適化と負荷の不均衡のトレードオフをとらなければならない。しかし、負荷の不均衡が与える影響は高負荷時と低負荷時で異なるため、単一のコードでは良好なトレードオフの点を求めることはできない。本研究では、PIE64 の負荷分散機能、負荷検出機能、管理プロセッサによるサポートを前提として、特定の実行状態に対応して最適化した複数のコードを静的に用意し、動的にこれらを切替えるシステムを提案した。複数コードを用いることによって、このトレードオフのポイントをそれぞれ低負荷時、高負荷時に最適化することを可能にしている。

負荷分散に関してはコンパイラ、ランタイムカーネル、ハードウェアの 3 段階の制御を提案した。コンパイル時にターゲットの負荷に応じた静的負荷分割を行ない、そのコードを実行時負荷に応じて切替えることで並列性とメモリ参照の局所性を両立させることに成功した。

静的スケジュールに関しては、低負荷時には静的負荷分割を行なうと同時に実行順序を指定することで、高負荷時には、展開による静的粒度調整によってスレッドを構成すること

によってそれぞれ実現した。

静的粒度粒度調整のためにプログラム変換による調整手法と、その調整の前提となる意味論を示した。ボトムアップに構築した意味論に基づいて、ボトムアップの展開を行なうだけでなく、そのあとにトップダウンの展開を行なうことによって、大域的な最適化を行なっている。また、スタックを用いた動的粒度制御を組み合わせることで高い効果を得られることを示した。

また、静的粒度調整をより有効に行なうための言語機構の支援について検討した。

本論文では fleng と PIE64 に関して議論したが、粒度制御、静的負荷分割、スケジューリングの問題は、細粒度高並列言語一般に共通して見られる問題点である。動的な管理に関しては PIE64 に依存した部分があるため、計算機によっては実装が難しいと思われるが、静的な解析及び最適化に関しては他の細粒度高並列言語にも適用が可能であると思われる。

9.2 将来の課題

将来の課題としては、以下のことがあげられる。

- 静的スケジュールを反映させる手法の検討

  現在の処理系では、複数のゴールに関してのぞましい実行順序が静的に解析できても、それを確実に実現する実装手法がない。ローカルに実行する場合には、ローカル IU のスレッドが埋まっているかどうかで、ゴールが FIFO 的に実行されるか、LIFO 的に実行されるかが変化する。低負荷時には FIFO を、高負荷時には LIFO を前提としてコンパイルしてはいるが、十分ではない。また、他 IU へ複数のゴールを送る際には、実行順序は相手 IU の状態に依存するため制御できない。例えば、下の例で、b と c をローカル で、d と e を他の同一の IU で実行するとしよう。

  ```
  a :- b, c, d, e.
  ```

  d と e を、この順番で送りつけた場合、相手の IU にゴールが十分にない場合には、この順番に実行される。しかし、ゴールが十分にあるとスタックを用いてスケジュールされるため、逆順で実行されることになる。このようにリモート実行の順番を十分に

制御する手法がないことが、サスペンド率を低下させることができない一つの原因となっている。

スケジュールの方法として、世代別ゴールプールを用いる方法が考えられる [95, 11]。これは、各 IU のゴールプールに世代を導入し、各世代のゴール実行が終了するまで、次の世代のゴールを実行しないという手法である。このような機構があれば、d とe を一つずらした世代のプールに入れることで、実行順の制御が可能である。この手法は、実行時のオーバーヘッドなどの点で課題も多いが、ひとつの解決策となりうると考えられる。

もう一つの手法としては、プログラム変換によって IU に送るゴールの集合をまとめた新たなクローズを作ってしまうことも考えられる。上の例でいえばd とe をまとめた新たな述語を作るのである。

```
a :- b, c, x.
x :- d, e.
```

新たな述語x に対応するゴールを送ると、リモートではこれを実行して新たにd とe のゴールを生成する。この手法はリダクション回数を増加させてしまう欠点を持つが、サスペンドの回数を低減できる。

- メモリ使用量の予測とそれに基づくスケジュール法の検討

  並列プログラムでは、スケジュールによってメモリ使用量が大きく変化し得る。例えば、生産者プロセスと消費者プロセスをストリームでつなぐ形でプログラミングした場合、生産者プロセス のみが先行してスケジュールされると、本来消費者プロセスによって消費されて解放されるべきデータがなかなか消費されずにヒープを圧迫してしまう。

  これに対応する手法としては、ゴールから派生するすべてのゴールのメモリ使用量を静的に推定し、メモリの使用状況とこの情報を用いて動的にスケジューリングを行なうことが考えられる。この際に問題になるのが、スケジュールの実現法である。プライオリティを用いることも考えられるが、今後の課題である。

- 言語機能の見直し

  fleng の言語セットは機能よりもむしろ意味的なエレガントさを重視して設計されている。その結果、fleng の言語機能はコンパクトで十分な記述力を持ってはいるが、他の言語、特に Id などの関数型言語と比較した場合、非力の感は否めない。今後、Committed-Choice 型言語が研究対象の言語ではなく、実用言語となるにはこの点の改良が必須であると考えられる。

  また従来存在する言語機能に関しても、記述時の必要性と実装に与える影響を比較し、取り去るべきものは取り去る必要がある。例えば、ユニフィケーションなどは、熟慮を要する機能である。Saraswat らの提案する Concurrent Constraint Programming [53, 54, 36] は、Committed-Choice 型言語の枠組を一般に拡張した枠組であると考えられるが、これにはアクティブなユニフィケーションは採用されていない。

  また、前章で述べたように、より効率的に静的粒度制御を実現するためには、言語機能のサポートが必要である。この観点からの言語機能の拡充も必要であろう。

以上をまとめると、今後のこの分野の研究の方向としては、計算機アーキテクチャを含めた実装手法をより洗練するとともに、その実装手法を念頭においた、静的な解析を中心にしたコンパイル手法を言語セットの変更、拡張を含めて確立することが重要である。これらの研究の後に、はじめて並列計算機システムが実用に耐えるものになると思われる。

謝辞

本研究を進めるにあたって、指導教官である田中英彦教授には、多くの御指導をいただきました。また、研究生活における多くのわがままも寛大に受け止めていただき、研究に専念することができました。ここに深く感謝の意を表します。

学位請求論文予備審査におきましては、井上博充教授、武市正人教授、相田仁助教授、松岡聡講師の各先生方に貴重なアドバイスを頂きました。特に松岡講師には、他研究室の学生であるにも関わらず、論文発表に関してもさまざまな御指導を頂きました。ありがとうございました。

SIGIE (Special Interest Group on Inference Engine) グループのリーダである小池汎平講師には、研究の細部にわたって多くの御指導をいただきました。PIE64 は小池講師の発案によるものです。本研究のベースとなった最初のコンパイラも小池講師の手によるものです。

本研究は、並列推論エンジン PIE64 の存在なくしては考えられませんでした。PIE64 の計画と研究、開発に参加されたすべての先輩 / 後輩方に感謝したいと思います。

日高康雄氏は、PIE64 の IU ボードの設計と、PIE64 全体の作成 / デバッグをして下さいました。日高氏がいなければ、PIE64 が稼働する日はけして来なかったと確信しております。その生活すべてを研究に費やす態度には学ぶ点が多々ありました。また、研究に関してのみならず日常的なことがらに関しても、数々の議論におつき合い頂きました。

舘村純一氏には、fleng のデバッガ HyperDebu を開発して頂きました。HyperDebu のおかげで、fleng でさまざまなプログラムの開発が可能になりました。また、研究室での生活を楽しくさせて頂きました。

高橋栄一氏には、研究環境を整備していただき、研究に関することから研究室生活に関することまで、数々の知識を教えていただきました。

吉田実氏には、並列言語の仕様、実装に関してさまざまな議論におつき合い頂きました。また、ネットワークから全般にいたるまでの計算機管理をしていただきました。

清水剛氏は、PIE64 の NIP を開発されました。また、研究者としてあるべき研究に対する姿勢を見せて頂きました。その粘り強い研究姿勢とパワーに敬意を表します。

島田健太郎氏は、PIE64 のメイン CPU である UNIRED を開発されました。また、fleng の最初の処理系も島田氏の手によるものです。コンパイラの設計思想に関しても数々の御示唆をいただきました。

村上聡氏、荒木拓也氏には研究をすすめる上で、コンパイラの改良などに関して多大な協力をして頂きました。ふたりの協力なくしては本研究の評価は不可能でした。

富士通の研究生であった笠原道治氏にはコンパイラの 最初の UNIRED 対応版を作成して頂きました。富士通の研究生であった前田一氏にはコンパイラの LUNA 88k 版を作成して頂きました。元卒論生の白石展久君には、コンパイラのクローズ・インデキシングの実装を行なって頂きました。

佐藤充氏には、データフロー計算機 /Hybrid 計算機に関して、いろいろ教えて頂きました。

白木長武氏、山本敬氏、鍋谷研一氏、小林健一氏、などの皆さんのおかげで、楽しい研究生活を送ることができました。

斎藤禎助手には、研究環境の整備をして頂きました。また、ここにいちいち名前を挙げることはしませんが、この他の田中英彦研究室の方々にも研究生活において大変お世話になりました。また、卒論生の皆さんにも研究生活を楽しくして頂きました。ありがとうございました。

情報工学専攻事務の楠本氏、中山氏にはいろいろ御迷惑をおかけしました。ありがとうございました。

最後に研究室以外の多くの友人、そして家族に感謝します。

発表文献

[1] Hidemoto Nakada, Takuya Araki, Hanpei Koike, and Hidehiko Tanaka. A Fleng Compiler for PIE64. In *Proceeding of PACT '94*, pp. 257–266. Elsevire Science Publishers B.V.(North-Holland), Aug. 1994.

[2] Hidemoto Nakada and Hidehiko Tanaka. Fleng on PIE64 and Its Programming Environment. In *PARALLEL LANGUAGE AND COMPILER RESEARCH IN JAPAN*, to appear.

[3] Takuya Araki, Yasuo Hidaka, Hidemoto Nakada, Hanpei Koike, and Hidehiko Tanaka. System Integration of the Parallel Inference Engine PIE64. In *Proceeding of FGCS'94 Workshop on Parallel Logic Programming*, pp. 64–76, Dec. 1994.

[4] Satoshi Murakami, Hidemoto Nakada, Yasuo Hidaka, Hanpei Koike, and Hidehiko Tanaka. Load distribution system of PIE64. In *Proceeding of FGCS'94 Workshop on Parallel Logic Programming*, pp. 77–90, Dec. 1994.

[5] 中田秀基, 田中英彦. Committed choice 言語へのオブジェクトの導入. 情報処理学会第 45 回全国大会講演論文集, pp. 5– 29–30, Oct. 1992.

[6] 中田秀基, 田中英彦. Immutable object に対する継承の導入による mutable object の継承の実現. In *WOOC '93*, 1993.

[7] 中田秀基, 小池汎平, 田中英彦. オブジェクト間の関係に基づいた記述モデル distributed object connection. 情報処理学会 第 43 回全国大会講演論文集, pp. 5– 231–232, Oct. 1991.

[8] 中田秀基, 小池汎平, 田中英彦. Distributed object connection: オブジェクト間の関係に基づいた記述モデル. 情報処理学会研究報告 プログラミング－言語・基礎・実践－, Vol. 91, No. 60, pp. 27–36, luly. 1992.

[9] 中田秀基, 小池汎平, 田中英彦. オブジェクト間の関係に基づいた記述モデル distributed object connection. In *WOOC '92*, 1992.

[10] 中田秀基, 小池汎平, 田中英彦. オブジェクト間の関係に基づいた記述モデル distributed object connection. オブジェクト指向コンピューティング I, pp. 361–373, 1993.

[11] 中田秀基, 小池汎平, 田中英彦. データフロー解析に基づく committed choice 型言語のスケジューリング. 情報処理学会 第 47 回全国大会講演論文集, pp. 5– 53–54, September 1993.

[12] 中田秀基, 小池汎平, 田中英彦. fleng の動的粒度制御のための静的解析手法. 情報処理学会研究報告 プログラミング－言語・基礎・実践－, Vol. 94, No. 18, pp. 1–8, july. 1994.

[13] 中田秀基, 小池汎平, 田中英彦. Pie64 における committed-choice 型言語 fleng の動的粒度制御のためのコンパイル手法. 情報処理学会 第 49 回全国大会講演論文集, pp. 5–3–4, September 1994.

[14] 中田秀基, 小池汎平, 吉田実, 舘村純一, 白木長武, 田中英彦. 高並列推論エンジン pie64 研究経過報告 – ソフトウェア –. 情報処理学会 第 46 回全国大会講演論文集, pp. 6–93–94, March 1993.

[15] 中田秀基, 田中英彦. 高並列言語 fleng における型システムの導入. 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, , Oct. 1992.

[16] 荒木拓也, 中田秀基, 小池汎平, 田中英彦. ゴールの融合による committed-choice 型言語 fleng の最適化. 情報処理学会 第 49 回全国大会講演論文集, pp. 5– 5–6, September 1994.

[17] 荒木拓也, 中田秀基, 小池汎平, 田中英彦. データフロー解析に基づく committed choice 型言語 fleng の静的負荷分割. 情報処理学会 第 48 回全国大会講演論文集, pp. 5– 71–72, March 1994.

[18] 村上聡, 荒木拓也, 中田秀基, 小池汎平, 田中英彦. Committed-choice 型言語 fleng における静的負荷分割の pie64 上での実装および評価. 情報処理学会 第 49 回全国大会講演論文集, pp. 5– 7–8, September 1994.

参考文献

[19] Arvind and Robert A. Iannucci. Two fundamental issues in multi processing. In *Proc. of DFVLR Conference 1987 on Parallel Processing in Science and Engineering*, June 1987.

[20] Arvind and Rishiyur S. Nikhil. Executing a program on the mit tagged-token dataflow architecture. *IEEE Trans. Computers*, Vol. C-39, No. 3, pp. 300–308, 1990.

[21] Arvind, Rishiyur S. Nikhil, and K. Pingali. I-structure: Data structures for parallel computing. *ACM Trans. on Programming Languages and Systems*, Vol. 11, No. 4, 1989.

[22] Arvind and R.E. Thomas. I-structures: An efficient data type for functional languages. TM 178, MIT, 1980.

[23] Takashi Chikayama. Operating system PIMOS and kernel language KL1. In *Proc. of International Conference on FIFTH GENERATION COMPUTER SYSTEMS 1992*, pp. 73–88, 1992.

[24] K. Clark and S. Gregory. PARLOG: Parallel programming in logic. *ACM Transactions on Programming Language and Systems*, Vol. 8, No. 1, 1986.

[25] David E. Culler, Sah Anurag, Klaus Erik Schauser, Thorsten von Eicken, and John Wawrzynek. Fine-grain parallelism with minimal hardware support:a compiler-controlled threaded abstract machine. In *4th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 164–175, April 1991.

[26] Steven Dawson, C. R. Ramarkrishnan, and L. V. Ramarkrishnan. Extracting determinacy in logic programs. In *Proc. of Int. Conf. on Logic Programming*, 1993.

[27] Guy L. Steele Jr. et al. *COMMON LISP*. 共立出版, 1986.

[28] T. Chikayama et al. A portable and efficient implementation of kl1. In *PLILP'94*, 1994.

[29] E.Tick. Practical Static Mode Analyses of Concurrent Logic Languages. In *Proceeding of PACT '94*, pp. 205–214, Aug. 1994.

[30] Masakazu Furuichi, Kazuo Taki, and Nobuyuki Ichiyoshi. A multi-level load balancing scheme for or-parallel exhaustive search programs on the multi-PSI. In *Proc. of PPOPP*, pp. 50–59, 1990.

[31] Yasuo Hidaka, Hanpei Koike, and Hidehiko Tanaka. Multiple threads in cyclic register windows. In *Proc. of International Symposium on COMPUTER ARCHITECTURE*, pp. 131–142, 1993.

[32] Yasuo Hidaka, Hanpei Koike, Jun'ichi Tatemura, and Hidehiko Tanaka. A static load partitioning method based on execution profile for committed choice languages. In *Proc. of ILPS*, 1991.

[33] K. Hiraki, T. Shimada, and K. Nishida. A hardware design of the SIGMA-1 – a data flow computer for scientific computations. In *Int. Conf. Parallel Processing*, pp. 524–531, 1984.

[34] Robert A. Iannucci. Toward a dataflow / von neumann hybrid architecture. In *Proc. of 15th ISCA*, pp. 131–140, 1988.

[35] M. S. Miller K. Kahn, E. Dean and D. G. Bobrow. *"Vulcan: Logical Concurrent Objects"*, Vol. 2, pp. 274 – 303. MIT Press, 1987.

[36] Kenneth M. Kahn and Vijay A. Saraswat. Actors as a sprecial case of concurrent constaraint programming. In *ECOOP/OOPSLA '90*, 1990.

[37] Vijay Karamcheti and Andrew Chien. Concert — efficient runtime support for concurrent object oriented programming languages on stock hardware. In *Supercomputing'93*, 1993.

[38] Yasunori Kimura and Takashi Chikayama. An abstract KL1 machine and its instructionset. In *Proc. of Sympo on Logic Programming '87*, pp. 468–477, 1987.

[39] Kazuhiro Konno, Masaaki Nagatsuka, Naoki Kobayashi, Satoshi Matsuoka, and Akinori Yonezawa. PARCS: an mpp-oriented clp language. In *Proc. of PASCO'94*, 1994.

[40] Kouichi Kumon and Keiji Hirata. A new transformation based on process-message duality for concurrent logic languages. In *Proc. of Int. Conf. on Logic Programming*, 1994.

[41] Shigeru Kusakabe, Eiichi Takahashi, Rin'ichiro Taniguchi, and Makoto Amamiya. Dataflow-based lenient implementation of a functional language, valid, on conventional multi-processors. In *Proceeding of PACT '94*, pp. 279–288, Aug. 1994.

[42] Eric Mohr, David A. Kranz, and Robert H. Halstead. Jr. Lazy task creation: A technique for increasing the granularity of parallel programming. In *ACM Conference on Lisp and Functional Programming*, 1990.

[43] T. Nakagawa, N. Ido, M. Asaie, and M. Sugie. Hardware implementation of dynamic load balancing in the parallel inference machine. In *Proc. of Int. Conf. on Fifth Generation Computer Systems*, pp. 723–730, 1992.

[44] H. Nakashima, K. Nakajima, S. Kondo, Y. Takeda, Y.Inamura, S. Onishi, and K. Masuda. Architecture and implementation of PIM/m. In *Proc. of Int. Conf. on Fifth Generation Computer Systems*, pp. 425–435, 1992.

[45] Rishiyur S. Nikhil. ID language reference manual ver.90.1. CSG Memo 284-2, MIT, 1991.

[46] Rishiyur S. Nikhil. Some comparisons of functional id and haskell and new syntax, for pH. Sep. 1993.

[47] Rishiyur S. Nikhil and Arvind. Can dataflow subsume von neumann computing. In *Proc. of 16th ISCA*, pp. 262–272, 1989.

[48] Martin Nilsson and Hidehiko Tanaka. Massively parallel implementation of flat GHC on the connection machine. In *Proc. of Int. Conf. on Fifth Generation Computer Systems*, pp. 1031–1040, 1988.

[49] Gregory M. Papadopoulos and David E. Culler. Monsoon: An explicit token-store architecture. In *The 17th Annual ISCA Conference Proceedings*, pp. 82–91, 1990.

[50] Lucas Roh, Walid A. Najjar, and A. P. Wim Bohm. Generation and quantitative evaluation of dataflow clusters. In *Proc. of the Conference on Functional Programming Languages and Computer Architecture*, 1993.

[51] Kenneth R.Traub, David E. Culler, and Klaus Erik Schauser. Global analysis for partitioning non-strict programs into sequential threads. In *Proc. of Lisp and Functional Programming'92*, 1992.

[52] S. Sakai, Y. Yamaguchi, K. Hiraki, Y. Kodama, and T. Yuba. An architecture of a data flow single chip processor. In *The 16th Annual ISCA Conference Proceedings*, pp. 46–53, 1989.

[53] Vijay A. Saraswat and Martin Rinard. Concurrent constaraint programming. In *POPL '90*, 1990.

[54] Vijay A. Saraswat, Martin Rinard, and Prakash Panangaden. Semantic foundations concurrent constaraint programming. In *POPL '91*, 1991.

[55] Vivek Sarker and John Hennessy. Partitioning parallel programs for macro-dataflow. In *Proc. of Lisp and Functional Programming'86*, 1986.

[56] Mitsuhisa Sato, Yuetsu Kodama, Shuichi Sakai, and Yoshinori Yamaguchi. EM-C: programming with explicit parallelism and locality for the EM-4 multiprocessor. In *Proceeding of PACT '94*, pp. 3–14, Aug. 1994.

[57] Klaus Erik Schauser, David E. Culler, and Thorsten von Eicken. Compiler-controlled multithreading for lenient parallel languages. In *Proc. of the Conference on Functional Programming Languages and Computer Architecture*, 1991.

[58] Ehud Shapiro, editor. *Concurrent Prolog*. The MIT Press, 1987.

[59] Kentaro Shimada, Yasuo Hidaka, Jun'ichi Tatemura, Hanpei Koike, and Hidehiko Tanaka. Studies on the parallel inference engine PIE64. *Annual Report of the Engineering Research Institute Faculty of Engineering, University of Tokyo*, Vol. 51, pp. 73–78, 1992.

[60] Kentaro Shimada, Hanpei Koike, and Hidehiko Tanaka. UNIRED II: The high performance inference processor for the parallel inference machine PIE64. In *Proc. of Int. Conf. of Fifth Generation Computer Systems*, pp. 715–722, 1992.

[61] T. Shimizu, T. Horie, and H. Ishihata. Low-latency message communication support for the ap1000. In *The 19th Annual ISCA Conference Proceedings*, pp. 288–297, May 1992.

[62] Takeshi Shimizu, Hanpei Koike, and Hidehiko Tanaka. Details of the network interface processor for PIE64 (in japanese). SIG Reports on Computer Architecture, Information Processing Society of Japan, 87-5, 1991.

[63] Eiichi Takahashi, Hanpei Koike, and Hidehiko Tanaka. A study of a high bandwidth and low latency interconnection network in PIE64. In *Proc. of IEEE Pacific Rim Conf. on Communications, Computers and Signal Processing*, pp. 5–8, 1991.

[64] Kazuo Taki. Parallel inference machine PIM. In *Proc. of International Conference on FIFTH GENERATION COMPUTER SYSTEMS 1992*, pp. 50–72, 1992.

[65] Kenjiro Taura, Satoshi Matsuoka, and Akinori Yonezawa. A future-based polymorphic typed concurrent object-oriented language – its design and implementation –. In *Proc. of the DIMACS workshop on Specification of Parallel Algorithms*, 1994.

[66] Kenjiro Taura, Satoshi Matsuoka, and Akinori Yonezawa. Stackthreads: An abstract machine for scheduling fine-grain threads on stock cpus. In *Proc. of JSPP'94*, pp. 25–32, 1994.

[67] K. Ueda. Guarded horn clauses. TR 103, ICOT, 1985.

[68] K. Ueda and M. Morita. A new implementation technique for flat GHC. In *Proc. of Seventh Int. Conf. on Logic Programming*, pp. 3–17. Mit Press, 1990.

[69] Kazunori Ueda. Moded flat GHC and its message-oriented implementation technique. *New Generation Computing*, November 1994.

[70] Kazunori Ueda and Masao Morita. Transformation rules for GHC programs. In *Proc. of Int. Conf. on Fifth Generation Computer Systems*, 1988.

[71] Kazunori Ueda and Masao Morita. Message-oriented parallel implementation of moded flat GHC. In *Proc. of Int. Conf. on Fifth Generation Computer Systems*, 1992.

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

[73] D.H.D Warren. An abstract prolog instruction set. Technical Note 309, SRI International, 1983.

[74] 吉田実. オブジェクト指向機能を持つコミッティッドチョイス型言 語 Fleng++ とそのプログラミングシステム. PhD thesis, 東京大学, 1992.

[75] 久門耕一, 平田圭二. FGHC プログラムにおけるメッセージとプロセスの交換 – 実行スレッドに着目した効率改善手法. ソフトウェア科学会第 9 回全国大会, pp. A2–3, 1992.

[76] 古市昌一. PIMOS 負荷バランスユーティリティマニュアル. TR 796, ICOT, 1992.

[77] 古川康一. GHC でのプログラム変換. プログラム変換, pp. 135 – 150. 共立出版, 1987.

[78] 久門耕一, 平田圭二. FGHC の双対変換に基づく continuation と migration の記 述. TR 846, ICOT, 1993.

[79] 今井明. VPIM 処理方式解説書. TM 1044, ICOT, 1991.

[80] 佐藤三久, 児玉祐悦, 坂井修一, 山口喜教. 高並列計算機 em-4 における分散データ構造を用いたマルチスレッドプログラミング. 情報処理学会 ARC 92, 92.

[81] 佐藤令子, 佐藤裕幸. 疎結合型マルチプロセッサ上の拡散型負荷分散の一方式. TR 794, ICOT, 1992.

[82] 坂井修一, 石川裕. Rwc における超並列システムの研究開発. 情報処理, Vol. 34, No. 12, December 1993.

[83] 児玉祐悦, 坂井修一, 山口喜教. 高並列計算機 EM-4 とその並列性能評価. 信学論, Vol. J75-D-I, No. 8, pp. 607–614, August 1992.

[84] 寿崎かすみ, 近山隆. KL1 上の並列オブジェクト指向言語 AYA(綾) の設計 . In *WOOC '91*, 1991.

[85] 寿崎かすみ, 近山隆. Kl1 上の並列プロセス指向言語 aya. TR 652, ICOT, 1991.

[86] 小中裕喜. Committed-choice 型言語におけるタイプ / モード推論システ ムの研究. Master's thesis, 東京大学, 1989.

[87] 小林健一. 分散メモリ型並列計算機における論理型言語 fleng の高性能 実装. Master's thesis, 東京大学, 1994.

[88] 森田正雄, 市吉伸行, 関田大吾, 近山隆. KLIC の共有メモリ並列実装方式. 情報処理学会 第 49 回全国大会講演論文集, pp. 5– 13–14, September 1994.

[89] 石川裕, 小中裕喜, 前田宗則, 友清孝志, 堀敦史. 超並列プログラミング言語 MPC++ の概要. 情報処理学会研究報告 PRG 13, 1993.

[90] 増井俊之. Perl 書法. アスキー出版局, 1993.

[91] 中村宏明, 田中英彦. 並列論理型言語 fleng に基づいたオブジェクト指向言語 fleng++. In *WOOC'89*, 1989.

[92] 仲瀬明彦, 六沢一昭, 近山隆. KLIC 処理系の分散メモリ処理系におけるメッセージ通信の 実現と評価. 情報処理学会 第 49 回全国大会講演論文集, pp. 5– 17–18, September 1994.

[93] 日高康雄. 高並列記号処理における並列実行制御方式の研究 (to appear). PhD thesis, 東京大学, 1995.

[94] 日高康雄, 小池汎平, 舘村純一, 田中英彦. 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 の静的負荷分散. 情報処理学会論文誌, Vol. 32, No. 7, pp. 807–816, 1991.

[95] 日高康雄, 小池汎平, 田中英彦. 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.

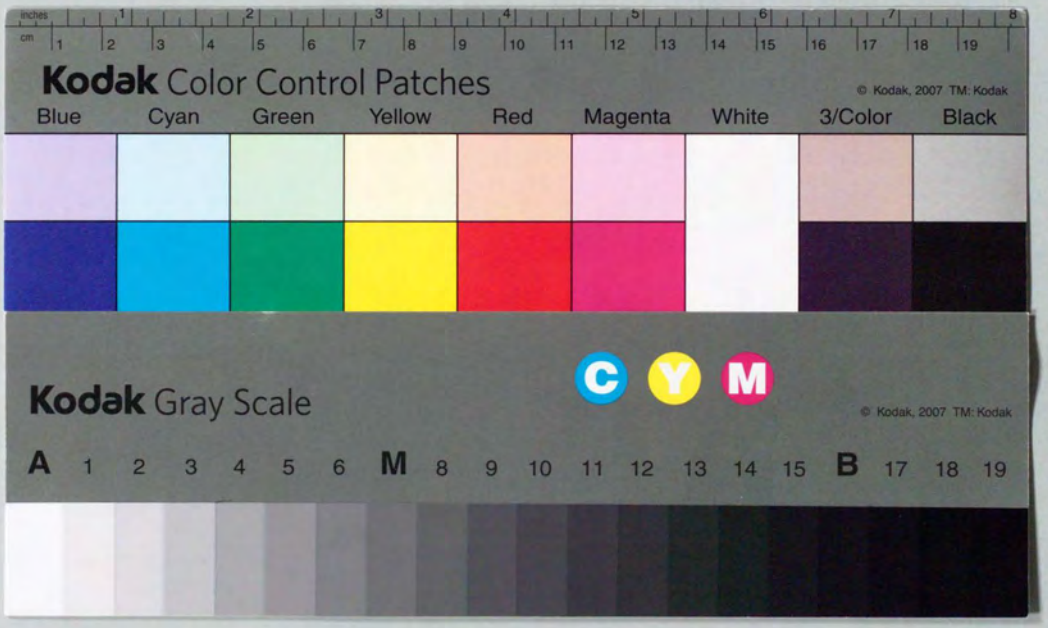
[96] 畑澤宏善, 新井進. 並列推論マシン PIM/p におけるクラスタ内自動負荷分散方 式の評価. In *Proc. of JSPP'93*, pp. 355–362, 1993.

[97] 平田圭二, 久門耕一. FGHC プログラムにおけるプロセスとメッセージの双対性. ソフトウェア科学会第 9 回全国大会, pp. A2–4, 1992.

[98] 木村康則. 並列論理型言語 KL1 の処理系に関する研究. PhD thesis, 東京大学, 1993.

[99] 近山隆, 藤瀬哲朗, 関田大吾, 中村豪一. KLIC 処理系核の評価. 情報処理学会 第 49 回全国大会講演論文集, pp. 5– 11–12, September 1994.

[100] 六沢一昭, 仲瀬明彦, 近山隆. KLIC 処理系の分散メモリ実装方式. 情報処理学会 第 49 回全国大会講演論文集, pp. 5– 15–16, September 1994.


inches
cm
Kodak Color Control Patches
© Kodak, 2007 TM: Kodak
Blue
Cyan
Green
Yellow
Red
Magenta
White
3/Color
Black
Kodak Gray Scale
C
Y
M
© Kodak, 2007 TM: Kodak
A 1 2 3 4 5 6 M 8 9 10 11 12 13 14 15 B 17 18 19