同期ビットを利用する細粒度並列コードの生成

稲垣達氏　　松本尚　　平木敬

東京大学大学院理学系研究科情報科学専攻

マルチプロセッサシステムのメモリ上の同期ビットを利用してイテレーションレベルの並列処理および命令レベルの細粒度並列処理を行なうためのコード生成手法について述べる。本稿のターゲットシステムはメモリアクセス単位の通信と同期を統一的に実現したMISC(A Mechanism for Intergrated Synchronization and Communication)システムを備える。DOACROSSループの高速化では、イテレーション間およびイテレーション内に存在する粒度の細かい並列性を効率良く利用することが重要である。我々はタスクグラフの持つ並列性を元にして、イテレーションレベルの並列処理と命令レベルの並列処理から適用すべき手法を選択し、それぞれについてメモリ上の同期ビットを使用するコードを生成を行なう。

Code Generation for Fine-Grained Parallel Processing utilizing Memory Based Synchronization Bits

Tatsushi INAGAKI　　Takashi MATSUMOTO　　Kei HIRAKI

Department of Information Science, Faculty of Science, the University of Tokyo

We describe a code generation method for efficient fine-grained parallel processing at iteration level and instruction level to utilize memory based synchronization bits. Our target system provides the MISC(A Mechanism for Intergrated Synchronization and Communication) mechanism to realize atomic synchronization and communication at a memory access level. To speed-up DOACROSS loops, it is important to exploit fine-grained parallelisms within an iteration and among iterations. We use two different level of parallel processing according to parallelisms lying in a task graph of a DOACROSS loop.

1 はじめに

プログラム中の逐次的なループ (DOACROSS ループ) は、そのままでは並列に実行できないためプログラム全体の高速化のボトルネックとなりやすい。しかし、逐次的なループのイテレーション間、イテレーション内の部分的な計算の間に粒度の細かい並列性がある場合、これらの並列性を抽出し、細粒度並列処理を行なうことによって、逐次的なループの実行時間を削減し、全体の実行時間に対する影響を緩和できる可能性がある。本稿では、逐次的なループを高速化するコード生成手法について述べる。

ここで扱う高速化手法は、従来逐次的なループの高速化に用いられてきたイテレーションレベルの DOACROSS 実行と、VLIW／スーパースカラ計算機で用いられる命令レベルの並列処理である。我々は、効率的な細粒度の同期・通信を支援する機構を備えたマルチプロセッサシステムをターゲットとして、ループから得られるタスクグラフの持つ並列性の観点に従ってこれらの並列化手法を適用する。

2 逐次化ループ

本節では DOACROSS ループ (逐次化ループ)[1] について述べる。逐次化ループとはループのイテレーション間で依存関係を持ち並列実行できないループのことである。依存を持つイテレーション間の距離を依存距離と呼ぶ。依存距離が大きいほど逐次化ループの各イテレーションは部分的に並列実行が可能になり、ループは DOALL ループに近付く。以下では簡単のため依存距離を 1 として話を進める。

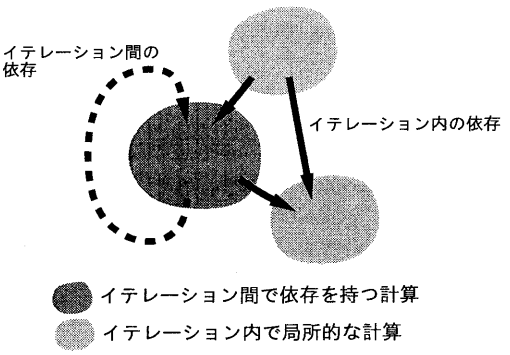

図 1: 逐次化ループの構造

逐次化ループのループボディの計算を依存関係によるタスクグラフで表すと、その構造は図 1のようになる。イテレーション内の計算は前後のイテレーションに対してイテレーション間の依存を持つ部分と、そこで使われる値を計算する部分および前者二つの値を使って計算される部分に分けられる。ここでイテレーション間の依存を持つ部分は生成されたタスクグラフの強連結成分を形成する。逐次化ループでタスクグラフの強連結成分以外の部分は各イテレーションで局所的に計算できる。

逐次化ループで、各イテレーションで局所的に行なえる計算の量が、イテレーション間のデータ依存を持つ計算に比べて、同等かそれ以上である場合は逐次化ループに対してイテレーションレベルの並列処理 (DOACROSS 実行) が適用できる。DOACROSS 実行ではプロセッサに対してイテレーションをサイクリックに割り当て、イテレーション間の局所的な計算のオーバラップによって逐次化ループの高速化を図る。この場合、イテレーション内の並列性も同時に利用できるソフトウェアパイプライニング [5] をマルチプロセッサシステムに適用することも考えられる。しかし、不規則なプロセッサ間の通信パターンを生成する可能性のある静的タスクスケジューリングよりも、規則的な DOACROSS 実行の方が資源競合によるオーバーヘッドを吸収する能力に優れるため、イテレーション間の並列性だけを利用する方が現実的である。

上では依存距離が 1 であるとしたが、イテレーションは一般に複数の先行するイテレーションに依存する可能性がある。DOACROSS ループとしての性質を決めるのはそれらのイテレーション間の依存距離の中の最小値である。この場合最小依存距離よりも長い依存距離を持つ値を用いる計算は、それだけ通信の遅延に関する制約が緩く、タスクの制約に関してはイテレーションに局所的な計算に近い。従って、最小依存距離 1 を持つ計算に対して、依存距離が長い計算が十分多くあれば DOACROSS 実行による高速化が見込める。

イテレーションに局所的な計算が、前後のイテレーションに依存する計算に比べて非常に少ない場合は DOACROSS 実行による高速化が望めない。このような場合でも、前後のイテレーションに依存する計算の中に並列性があれば、これらの計算に静的タスクスケジューリングを用いた細粒度並列処理を施し、ループの実行を高速化できる可能性がある。この場合はプロセッサ間において短い時間間隔で不規則な通信を行なうため、メモリアクセスレベルの通信と同期を効率的に実現する支援ハードウェアが重要である。

3 MISC

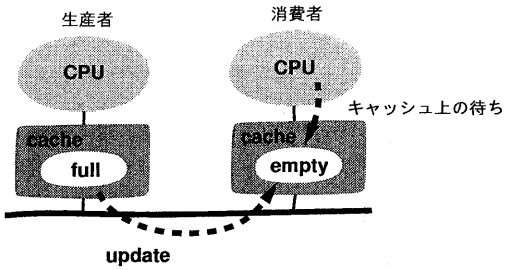

図 2: MISC を用いたキャッシュ上での同期

マルチプロセッサシステムにおいて効率的な並列処理を行なうためには、ハードウェアによってできるだけ低コストのプロセッサ間の同期および通信手段を提供することが不可欠である。本節では我々がターゲットとするマルチプロセッサシステムにおいて細粒度並列処理を支援する同期および通信機構について説明する。

MISC(A Mechanism for Integrated Synchronization and Communication)[7] はコヒーレントキャッシュを用いてメモリアクセスレベルの通信と同期を統合した機構である。MISC を用いることによって、ワード単位の生産者／消費者関係を付加的な命令オーバーヘッドを伴わずに実現することができる。

MISC はキャッシュメモリ上にワード単位の同期ビットを持つ。読みだしは同期ビットが full 状態のときは値を返し、empty 状態のときは full 状態になるまでブロックされる。同期ビットの物理的な状態と論理的な full/empty 状態の対応を同期極性 (synchronization polarity) と呼ぶ。MISC では同期極性は動的に解釈されるため、一回の書き込みを次の同期のための初期化として利用できる。実際には同期極性は論理的にはページエントリの属性、物理的にはシステムで使用されていない物理アドレスの上位ビットによって表現されるので、ある MISC を使用するアドレス領域に対して二つの異なるベースアドレスを用意して、同期極性を使い分ける。

同期ビットもデータと同じくキャッシュプロトコルによって管理されるので、update 系のプロトコルを用いることによって通常のメモリアクセスと同じコストで同期を伴ったメモリアクセスを実現することが可能である。すなわち、共有されているデータに関して同期待ちに入るプロセッサは、ローカルキャッシュ上にあるコピーをタグとして待ちを行ない、プロセッサ外にアクセスが出ることはない。MISC を用いた書き込みが行なわれると、コヒーレンス管理の update メッセージによってコピーの同期ビットが full 状態に更新され、同期が実現される (図 2)。

4 細粒度並列処理

本節では逐次化ループを高速化するための細粒度並列処理手法について説明する。ある逐次化ループの実行時間の理想的な下限は、タスクグラフの強連結成分の最大パス長 × イテレーション数で与えられ、我々の目標は逐次化ループの実行時間をできるだけこの下限に近付けることである。2節で述べたように、逐次化ループに利用できる並列性がある場合、並列性の種類によってイテレーションレベルの並列処理または命令レベルの並列処理手法が適用できる。以下ではそれぞれの並列化における MISC を用いたコード生成手法を述べる。

4.1 イテレーションレベル並列処理

MISC が保証できる先行関係はメモリアクセスのフロー依存なので、逐次ループをイテレーション間でフロー依存だけをもつような形にできれば、依存を持つメモリアクセスに対して MISC を適用して効率的な DOACROSS 実行が行なえる。

配列アクセスがイテレーション間依存を持つ場合は、依存を与えるその配列へのループ中の書き込みが互いに干渉しないことが、MISC が適用できる条件である。

スカラ変数によるフロー依存を持つ DOACROSS ループの場合は、変数に対応するアドレスで MISC による同期を行なうとイテレーション間で逆依存と出力依存が生じるため並列実行ができない。この場合簡単なコード生成法としては、スカラ展開 (scalar expansion) を行なってこのスカラ変数をループと同じ次元数を持つ一次元配列に展開する方法がある。しかし、スカラ展開を行なうと必要な一時領域が大きくなりやすいので、この場合はスカラの局所化 (scalar privatization) を行なう。各プロセッサで元のスカラ変数のコピーを用意し、それらのアドレスを使用して MISC による同期を行なう。各プロセッサ内では、1 イテレーション毎に局所的なスカラ変数に対して同期極性を反転した書き込みを行なう。この場合、同期極性が反転した書き込みどうしが出力依存を生じるが、各プロセッサが書き込むアドレスは独立であり、この出力依存はプロセッサ内の逐次実行によって保証される。また、プロセッサ間には逆依存が存在するが、スカラ変数の依存距離は 1 なので、逆依存は推移的に保証される。

スカラ変数の場合の初期値、また配列の場合ではループの外で生成される値を読むイテレーションは、

MISCを用いない読みだしを行なうようにするか、予め該当する領域にMISCを用いた書き込みを行なっておくかしておかなければいけない。このような情報を管理するために、コンパイラはLWT(Last Write Tree)[8]で表されるデータフロー情報に基づいてコード生成を行なう。LWTは読みだしのイテレーションがフロー依存する書き込みのイテレーションを与える関数である。LWTの判定部分をループボディで直接実行すればコードは短くて済むが、すべてのイテレーションで境界条件の判定を行なうことになり効率が悪い。したがってループ外の値を読み出すイテレーションを分離して、ループの前処理および後処理としてコードを生成するのがよい。

MISCはアドレスをタグとして同期を行なうので、アドレスが動的に生成され、静的に依存距離がわからないループや、条件実行によって依存が生じるかどうかを静的に決定できないループにもMISCによる同期が適用できる。

4.2 命令レベル並列処理

イテレーション分割による計算のオーバーラップが期待できないDOACROSSループにおいても、ループボディの計算の中に並列性があれば、静的タスクスケジューリングを用いた細粒度並列処理によって、1イテレーションの実行を高速化できる可能性がある。

4.2.1 タスクスケジューリング

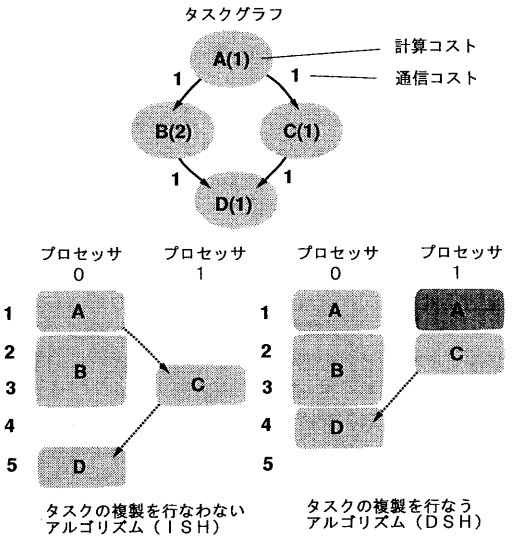

図 3: DSHの例

まず始めに、ループボディ内の計算を命令レベルのタスクグラフとして表す。タスクグラフはイテレーション内の依存を表すエッジとイテレーション間の依存を表すエッジを含むが、イテレーション内のエッジに注目するとタスクグラフはDAG(directed acyclic graph)として扱える。これにリストスケジューリングアルゴリズムを適用してマルチプロセッサ上のスケジューリングを得る。我々が採用するアルゴリズムはBoontee Kruatrachue[4]によって提案されたISH(Insertion Scheduling Heuristic)とDSH(Duplication Scheduling Heuristic)である。ISHは通信によって生じた遅延スロットにタスクを挿入するアルゴリズムで、DSHは先行タスクをプロセッサ内の空きスロットに複製することによって通信および通信遅延を削減するアルゴリズムである(図3)。ISHは演算パイプラインの競合によって生じる空きスロットを考慮することによって、パイプライン化された要素プロセッサのための命令スケジューリングにも適用することができる。DSHは中間コード最適化における共通部分式の削除の逆変換に相当しており、スケジューリング時に与えられたプロセッサシステムで利用できるだけの並列度を抽出するアルゴリズムであるといえる。

ソースコード中の基本ブロック内の計算は直接DAGに対応させることができる。DAGが大きいほど多くの並列性が見込まれる。従って、条件実行がある場合には、VLIW計算機のスケジューリングと同様にトレーススケジューリング[2]を適用する。トレースとは、頻繁に実行される制御パス上の基本ブロックを融合したものである。トレースにおいて基本ブロックを越えるコード移動を行なった場合はプログラムの意味を変えないために補償コードが必要になる。

実際のマルチプロセッサシステムでの命令の実行タイミングを見積もるためには、プロセッサだけでなく通信のタイミングも考慮しなければならない。特に単一の共有バスのように、競合を起こし易い資源が関係する場合は通信のスケジューリングが全体の実行時間に与える影響が大きい。しかし、greedyなリストスケジューリングでは、キャッシュメモリを備えたシステムでバスアービタによって本来動的に管理される資源である共有バスが使用されるタイミングを、完全に静的に見積もることはコストが高い。現実的な解決策は共有バスをプロセッサとして扱うことで近似的に使用タイミングを推定することである。

4.3 同期コードの生成

マルチプロセッサシステム上で命令レベルの細粒度並列処理を行なう場合、プロセッサ間にマップされ

たフロー依存を持つエッジはプロセッサ間通信と同期によって実現される。ハードウェアによる共有メモリを備えたシステムでは、細粒度のデータ通信は共有メモリアクセスによって実現される。したがって、プロセッサ間でフロー依存を持つエッジについては、MISCによって簡単に個々の通信と同期を行なうことができる。

このようにしてMISCを用いる場合、1イテレーション内を並列化しているため、DOACROSSループを実行する間に同じメモリ領域で何度もMISCによる同期を行なう必要がある。よってイテレーション間でMISCを用いるアドレスの再利用をしなければならない。アドレスの再利用を行なうためには、ある領域が書き込みによって次回の使用のために初期化されることを保証する必要があるので、基本ブロックを管理の単位とするのが簡単である。しかし、極性管理のオーバーヘッドをなるべく減らすためには、管理の単位をトレースにまで広げた方がよい。なぜなら、同期極性の管理時にはMISCによる書き込みに対する逆依存(以前の極性によるすべての同期が終了している)を保証しなければならないので、極性を切替える際に何らかの他の全体の同期手段(バリア同期など)を用いなければならないからである。従って一つのイテレーションを覆うトレースを実行する間はなるべくこの全体の同期を必要としないことが望ましい。トレース単位で同期極性の管理を行なった場合、条件実行によって頻度の高いトレースの実行が一部行なわれなかったときには、頻度の低い方のトレースに、元のトレースで実行されなかったMISCによるダミーの書き込み(=初期化)のコードを挿入する必要がある(図4)。

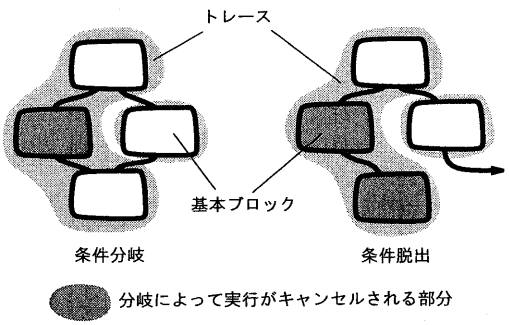

図4: MISCによるダミー書き込みの挿入

タスクグラフがフロー依存しか含まない場合はMISCのみを用いて並列化されたコードを実行できるが、メモリアクセスなどの副作用による逆依存や出力依存は直接MISCで保証することができない。このような場合はパイプライン的にバリア同期を処理することができるElastic Barrier同期機構を用いた細粒度並列処理[3]を併用するのがよい。Elastic Barrierは先行タスク側をブロックしないバリア同期機構であり、MISCと同様に細粒度の命令レベルの並列処理を実現できる。

4.3.1 プロセッサ内の命令スケジューリング

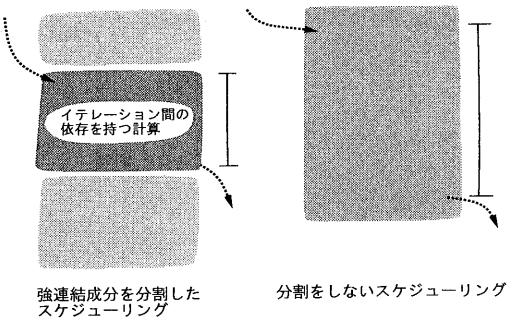

図5: イテレーション内の計算のスケジューリング

命令レベル並列処理では、コードスケジューリングアルゴリズムによってプロセッサ内の命令実行のタイミングが、プロセッサ間の実行タイミングと同時に決定されていた。イテレーションレベルの並列実行においても、プロセッサ間の計算のオーバーラップが大きくなるようにプロセッサ内の命令のコードスケジューリングを行なうことが重要である。

イテレーションレベル並列処理によってDOACROSS実行を行なう場合、イテレーション間でデータ依存を持つタスクグラフの強連結成分と、イテレーションに局所的な計算に対しそれぞれ別々に演算パイプラインの資源制約を考慮したタスクスケジューリングを適用して、なるべく強連結成分のスケジューリング長が短くなるようなコードを生成することができる(図5)。この場合、1イテレーションのスケジューリング長自体は長くなりうるが、全体の実行時間は並列化によって短縮できる。

プロセッサ内に強連結成分をスケジューリングするときは、コンパイラはイテレーション内の依存だけを考慮してタスクスケジューリングを行なうが、この際にも強連結成分同士の通信・計算をなるべくパイプライン化するようなコード生成を行なうことができる。タスクにbottom levelによって優先順位を付ける場合、バックエッジを持ち次のイテレーションで使用される値を通信するストアノードは通常は全てレベル0

```
      DO 20 k= 1,n
           DI= Y(k)-G(k)/(XX(k)+DK)
           DN= dw
           IF(DI.NE.0.0) DN=MAX(S,MIN(Z(k)/DI,T))
         X(k)= ((W(k)+V(k)*DN)* XX(k)+U(k))
     &          /(VX(k)+V(k)*DN)
       XX(k+1)= (X(k)- XX(k))*DN+ XX(k)
   20 CONTINUE
```

図 6: リバモアループカーネル #20 のコード

に揃えられる。このストアノード間に、対応するロードノード間のレベルを順序を反映する順序を付けるように、制御依存を導入することによって、イテレーション間で強連結成分同士がオーバーラップして実行できる可能性がある。

5 実験

MISC を用いた細粒度並列処理コードの効果を見るために、共有メモリマルチプロセッサシミュレータ [6] 上で簡単なループカーネルを用いて実験を行なった。本シミュレータはスヌープキャッシュを備えたバス結合集中共有メモリ計算機におけるクロックレベルの命令シミュレーションを行なうものである。シミュレータの演算とメモリアクセスに関するコストは以下のように設定した。

- プロセッサの演算パイプラインの構成は MIPS R4000 に準拠。
- バスアクセスを伴わないメモリアクセスは 3 クロックかかる。共有バス上での転送はさらに 3 クロックかかるので、共有バスを使用した通信のコストは 9 クロックである。
- MISC を用いる場合のレイテンシは通常の通信と同じ。
- スヌープキャッシュのプロトコルは update を用いる。

以下のグラフでは、横軸はシステムのプロセッサ台数、縦軸はプログラムの実行時間を表す。各実験では並列化されたプログラムの実行時間を測定し、逐次版プログラムの実行時間で正規化した。

図 7はイテレーションレベル並列処理の実行例である。プログラムとしてリバモアループカーネルの 20 番 (図 6) を用い、1 イテレーション内の計算を行なうコードは gcc が生成したコードを元にした。

このプログラムでは配列 XX が依存距離 1 のフロー依存を持つ。従って MISC を用いて DOACROSS 実行を行なうコードは XX(k) と XX(k+1) のアクセスが MISC を用いたメモリアクセスのコードである他は逐次コードと同じである。

Base で示されるグラフが図 6から生成された逐次コードを用いて DOACROSS 実行を行なったときの実行時間を示している。Split で示されるグラフは、イテレーション内の計算を前後のイテレーションに依存する強連結成分と、各イテレーションで局所的に計算できる成分に分割してコード生成を行なったものの実行時間を示す。イテレーション間の依存のある計算とない計算を分けずに、コードスケジューリングで扱える基本ブロック大きくとって生成されたコードの方が、プロセッサ内のパイプラインによる並列性をより利用できるため、1 台で実行した場合は Base の性能が Split の性能を上回っている。しかし、並列実行する場合には強連結成分のスケジューリング長をできるだけ短くした方が、マルチプロセッサシステムで処理できるイテレーション数のスループットが上がるため、Split の方が性能がよくなる。この例では Base は並列化に伴うオーバーヘッドの方が DOACROSS 実行による利得より大きく、逐次実行よりも遅いが、Split は約 15% の実行時間の短縮を実現している。

DOACROSS 実行を行なったときの実行時間の下限は、プロセッサ内の強連結成分のスケジューリング長 × イテレーション数で与えられる。この例では強連結成分は XX(k) の値を使って XX(k+1) の値を計算する部分である。Limit のグラフはプロセッサ内のスケジューリング長から求めた実行時間の下限を表しており、これは Split のグラフとほぼ一致する。

図 8は命令レベル並列処理による逐次ループの並列実行例である。プログラムはセカント法 (1 次補間法) の反復計算

$$x^{(k+1)} = x^{(k)} - f(x^{(k)}) \cdot \frac{x^{(k)} - x^{(k-1)}}{f(x^{(k)}) - f(x^{(k-1)})}$$

を $f(x) = 0.5 - x^3/3! + x^5/5! - x^7/7! + x^9/9!$ としたものを用いた。x の値がイテレーション間でフロー依存を持つため、イテレーション間の並列性は少ないが、強連結成分の部分式の計算を並列に実行できるため、イテレーション内の細粒度並列処理を適用することができる。

このプログラムを静的タスクスケジューリングを行なうプロトタイプコンパイラを用いてコンパイルし、実行時間を測定した。コンパイラに現在インプリメントされているスケジューリングポリシーは ISH である。

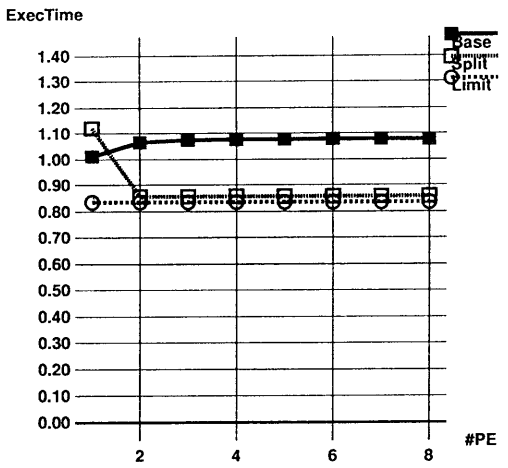

図 7: リバモアループカーネル #20 の実行時間

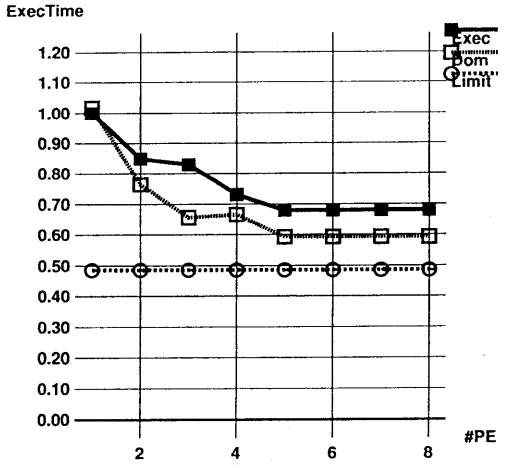

図 8: セカント法の反復計算の実行時間

Exec のグラフはシミュレータ上での実行時間、Dom のグラフはコンパイラが生成したスケジューリングのスケジューリング長を示す。Exec と Dom のグラフ間の隔たりは、バスの使用タイミングの見積りやメモリアクセスに起因するコンパイラの静的な実行時間の見積りと、実際の実行タイミングのずれを示している。

イテレーション間では極性管理とタイミングのためにバリアを張っているため、プロセッサ数を増やした時の実行時間の下限は、タスクグラフのイテレーション内最長パスの長さで評価できる。Limit のグラフはコンパイル時に求めた最長パス長を示す。先行タスクの複製を組み込めば 1 イテレーションのスケジューリング長をもっと短くすることが期待できる。本例では命令レベルの細粒度並列実行によって約 30% の実行時間の短縮が達成されている。

このプログラムでは、メモリアクセスはすべてスカラデータの書き込みと参照から生じたものである。配列データのアクセスやポインタなどの共有バスの使用が静的に予測しにくい要素がある場合は、バス使用のタイミングを静的に評価することはさらに難しくなる。

6 おわりに

逐次化ループを高速化するために、MISC を用いてイテレーションレベルの並列化と命令レベルの細粒度並列処理を行なうコード生成手法を述べた。二つの方法の適用はタスクグラフ中の並列性によって判定されるので、本手法を逐次化ループに適用する部分は最適化コンパイラに組み込むことができる。今後、本コード生成手法を最適化コンパイラに実装し、実プログラムに対する適用範囲と有効性を検証する予定である。

参考文献

[1] Cytron, R., "Limited Processor Scheduling of Doacross Loops," in *Proc. of 1987 International Conference on Parallel Processing*, pp. 17–21, Aug. 1987.

[2] Fisher, J. A., "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Transactions on Computers*, vol. C-30, no. 7, pp. 478–490, July 1981.

[3] 稲垣 達氏, 松本 尚, 平木 敬, "システムの階層的並列性を統一的に扱う最適化コンパイラ," 電子情報通信学会技術研究報告 CPSY-94-40, pp. 105–112, July 1994.

[4] Kruatrachue, B., *Static Task Scheduling and Grain Packing in Parallel Processing Systems.* PhD thesis, Electrical and Computer Engineering Department, Oregon State University, Corvallis, 1987.

[5] Lam, M., "Software Pipelining: An Effective Scheduling Technique for VLIW Machines," in

Proc. of '88 ACM SIGPLAN Programming Language Design and Implementation, pp. 318–328, June 1988.

[6] 松本 尚, "スヌープキャッシュ制御機構の DOACROSS ループへの適用," 並列処理シンポジウム JSPP'92 論文集, pp. 297–304, June 1992.

[7] 松本 尚, 田中 朋之, 森山 孝男, 渦原 茂, "スヌープキャッシュを用いて通信と同期を統合する機構," 電子情報通信学会技術研究報告 CPSY90–42, vol. 90, no. 144, pp. 25–30, July 1990.

[8] Maydan, D. E., S. P. Amarasinghe, and M. S. Lam, "Array Data-Flow Analysis and its Use in Array Privatization," in *Proc. of 20th ACM Symposium on Principle of Programming Language*, pp. 2–15, Jan. 1993.