ループ細分を適用したパスベーススレッド分割手法の初期評価

小川 大仁 [†] 伊里 拓也 [†] 大津 金光 [†] 横田 隆史 [†] 馬場 敬信 [†]
† 宇都宮大学工学部情報工学科

1 はじめに

我々は，バイナリレベルでシングルスレッドコードからマルチスレッドコードへ変換するシステムの研究開発を行っている．

これまで対象となるアプリケーションのループイテレーション単位をスレッドとしてマルチスレッド化を行ってきた．しかし，制御構造が複雑なプログラムに対しては，変数によるスレッド間依存が原因で，スレッド間で同期を取る必要が生じるため並列性が低くなり性能向上を得ることが困難であった．そこで，我々はプログラムの実行経路を任意のポイントで分割しマルチスレッドコードを生成する手法 [1] を開発した．

この手法により，制御構造が複雑なプログラムに対して性能向上を達成した．しかしこの手法では，ループ構造内を分割対象外としていたため対象パス内にループ構造が存在した場合，スレッドサイズの大きさに偏りが出来やすい傾向があった．

本研究では，ループ細分を適用しスレッドサイズの偏りの問題を解消する．

2 ループ細分を適用したパスベーススレッド分割手法

本手法は，実行頻度が最も高いパス (以下#1 パス) に着目しスレッド間依存が存在しないポイントで分割することで，スレッド間の同期を用いない投機的なマルチスレッド実行を可能とする．また，対象内にスレッド間の同期を用いることなく並列化が行えるループ構造が存在した場合，その並列性を利用し更なる高速化を達成する．以下に具体的な手順を示す．

2.1 関数の選定条件

本手法では，対象関数に再帰呼び出しが含まれていると正しいマルチスレッド実行を行うことができない．そのため，再帰呼び出しを含まない関数を前提とする．

2.2 マルチスレッドコード生成手順

マルチスレッドコード生成の様子を図 1(a) に示す制御フローを持った関数と以下の手順を設けて説明する．このとき，楕円はそれぞれ基本ブロックを示し，ループ構造が存在した場合はこれも 1 つの基本ブロックとして扱う．

1. 対象関数内にループ構造が存在する場合，そのループがスレッド間依存による同期を必要とせず並列化が可能か否かの判定を行う．

2. 対象関数のパスの実行頻度やループイテレーション数の情報を得る．

3. 対象パスの依存関係を調査．スレッド間依存が存在しないポイントを分割点候補としてチェックする．

4. 手順 1. の条件を満たすループ構造が存在した場合，ループ細分を適用する．(ここでは，性能低下とならないループイテレーション数の計算を行う)

5. 手順 3.，4. を踏まえ，スレッドサイズが均等になるように分割点を決定する．

手順 1. では，ループ細分が適用可能かどうかの判定を行う．次に，手順 2. によって#1 パスを特定しこれを分割対象パスとする．手順 3. では，分割点となり得る全てのポイントを特定する．図 1(a) では，白抜きされた基本ブロック列が#1 パスを示し，基本ブロック 3 と 4，5 と Loop の間が分割点の候補となる．手順 4. でループ細分が適用可能なループ構造が存在したとき，ループを複数のイテレーション単位に細分する．このとき大きいイテレーション単位でループを細分すると性能に悪影響を与える可能性があるため影響を受けないイテレーション数を求める必要がある．この問題解決に関しては，次節にて述べる．最後に，手順 3.，4. の情報をもとにスレッドサイズが均等になるように調整を行い，スレッド間の同期を必要としないマルチスレッドコード (図 1(b)) が生成される．また，このループ構造がループ細分適用対象外だった場合は，手順 4. を踏まずに図 1(b') に示す形でコードが生成する．

このとき，#1 パス以外の実行パスもスレッド間依存が存在しない限りは，スレッド内に含めるものとする．図 1 で基本ブロック 7 は，スレッド間依存とはならないためスレッド内に含める．こうすることで，投機ミスの機会を削減し性能向上の機会を増加させる．また，本手法はループレベル並列による高速化と比較してもループ部分以外も並列実行しているため，より高い性能を期待できる．

2.3 ループ細分の制約条件

2.2 節の手順 4. より，ループ細分を適用する際，対象ループ内でメモリアクセスが多いあるいはループイテレーション数が多い場合，大きいイテレーション単位でループを細分してしまうと性能低下を引き起こす可能性がある．これは，データキャッシュミス及び我々が使用しているシミュレータが所有するメモリバッファ (投機状態であるスレッドのストア情報を保持する) の情報が溢れてしまうことが原因である．

そこで，以下の条件式により性能低下が起こらないループ細分時のイテレーション数を求める．

○データキャッシュ

$$X \leqq \frac{\text{データキャッシュサイズ} - BeforeMA}{MA \times \text{スレッドユニット数}} \quad (1)$$

○メモリバッファ

$$X \leqq \frac{\text{メモリバッファサイズ}}{\text{1 イテレーションでのストアデータサイズ}} \quad (2)$$

X:性能低下とならない最大ループイテレーション数
Before MA:対象ループ以前のメモリアクセスデータサイズ
MA:1 イテレーションでのメモリアクセスデータサイズ

式 (1)，(2) より求められるイテレーション数の中で小さい値のイテレーション数を採用することで，性能低下の影響を受けることがなく本手法を適用することが出来る．

3 評価

評価にはスレッドパイプライニングモデルシミュレータの SIMCA[2] を用いた．評価手順として，まず対象アプリケーションのコードを SIMCA 用 gcc クロスコンパイラ (version 2.7.2.3) に最適化オプション-O3 を適用してコンパイルする．そして，その生成されたバイナリコードに対して本手法を適用しマルチスレッドコードへと変換する．

Preliminary Evaluation of a Path Based Thread Partitioning Technique Using Strip Mining
†Hirohito Ogawa, Takuya Iri, Kanemitsu Ootsu, Takashi Yokota and Takanobu Baba
Department of Information Science, Faculty of Engineering, Utsunomiya University (†)

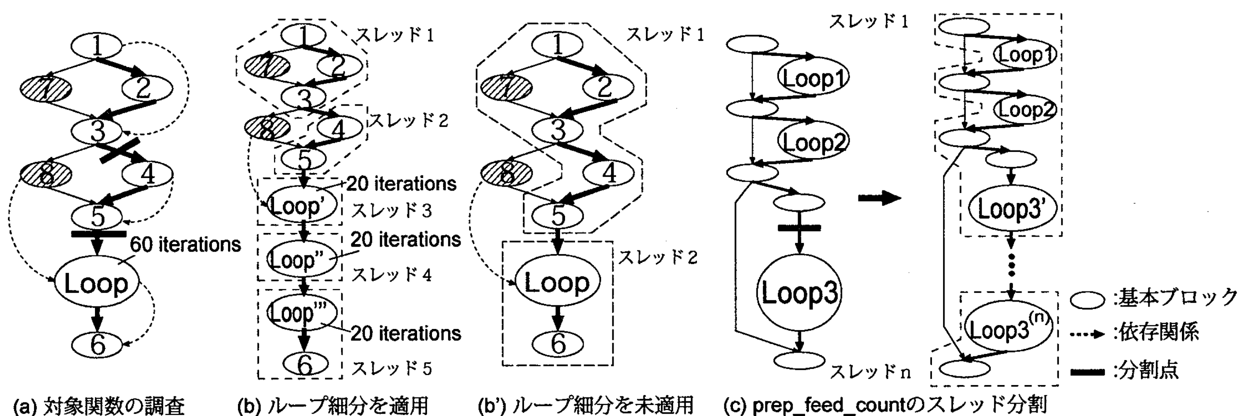

図 1: ループ細分を適用したスレッド分割手法

表 1: 性能低下とならないループイテレーション数

	4台	8台	16台
イテレーション数	14回	10回	5回

評価方法は，スレッド分割対象関数の最初から終了までの実行サイクル数を計測し速度向上率を求める．速度向上率は，マルチスレッド実行サイクル数をシングルスレッド実行サイクル数で割ることで求められる．そして，本手法による性能向上はループレベル並列のみによる性能向上よりも高い性能を得ることが出来ることを示すため，ループレベル並列による高速化との比較も同時に行う．また，前節でループ細分時に性能低下を起こさないイテレーション数を求める説明をしたが，本研究で用いるシミュレータ SIMCA ではイテレーション数を考慮してもデータキャッシュミスを起こす可能性がある．これは，投機状態のスレッドのストアデータ情報は，データキャッシュではなくメモリバッファへと格納され，メモリにデータを書き戻す (Write-Back) 際にデータキャッシュにデータがなくデータキャッシュミスを起こすというアーキテクチャ上の問題が原因となっている．本研究では，データキャッシュミスを考慮したスレッド分割を行っているため，Write-Back 時におけるデータキャッシュミスを無視するシミュレータのオプション"ideal" を採用する．

評価対象は，2.3 節の条件式により性能低下を防ぐことが可能な関数として，*SPEC CINT2000* から *300.twolf* より関数 *prep_feed_count* を選んだ．また入力データとして *train* を用い，4 台，8 台，16 台のスレッドユニット台数でそれぞれシミュレーションを行った．

prep_feed_count のスレッド分割の様子を図 1(c) に示す．まず，この関数にループ構造は 3 つ存在し，その中で 3 番目のループがループ細分を適用可能であった．実行パスは，1 通りしか存在せず実行頻度は 100%であった．次に，対象パス上の依存関係を調査した結果，図 1(c) の左図の太線が示す位置が分割点候補となった．さらに，細分対象ループにおけるメモリアクセスデータ量を考慮した結果，表 1 に示す回数以下でループを細分すれば性能低下とならない．そこで，表 1 のイテレーション数を越えないかつスレッドサイズが均等になるように調整し図 1(c) の右図のようにスレッドへと分割した．

速度向上率を図 2 に示す．“本手法” は本手法によるシミュレーション結果を示し，“ループ並列” はループレベル並列による高速化のシミュレーション結果を示す．本手法で実行したときスレッドユニット台数が 4 台で 3.57 倍，8 台で 6.60 倍，16 台で 11.58 倍と顕著な性能向上を達成した．これは，本手法によって各スレッドのサイズが均等になるように調整されたため高い並列性を実現したものと考えられる．また，ループレベル並列のみによる高速化と比較しても高い性能を得る結果となった．

本手法は，投機的なマルチスレッド化手法であるため，スレッド内に含むことが出来ない実行パスが存在すれば投機ミスを起こしてしまう．今回の評価では投機ミスとなる機会がなかったため，今後投機ミスを起こした際どのように性能に影響してくるかを調査する必要があると考えられる．

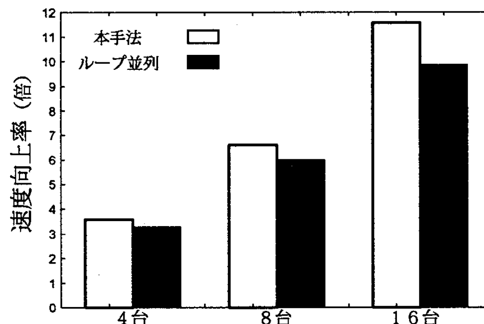

図 2: 速度向上率

4 おわりに

本稿では，ループ細分を適用したスレッド分割手法を考案し評価を行った．その結果，スレッドサイズが均等に保たれることから高い性能向上を得ることに成功した．また，ループレベル並列による高速化と比較しても高い性能を得ることを示した．今後の課題として，評価対象を増やすことで投機ミスに関する性能の影響を調査する必要がある．

謝辞

本研究は，一部日本学術振興会科学研究費補助金（基盤研究 (B)18300014，同 (C)19500037，若手研究 (B)17700047）および宇都宮大学重点推進研究プロジェクトの援助による．

参考文献

[1] 小川大仁, 小林崇彦, 大津金光, 横田隆史, 馬場隆信, “パスの実行頻度を考慮したスレッド分割手法の初期評価”, 情報処理学会 第 69 回全国大会講演論文集, pp., 2007

[2] J. Huang, “The SImulator for Multi-threaded Computer Architecture(Release 1.2)”, http://www.cs.umn.edu/Research/Agassiz/Tools/SIMCA/simca.html.