グラフ理論に基づくスレッド分割手法の適用検討

阿部 武志 † 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

近年マルチコアプロセッサが計算機の標準搭載部品になりつつある．しかしながら既存のプログラムの大多数はシングルスレッドコードで書かれているため，マルチコアプロセッサの性能を最大限に活用するためにはマルチスレッドコードへの変換が必要不可欠である．

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には様々な問題を考慮しつつスレッド分割を行う必要があるため，様々な研究がされている．そういった研究の 1 つに Johnson らの，制御フローグラフにおける最小カット集合を求めることで最適なスレッド分割箇所を決める手法 [1] がある．彼らの手法は，マルチスレッド化の際に問題となるデータ依存，スレッド予測，ロードバランスを考慮しつつ，ループと直線コードを一緒に扱いマルチスレッド化できるという特徴がある．

そこで本研究では，Johnson らの手法を実用プログラムに適用し，その実用上の問題点と改善方法を検討する．

2 グラフ理論に基づくスレッド分割手法

この節では本稿が前提とする Johnson らのグラフ理論に基づくスレッド分割手法について説明を行う．この手法は大きく分けて以下の 6 つの工程によってマルチスレッド化を行う．

(1) プロファイル情報の取得

まず始めに，対象となるプログラムの制御フローグラフを作成する．また，1 ブロックあたりの命令に関する静的な情報，分岐確率，1 関数呼び出しあたりの平均サイクル数，依存関係の情報を得る．ここで作成した制御フローグラフの基本ブロックをグラフ理論のノード（頂点），制御フローをグラフ理論のエッジ（辺）としてスレッド分割を行っていく．

(2) エッジの重み計算

(1) で作成した制御フローグラフの各エッジに重み $\mathcal{W}$ を割り付けていく．重み $\mathcal{W}$ はエッジ e_i がスレッド境界に選ばれたときに，そこでスレッド分割することによって生じるオーバーヘッドを表し，$\mathcal{W}(e_i) = \mathcal{D}(e_i) + \mathcal{P}(e_i)$ の式によって計算される．$\mathcal{D}$ は依存ペナルティを表し，$\mathcal{P}$ は予測ペナルティを表す．

依存ペナルティとは，スレッド間でデータ依存があった際にそれを適切に処理するためにスレッド間で行われる最も長い同期待ちに費されるサイクル数のことを表し，依存のプロファイル情報から見積もられる．予測ペナルティとは，スレッド境界が予測できない制御フローの領域にあるとき，スレッド予測のミスにより廃棄される後続スレッドによって無駄にされるサイクル数の期待値を表し，$\mathcal{P} = thread\_size * (1 - 2 * |0.5 - bfreq|)$ の式によって見積もられる．$thread\_size$ は後続スレッドの処理に要するサイクル数，$bfreq$ は分岐確率を表す．この式によって完全に公平な分岐 (50%) ほど予測ペナルティは大きく，完全に偏った分岐 (0%または 100%) ほど予測ペナルティは小さくなる．

Evaluation and Consideration of a Thread Partitioning Method based on Graph Theory
†Takeshi Abe, Kanemitsu Ootsu, Takashi Yokota and Takanobu Baba
Department of Information Science, Faculty of Engineering, Utsunomiya University (†)

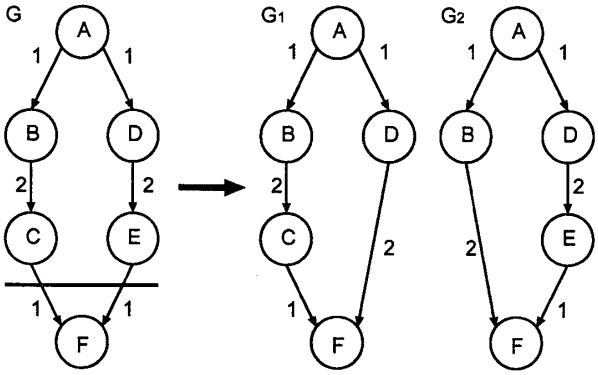

図 1: ノード移動

(3) スレッド境界の決定

(2) で割り付けた重みをもとに，スレッド境界を最小カット (Min-cut) を使って求める．最小カットとは，フローネットワークにおける最大フロー問題と呼ばれる最適化問題に関する定理である最大フロー最小カット定理を示し，任意の 2 点間で，一方から他方へ流れる物量は，その経路内で最も弱い部分によって制限され，そこが最大物量となる，というものである．本手法ではそれを用いて，1 つのグラフを 2 つに分割する際にエッジの重みが最小となる箇所で分割を行いマルチスレッド化をしている．なお，ここで求められたスレッド境界は候補であって，実際に分割するスレッド境界は (4) の工程で決定される．

(4) ロードバランス化

負荷不均衡 (Load Imbalance) を防ぐために，(3) で選考したスレッド境界のバランスをとる．これはマルチスレッド実行の性能を表す測量 $\mathcal{M}$ を使い最高性能のスレッド境界を求める次の①〜⑤の作業によってバランスがとられる．測量 $\mathcal{M}$ は実行時間とスレッド境界に選ばれたエッジの重みとの和で表され，値が小さいほど性能はいい．

① (3) で決定したスレッド境界以外の分割箇所を選定するために，図 1 のようにスレッド境界に隣接しているノードを 1 つのノードに結合して②で最小カットを使ったときに前回と同じスレッド境界が選ばれないようにする．

② 図 2 のように，①でノードを移動したグラフのスレッド境界を最小カットを使って求める．

③②で得られたグラフの測量 $\mathcal{M}$ を計算し，ノード移動をする前のグラフの測量 $\mathcal{M}$ と比較する．

④ 比較した結果，ノード移動をしたグラフの方が性能がよければ，そのグラフを使って①〜④を繰り返し行い，性能が上がらなくなるまで続ける．

⑤ ①〜④によって求められた最高性能となるスレッド境界による分割と，分割しないものとを測量 $\mathcal{M}$ を使って比較し，分割した方が性能が上がる場合は実際にそこがスレッド境界に決定され，分割をしない方が性能がいい場合は分割は行われない．

(5) スレッド生成

(2)〜(4) を繰り返し行い，プログラムを複数のスレッドに分割する．

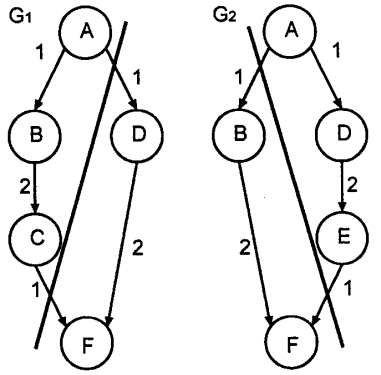

図 2: 図 1 の制御フローグラフの最小カット

(6) エッジの摂動

局所的な最適化を避けるために，1 度も調べられなかったエッジを，そこをスレッド境界とすることで性能が上がるかどうかを厳密に 1 回プログラムの終わりから調査し，性能が上がるようならば実際に分割される．

3 評価

3.1 評価方法

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である SIMCA[2] を使用する．

評価対象アプリケーションとして *SPECCINT2000* の *164.gzip* を用いる．本手法の対象としたのは，コードが長すぎず，ループ，分岐を持つ関数 *fill_window* である．入力データには *test* および *train* を用いる．バイナリレベルでマルチスレッド化を行うために上記のアプリケーションのコードを SIMCA 用クロスコンパイラ (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を行い，それによって生成されたバイナリコードに対して本手法を適用し，マルチスレッド化を行う．

評価は，対象とした関数のシングルスレッド実行サイクル数とマルチスレッド実行サイクル数の比によって求められる速度向上率によって行う．

3.2 評価結果

本手法を適用する上で必要となるプロファイル情報は，SIMCA 上で *164.gzip* のシングルコードを実行し，関数 *fill_window* の各基本ブロックのサイクル数，データ依存による同期待ちにかかるサイクル数，分岐確率，1 関数呼び出しあたりの平均サイクル数の計測を行うことによって取得した。その情報をもとに本手法を適用し，マルチスレッド化を行った。

関数 *fill_window* の速度向上率を表 1 に示す．スレッドユニット台数は 4 台である．速度向上率が *test* で 1.32 倍，*train* で 1.29 倍と高速化を達成した．スレッドユニット台数 8 台，16 台でも実行したが速度向上率は変わらなかった．これは関数 *fill_window* にこの手法を適用してもあまりスレッドが作成されず，結果的に 5 台以上のスレッドユニットを使わないためである．

関数 *fill_window* でスレッドがあまり作成されなかった原因の 1 つに，この関数が 1 イテレーションあたりの実行にかかるサイクル数が小さく，イテレーション数が多いループを持っていたことが考えられる．この手法では，ループ内でスレッド境界がとられると 1 イテレーションごとにスレッドが作成される．そのため，この関数のようにループの 1 イテレーションあたりの実行にかかるサイクル数が小さいプログラムでは，ループを分割する利益より，分割によって生じるスレッド制御によるオーバーヘッドのコストの方が大きいため，ループでの分割が行われない．関数 *fill_window* ではそのようなループが複数存在し，分割時に負荷不均衡を引き起こしやすいため，依存関係が複雑でなかったわりにスレッドがあまり作成されなかった．

表 1: 速度向上率

ベンチマーク	*fill_window*	
入力データ	*test*	*train*
速度向上率	1.32 倍	1.29 倍

3.3 改善手法の検討

164.gzip の関数 *fill_window* のように 1 イテレーションあたりの実行にかかるサイクル数が小さく，イテレーション数が多いループを持ったプログラムに本手法を適用する上でさらなる高速化を達成するためには，ループの複数回ぶんの命令列を一回の繰り返しで行うよう展開するループアンローリングが重要であると考えられる．本手法を適用する前にループアンローリングを適用し，あらかじめ該当するループを適切なアンローリング回数で処理し，マルチスレッド化することによって，ループを分割する利益がスレッド制御によるオーバーヘッドのコストを上回るため，本手法では分割されなかったループの適切な分割が可能となり，さらなる速度向上の可能性が期待される．また，ループアンローリングをあらかじめ適用することによって，本手法を適用することによって生成されるスレッドのサイズもループアンローリングを適用しないものと比べて全体的に小さく，負荷不均衡も引き起こりづらくなると考えられる．

4 おわりに

本研究では，シングルスレッドコードをマルチスレッドコードに変換するために，データ依存，スレッド予測，ロードバランスを考慮した Johnson らの，制御フローグラフにおける最小カット集合を求めることで最適なスレッド分割箇所を決める手法の適用検討を行った．評価は *SPECCINT2000* の *164.gzip* を対象とし，コードが長すぎず，ループ，分岐を持つ関数 *fill_window* に本手法を適用した．

評価の結果，ある程度の速度向上は得られたが，期待した速度向上は得られなかった．その原因の 1 つにこの関数が，1 イテレーションあたりの実行にかかるサイクル数が小さく，イテレーション数が多いループを複数持っていたことが挙げられた．

本手法ではループを分割する際に 1 イテレーション毎にスレッドの生成が行われるため，1 イテレーションの実行にかかるサイクル数が小さく，イテレーション数が多いコードに対して本手法適用しても性能向上が小さいことが分かった．その改善手法としてループアンローリングを本手法を適用する前に適用する方法が考えられる．そのため今後の課題として，ループアンローリング適用後に本手法を適用する有効性について検討し，性能評価を行うことが挙げられる．

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] Troy A. Johnson, et al.., “Min-Cut Program Decomposition for Thread-Level Speculation”, Proc. PLDI 2004, pp.59-70, 2004.

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