投機的実行による難並列化ループの高速化

[1]石川 隼輔 [1,2]山名 早人
[1]早稲田大学大学院理工学研究科 [2]アドバンスト並列化コンパイラ研究体

1. はじめに

プログラム実行時間の大部分をループ構造が占めることからループ並列化による速度向上率は高い。しかし、静的に解析できないループ運搬依存(LCD)が存在する場合、従来の並列化手法では実行時の依存関係の有無に関わらず、依存が存在すると仮定し解析を進める。このため、例えば、実際には1万回に1回しか LCD が発生しないようなループであっても並列化不可能である。しかし、このような難並列化ループに対しては投機的実行(SpeculativeExecution)を適用することでプログラムを高速に実行することが可能となる。我々は既にループへの投機的実行適用手法として、従来考慮されてこなかった投機的実行失敗時の復帰処理に必要なオーバーヘッドをパラメータとして取り入れ、投機的実行の効果が期待できる部分のみを選択的に投機的実行する方式[1]を提案している。

本稿では、提案した投機的実行適用手法の概要を示すと共に、提案手法を OpenMP によって実装する場合に問題となる Section 切り替えオーバーヘッドを隠蔽するための新たな投機的実行の実装方式を提案する。

2. 選択的投機的実行適用手法

投機的実行適用手法について概要を示す。対象とする難並列化ループは、図 1 の左上に示したような LCD 依存距離が不定で静的に Iteration 間の依存関係を解析できないループと定義する。ここで、LCD が実行時にどの程度の確率で存在するのかを表す値を**LCD 存在確率**、ループ Body 開始位置から LCD の原因となる変数の値が決定さ[1]れるまでにかかる時間を **LCD ディレイ**、ループ Body 開始位置から LCD へ DataSpeculation を適用し投機的実行を開始するまでにかかる時間を**投機的実行開始ディレイ(SE ディレイ)**と定義する。

提案手法では図 1 のように、投機的実行開始位置において LCD の原因となるすべての変数に DataSpeculation を適用し、後続の Iteration を投機適時実行開始することによってループを高速実行する。投機的[2]実行を効果的に行うためには以下の 2 点を投機的実行の効果が得られるように決定しなければならない。

- 投機的実行適用実行 Path
- 投機的実行開始位置

まず、ループ中に存在する複数の制御フロー集合である実行 Path に分割する。実行 Path はループ Body 内の出口が等しい制御フローの集合である。複数存在する実行 Path の中から、投機的実行を適用する実行 Path を一本選択し、これを**投機的実行適用実行 Path** とする。実行時には、投機的実行適用実行 Path が常に選択されると仮定し ControlSpeculation する。一方、投機的実行適用実行 Path 上に存在する LCD に対しては DataSpeculation を適用することによりループ運搬依存を除去する。したがって、投機的実行適用実行 Path 選択においては、対象とする実行 Path の実行頻度が高く、LCD に対して適用する DataSpeculation 成功率が高いほど、投機的実行の効果を得ることができる。

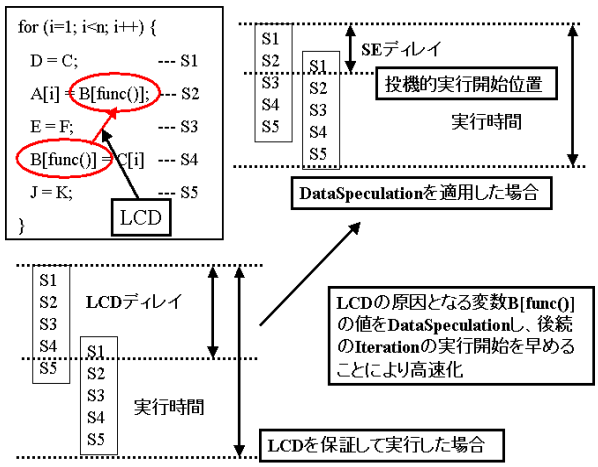


図 1 投機的実行によるループ並列化

次に、投機的実行対象実行 Path において、投機的実行開始位置(DataSpeculation 適用開始位置)を選択する必要がある。従来の研究[2]では、ループ Body 開始時点で投機的実行が適用されている。しかし、投機的実行開始位置が遅れるにつれ、すでに実行されたコードから得られる情報量は多くなる。得られる情報量が多ければ、投機的実行対象実行 Path への ControlSpeculation の成功率、LCD への DataSpeculation の成功確率が上がる。すなわち、Iteration に対する投機的実行成功確率が上昇する。一方、投機的実行開始位置が遅れるにつれ、SE ディレイは大きくなり、投機的実行の効果が少なくなる。よって、投機的実行成功率と SE ディレイを考慮して、適切な位置で投機的実行を開始することが重要となる。

提案手法では、投機的実行失敗時のオーバーヘッドをパラメータとして考慮した上で、投機的実行実行 Path と投機的実行開始位置のすべての組み合わせにおいてプログラム実行

"A Speedup Technique to Difficultly Pallalyzing Loops Using Speculative Execution" [1] Shunsuke Ishikawa, [1,2] Hayato Yamana·

[1] Graduate School of Science & Engineering, Waseda University

[2] Advanced Parallelizing Compiler Project

時間をシュミレーションし、最速となる組み合わせにおいて投機的実行を行う。

3. 実装方式の改良

実装にはOpenMPを使用し、ループの各Iterationを複数のSectionにより並列実行する。具体的には、i番目のSectionがn番目のIterationを実行し、投機的実行開始位置に至った時点で、i+1番目のSectionを起動しn+1番目のIterationを実行開始させることを繰り返すことでループを並列実行する。本手法ではSection起動のOverHeadを削減するために、必要数のSectionをParallelSectionsにより一度に生成し、各Sectionをスピンロックによる Busy/Wait コードによって制御する。従来手法と提案手法のOpenMPによる実装イメージを図2,3に示す。Sectionを起動する側をSection1、起動される側をSection2とし、n番目のIterationにおいて、投機的実行開始位置を境界として前半部をIteration(n)-Part1、後半部をIteration(n)-Part2と定義する。また、仮定するループのIterat依存関係は、図2に示すようにIteration(n)-Part2とIteration(n+1)-Part1の間にデータ依存が存在するとする。

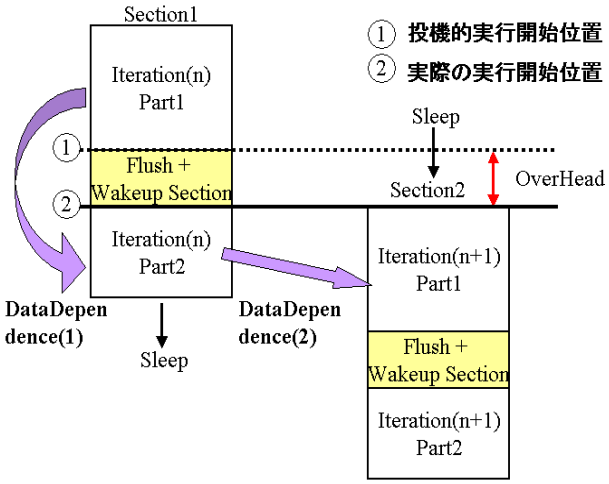


図2 従来手法の動作イメージ

図2のように従来手法ではSection1でIteration(n)-Part1の実行が終了した時点で、DataSpeculationしたデータ及び、自Section内で変更したすべての変数をflushし、Section2をWakeUpする。その後、Section1では引き続きIteration(1)-Part2実行を行ったあと、他Sectionによって起動されるまでSleepする。また、Section2では次のIterationであるIteration(n+1)-Part1を実行開始する。従来手法は上記処理を繰り返し実行する。

従来手法における最大の問題点はSectionのWakeUpと実際にSectionが起動するまでの間にOSレベルでの時間的オーバーヘッドが存在することである。また、従来手法ではSection1はIteration(n)を実行後Sleepし、次にWakeUpされるまでIdleとなっていることから計算資源を十分に活用できていない。これに対し提案手法では、図3のようにSection1は常にIteration(x)-Part1[x=1,2…end]を実行し、Section2はIteration(x)-Part2[x=1,2…end]を実行する。また、SpeculativeTaskQueueを導入することでSection間のWakeUp/Sleepによるオーバーヘッドを削減している。提案手法では、Section1においてIteration(n)-Part1の実行が終わった時点でIteration(n)-Part2で必要となるデータを予測し、データ(Task)をSpeculativeTaskQueueへとPutする。その後、Sleepには入らず引き続きIteration(n+1)-Part1を実行開始する。このように提案手法ではSection1がSection2をWakeUpする処理が省略される。Section2は常にSpeculativeTaskQueueを監視して、Taskが存在する限りPopして実行することを繰り返す。このように提案手法では従来手法に比べ、Section切り替えオーバーヘッドを減少することが可能であり、また同時に計算資源を無駄なく活用することが可能である。

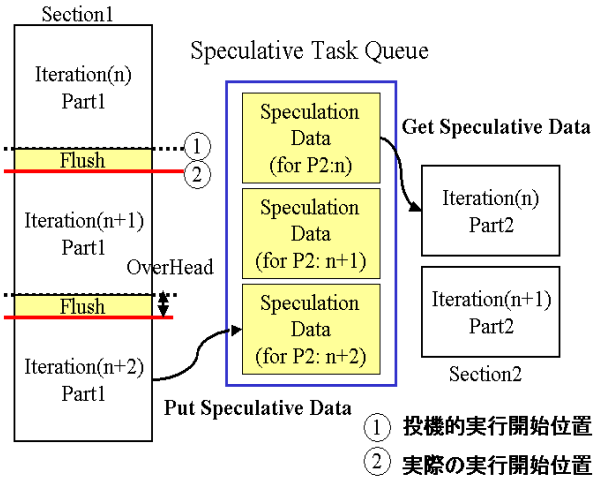


図3 提案手法の動作イメージ

また、図3ではSeciton1において1Iteration実行ごとにFlush()を実行している。しかし、一般にFlush()の実装では、全ての変数を共有メモリに書き戻す作業を行うため、キャッシュミスが増大しパフォーマンスが低下する。よって、提案手法では、Iteration-Part1とIteration-Part2のサイズを考慮してFlush()の回数を制限する手法をとっている。具体的には、k個のIteration実行ごとにFlush()を行う方式をとり、kはIterationの実行時間を考慮して決定する。

4. ベンチマークによる検証

上記手法を、SPECInt95 129.compressの実行時間の99%を占める関数compressに対して適用した場合、RegattaH690上で従来手法にくらべて1.43倍高速化が可能であることがわかった。今後は、投機的実行を前提としたLoopUnrollingを提案アルゴリズムに導入するなどして、さらなる高速化を目指す。

参考文献

[1] 石川隼輔, 山名早人:"ループへの効果的な投機的実行適用手法", 情処研報(HPC), Vol.2002, No.22, pp.121-126 (2002.3)

[2] J.GSteffan and T.C.Mowry: "The potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization", Proc. of HPCA-4, pp.2-13 (1998.2)