並列データ処理基盤を用いた並行バグ並列検査方式の検討

荒堀　喜貴
東京工業大学

1　はじめに

データ競合やデッドロックに代表される並行処理の不具合（並行バグ）の検査は古典的な問題であり，現在までに多数の検査手法が提案されている．しかし，これらの検査手法のほとんどが現代のチップマルチプロセッサや並列データ処理基盤の登場以前に考案された逐次アルゴリズムに基づいている．そのため，現代的な並列計算環境の活用という観点から見た場合，従来の並行バグ検査方式は性能が十分でない．計算環境の並列化の進展に伴い今後ますます並行処理の普及が進む一方，並行処理の大規模・複雑化とそれによる検査時間の増大が深刻になっており，従来の性能を大幅に上回る並行バグ検査方式が求められている．

そこで，本研究は，並行バグ検査の基本的な構成要素である動的競合解析に的を絞り，それを現代の並列計算環境に合わせて並列化する方式を検討する．具体的には，動的競合解析の代表であるロックセット解析の並列化を目指し，従来の解析方式が直面する低性能の要因を明らかにした後に，その要因をチップマルチプロセッサ上での並列データ処理技術によって克服する新たな解析方式を検討する．

2　並行バグの検査

2.1　背景：動的競合解析

動的競合解析（Dynamic Datarace Detection，以下では DDD と略称する）は，並行処理の不具合（並行バグ）の検出手法の一種であり，対象プログラムの実行時の情報を観測することでデータ競合を検出する．ここで，実行時の情報とは，対象プログラムを実際に実行することで取得したメモリアクセス，スレッド生成/破棄，ロック/アンロック等同期操作の実行履歴である．また，データ競合とは，(1) 異なる 2 つのスレッドが同じメモリオブジェクトをアクセスし，(2) その内の少なくとも 1 つが書込みアクセスであり，(3) 両アクセスの間に順序の強制がない，という条件（競合条件）を満たす 2 個のメモリアクセスのペアである．

従来の DDD では，対象プログラムが使用するメモリオブジェクトごとに個別のメタデータを関連付け，各オブジェクトに対してどのようなアクセスが行われたかという履歴を管理する．例えば，対象プログラムのスレッド `t1` が配列 `a` に対して書込みアクセスを実行した場合，`a` に関連付けられた履歴にはそのアクセスが記録される．このようにして，対象プログラムの実行中に発生するメモリアクセスを対象の履歴に逐一記録して行き，記録と同時に履歴内のアクセスを調べることで各アクセス間での競合条件の成立を判定する．競合条件が成立した場合，それらのアクセスの組を競合として警告する．

このように，従来の DDD は（1）各オブジェクトに履歴を関連付け，(2) 各アクセスを該当の履歴に記録し，(3) 履歴内のアクセスを調査して競合条件を判定する，という 3 つの処理によって競合解析を実現する．

2.2　問題分析

従来の DDD は発案当時の時代の計算環境では有効性を示せたが，現代のコア数の多いチップマルチプロセッサ上で実行した場合，次の問題点に直面する．

- **問題点:** 多数のスレッドによる履歴操作が頻繁に衝突を起こすため，競合解析の実行オーバヘッドが無視できないほど大きい．

従来の DDD では，上述したように各メモリオブジェクトに 1 個の履歴を関連付けるが，これらの履歴は検査器が集約管理する．検査対象の各スレッドがメモリアクセスを実行する度に毎回，検査器はアクセス対象のオブジェクトの履歴を参照し場合によっては更新する．したがって，この間に他のスレッドから履歴の参照や更新の要求があった場合，その要求は先のスレッドの履歴操作が完了するまで待たされる．このような履歴操作の衝突は，並列実行されるスレッドの数が少ないうちは大きな問題にはならない．しかし，現代的なコア数の多いチップマルチプロセッサ上で多数のスレッドを同時に走らせる並行処理では，履歴操作の衝突による検査時間の増大は無視できないほど大きい．

この問題に対する一つのアプローチは，対象プログラムのソースコードを解析して無駄な競合検査（履歴操作）を削減することである．例えば，データフロー解析を応用することで冗長な検査を発見し削除したり，エスケープ解析を応用することで複数スレッド間で共有されないメモリオブジェクトを特定し履歴を関連付けないようにする（履歴操作を無くす）などの最適化

が考えられる．実際，現在までに提案されてきた競合解析の高速化手法は，このアプローチによるものが主流を占める．

しかし，このアプローチは履歴操作の衝突問題をある程度緩和することはできるものの，衝突の発生を根本から解決するものではない．なぜなら，各スレッドの共有オブジェクトに対する競合検査の省けないアクセスは必ず履歴操作の衝突を引き起こすからである．

3 提案: 並列データ処理よる並列競合解析

3.1 概要

前節の問題分析に基づき，本研究は，履歴操作の衝突を並列データ処理によって回避する競合検査方式を提案する．具体的には，Choi らが PLDI'02 で発表したロックセット解析 [1] を以下の要領で MapReduce[2] に基づく実装に改変することで衝突の起きない競合解析を実現する：

- **要点 1:** 各メモリオブジェクトにではなく各スレッドに個別の履歴を関連づけ，そのスレッドが実行したメモリアクセスを記録する．
- **要点 2:** 一定の間隔で，各スレッドの履歴を連結したものに Map 操作を適用し，スレッド単位の履歴からメモリ単位の履歴を構築する．
- **要点 3:** Reduce 操作では各メモリオブジェクトの履歴を調べて競合条件を判定する．

この実装のポイントは，履歴をメモリ単位ではなくスレッド単位で管理することにより衝突を回避することと，スレッド単位の履歴管理で一時不可能となった競合判定を MapReduce で並列に実行することである．なお，ここでの MapReduce 処理系には，速度と処理対象のデータ量（履歴サイズ）を考慮し，Stanford 大が開発したマルチコア MapReduce 基盤である Phoenix[3] を使用している．

3.2 実験結果要約

提案手法の有効性を確認する予備実験の一部として，Intel Xeon E5-2660（8 core/chip）2.2GHz を 2 チップ（計 16 コア），32GB RAM，1TB SATA 6Gb/s ハードディスク，Linux 2.6.32 を備えた並列計算機上で，マルチスレッド HTTP クライアント aget-0.4.1 の競合検査を実行し，(1) 衝突の回避状況，(2) MapReduce による競合解析のスケーラビリティ，(3) 履歴サイズの変化による速度向上比の変化，を計測した．その結果，次のことを確認できた：

- **結果 1:** 提案する並列ロックセット解析では複数スレッド間の履歴操作の衝突は（Map フェーズと Reduce フェーズの境界での同期を除いて）ほぼ解消された．
- **結果 2:** 並列ロックセット解析の実行時間性能は各コアに 1 つの Mapper または Reducer スレッドを割り当てる方式で 14 コアまで線形にスケールした．
- **結果 3:** 最大の速度向上比（16 コアでの解析時間/1 コアでの解析時間）が得られた履歴サイズは 2.4GB であった．

このことから，総合的な検査時間の評価は今後の課題であるが，現代的な並列計算環境での競合検査の性能向上という目的に本研究の提案方式が有望であることが分かった．なお，上記の結果 2 で 16 コアまでスケールしなかった理由は，Phoenix の実装の都合で Mapper と Reducer 以外のスレッドが特定のコアを占有するためであるが，並列計算機のコア数が 16 より増えた場合もコア数が約 30 程度までなら線形にスケール可能と予想している．また，上記の結果の他に Map フェーズと Reduce フェーズの実行時間の比率を計測した結果，Reduce フェーズの実行時間が約 5 割と非常に大きかったため，両フェーズのパイプライン実行等の最適化により更なる性能向上を見込めることも分かった．

4 議論

発表会場での質疑応答をもとに，著者がその後の検討を加えた提案方式の改善，限界，課題について以下の通り説明する．なお，ここでの考察の誤りや不備は，会場での質問者によるものではなく，全て著者の責任に帰属するものである．

- **考察 1:** 履歴サイズ増加への対処法
  MySQL 等の大規模マルチスレッドプログラムを対象に競合検査を行う場合，上記実験に比べ履歴サイズの大幅な増大が予想される．その場合の対処法としては，(a) ソースコード解析でイベント履歴のサイズを減らす，(b) サイズにスケールアップするよう並列データ処理自体を改良する，などが考えられる．本研究の今後の方針としては，(a) の戦略を採る予定である．その根拠は，Choi らの元の解析自体が履歴サイズの上限を抑える最適化を備えており，その最適化が強力であるため，大規模なプログラムに対しても履歴サイズが現状のデータ処理基盤で取扱い可能なサイズに収まると予想しているためである．

- **考察 2:** 分散 MapReduce 処理系の適用可能性
  上記実装及び実験で使用したインメモリ型の MapReduce 処理系の代わりに Hadoop 等の分散型の MapReduce 処理系を適用した場合の性能評価は，(a) 考察 1 の予想が外れ履歴サイズが実際は巨大になる場合もあるのでは，(b) その場合は競合解析の部分を例えば Amazon EC2 で数万台の解析用インスタンスで超並列実行した方が有利にならないか，という観点から検証する必要がある．この点の検証については，今後，履歴サイズによる提案方式の有効性の限界を分析する上で明らかにする予定である．

- **考察 3:** 小さ過ぎる履歴への対処
  上記実験では，履歴サイズが小さ過ぎる場合（例えば 100MB 級）にも提案方式は満足な速度向上比が得られなかった．この問題の主要因は「履歴が小さい場合は Mapper と Reducer の間で比較的頻繁に同期が起きるため」と予想しているが，現時点での原因分析は不十分であり，今後詳細な分析を行う必要がある．

- **考察 4:** 他の競合解析技術との使い分け・併用
  本研究の方式を含め DDD は，他のソースコード検証やモデル検査等の静的解析技術と相補的な役割を果たすと考えられる．つまり，対象に含まれる全ての並行バグを完全に検出しようとするなら，DDD だけでは検出漏れ（false negative）が多いため，静的解析の活用が必要な場合もある．また逆に，対象に含まれる並行バグを正確かつ高速に検出しようとするなら，静的解析技術だけでは誤検出（false positive）や検査コスト（モデル記述やソースコード修正等）が多いため，DDD の活用が必要な場合もある．要するに，現状は DDD と静的解析のどちらか一方だけで利用者の多様な要求を満たせないので，両者の特性に応じた使い分けないし併用が必要である．

- **考察 5:** 競合解析の応用
  とあるゲームのプログラムでは，複数プレイヤが危険なタイミングで同時にアイテム操作を行うと本来ならば違法なアイテム増殖が達成されてしまう，というバグがあった．このように，複数の実行スレッドが共有資源をアクセスするタイミングないし順序に依存する並行バグ一般は，本研究を含め競合解析によって検出できる可能性がある．実際，Yu らの報告 [4] によると，一般的な並行バグを 6 つのイディオムに分類し，それらの全てを競合解析に基づく検出法によって摘出できたとの実験結果も存在する．本研究の立場は，そのような競合解析に基づく検出法全般を高速化する並列検査方式を提供しようとするものである．

5 おわりに

本稿では，従来の動的競合解析を現代的なコア数の多いチップマルチプロセッサ上で実行した場合に問題となる検査時間の増大に焦点を当て，その原因を履歴操作の衝突と分析した上で，衝突を回避しつつ高速な検査を達成する並列検査方式を検討した．この方式には，インメモリ型の並列データ処理を活用しており，予備実験の範囲内では良好な性能を示した．会場での質疑応答に基づく考察により，提案方式の改善と限界についての分析が進むとともに今後の重要な課題が明確になった．

参考文献

[1] J-D. Choi, K. Lee, A. Loginov, R. O ’ Callahan, V. Sarkar, and M. Sridharan.: Efficient and precise datarace detection for multithreaded object-oriented programs. In *PLDI*, 2002.

[2] J. Dean and S. Ghemawat.: MapReduce: Simplified data processing on large clusters, In *OSDI*, 2004.

[3] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis.: Evaluating MapReduce for multi-core and multiprocessor systems. In *HPCA*, 2007.

[4] J. Yu , S. Narayanasamy, C. Pereira, and G. Pokam: Maple: A Coverage-Driven Testing Tool for Multithreaded Programs, In *OOPSLA*, 2012.