Vol.2018-SE-198 No.28
2018/3/10

ハイブリッド競合検査の負荷分散を考慮した並列化

櫻井 義孝[†1,a)]　荒堀 喜貴[b)]　権藤 克彦[c)]

概要：マルチスレッドプログラミングにおけるデータ競合は発見と再現が困難である．そのため自動でデータ競合を検出するツールが必要である．これまでの研究ではデータ競合の誤検出，検出漏れと検査時間のオーバーヘッドが問題になっていた．この問題を解決すべく，検査時間のオーバーヘッドを削減する手法として，検査を並列化する Parallel FastTrack が提案されている．しかし，Parallel FastTrack の方式では検出漏れが多く，また並列化においても検査スレッド間の負荷の偏りを解消する仕組みが存在しない．このため，特定の検査スレッドに負荷が集中した場合に十分に検査時間のオーバーヘッドを削減できない．そこで，本研究では高精度かつ高効率なオフライン競合検査の並列化手法を提案する．本研究の提案する手法は Parallel FastTrack とは異なり，ハイブリッド競合検査をベースにすることで検出漏れと誤検出を抑制する．また，検査時間のオーバーヘッドを減らすために競合検査を並列化する．更に，検査スレッド間の負荷の偏りを解消する並列競合検査方法を提案する．提案手法では，検査スレッドの管理するアドレス数を負荷と捉え複数のスレッド間で負荷を偏りを分散する．
Phoenix ベンチマークを用いた実験により，従来の単一スレッドによるハイブリッド競合検査と Parallel FastTrack 方式で検査を並列化した場合と比較して，提案手法による負荷分散を考慮した検査の並列化の効果を調査した．その結果，Phoenix を用いた行列演算プログラムにおいて提案手法は競合検査の時間オーバーヘッドを Parallel FastTrack 方式で分散したときと比較して 47%まで削減できた．

1. 導入

CPU の発達によりコア数が増加し、プログラムをマルチスレッドで実行することでプログラムの性能をあげるようになった．マルチスレッドプログラムはスレッドが並列に動作するためにプログラマのミスによるデータ競合が発生することがある．しかし，マルチスレッドプログラムにおけるデータ競合はスレッドのインターリービングにより実行の度に振る舞いが異なるのでデータ競合の再現と発見が困難である．そのため，データ競合を自動で検出するためのツールが必要である．

データ競合は 2 つ以上のスレッドが同時に同じメモリロケーションにアクセスし，同時に発生したアクセスのうち少なくとも 1 つのアクセスが書き込みアクセスである場合に発生する．過去の研究では静的解析を行うものや，動的解析を行うもの，更にはその両方を行うものが存在した．しかし，静的解析だけでは誤検出が多く動的解析を用いることが必要となる．

過去の研究には主に 2 つの競合検査の方法があった．1 つめはロックセットアルゴリズムである．これは，各メモリロケーションを保護しているロックに注目して競合検査を行い，メモリロケーションを保護するロックが無い場合に競合として扱う．2 つめは Happens-Before 関係 [1] を用いた競合検出である．これはアクセスの順序関係に注目して競合検査を行い，2 つのアクセスに順序関係が無い場合に競合として扱う．

ロックセットアルゴリズムを用いた競合検査 [2], [3] は，ロックでは保護されていないが明確な順序関係により競合を起こさない場合などに誤検出を起こす．一方で，Happens-Before を用いた競合検査 [1], [4] は誤検出は無いが，実行しているスレッドのインターリービングにおける競合しか検出することができず，異なるスレッドインターリービングにおいてのみ発生する競合を検出できない．そこで，この 2 つの検出方法を組み合わせることで誤検出を抑制する研究 [5] も存在する．

競合検査のオーバーヘッドを削減するために，アプリケーションと競合検査スレッドを分離し，かつ競合検査を複数スレッドで行うようにする研究も存在する．これは Parallel FastTrack[6] と呼ばれており，Happens-Before 関係を効率的に計算する競合検査方法である FastTrack を並列化したという研究である．Parallel FastTrack はオリジ

†1 現在，東京工業大学
Presently with Tokyo Institute of Technology
a) sakurai.yo@sde.cs.titech.ac.jp
b) arahori@cs.titech.ac.jp
c) gondow@cs.titech.ac.jp

ナルな FastTrack に対して 8 コアマシンで競合検査のオーバーヘッドにおいて 2.2 倍の性能が確認された．

Parallel FastTrack では各検査スレッドは定数 C と検査スレッド数 n を用いて $tid = (addr >> C)\ mod\ n$ でアドレスからそのアドレスを検査するスレッド tid を決定していた．これはメモリ領域を 2^C バイト毎に分割し，分割された各領域に対して検査スレッドが 1 つ割り当てられ，その検査スレッドがその領域のメモリ領域へのアクセスに対して競合検査を行う．

しかし，この方法で検査スレッドを決めた場合，アプリケーションプログラムによる特定のメモリ領域へのアクセスが多い場合にその領域を担当する検査スレッドの競合検査にかかる負荷が他の検査スレッドに対して大きくなる．実際にマルチスレッドライブラリを用いて行列計算を行うプログラムのメモリアクセス履歴を検査すると検査スレッドごとに，検査を担当しているアドレスの数は大きく異なっていた．しかし，並列検査にかかる時間は競合検査の終了がもっとも遅いスレッドに依存するのでスレッド間の負荷の偏りが大きい場合，検査中のスレッドと検査が終了しているスレッドが存在してしまい CPU を十分に活用できない．

そのため，この研究ではオフライン解析を行い，各検査スレッドの管理するアクセスの先頭アドレス数を負荷として捉えるた上で，これを均一化することでスレッド間の負荷の偏りを小さくし競合解析を高速化することを目的としている．

本研究の貢献は以下のとおりである．既存の研究における競合検査の並列化において，検査スレッド間に負荷の偏りが存在することを確認し，この負荷の偏りをオフライン解析で解消することを提案した．この提案を Intel Pin を用いて実装し，実験を行った結果 Phoenix ベンチマークにおいて最大で 53%の検査時間オーバーヘッドの削減を確認した．

2. 背景と動機

2.1 背景

この節では, ロックセットアルゴリズム [2], [3]・Happens-Before[1], [4]・ハイブリッド競合検査 [5]・Parallel FastTrack[6] について説明する．ロックセットアルゴリズムと Happens-Before は動的競合解析の一般的な方法である．ハイブリッド競合検査はロックセットアルゴリズムと Happens-Before を組み合わせた検査手法であり，本研究が提案する競合解析ではこれを用いる．Parallel FastTrack は過去に提案された Happens-Before ベースの競合解析である FastTrack[4] という競合検査手法が並列化された競合検査の手法である．

この節では，e はアクセスイベントを，$e.t$ はアクセスイベント e を実行したスレッドを，$e.L$ はアクセスイベント e を実行したときに実行したスレッドが持っているロックの集合（ロックセット）を，$e.a$ はアクセスイベント e のアクセスタイプ（$READ, WRITE$）を意味する．

2.1.1 ロックセットアルゴリズム

ロックセットアルゴリズムはロックに注目して競合を検出する手法である．具体的には各スレッドは 2 つのメモリアクセスが同じメモリロケーションにアクセスし，2 つのアクセスは異なるスレッドによって行われ，アクセス少なくとも一方が書き込みアクセスであり，2 つのアクセスが共通ロックを持たない場合に競合として扱う．

この検出手法は，ロックを用いないがデータ競合が起きないようになっている場合に誤検出が発生する．

2.1.2 Happens-Before 関係に基づく競合検査

Happens-Before 関係を用いた競合解析はアクセスの順序関係を観察し，2 つのアクセスに順序関係が無い場合にデータ競合が存在すると判定する．アクセスの順序関係（Happens-Before 関係）はメッセージイベントというアプリケーションプログラムにおけるイベントを介して 2 つのアクセスに順序関係が定められる．このメッセージイベントにはスレッドの Fork/Join やロックの Lock/Unlock が用いられる．Happens-Before 関係の計算には VectorClock[7] を用いて効率的に行われる．

Happens-Before 関係を用いた検出手法は，異なるスレッドのインターリービングにおいて発生する競合を検出することができない．

2.1.3 Hybrid 競合検査

Hybrid 競合検査はロックセットアルゴリズムと Happens-Before 関係を組み合わせることで，ロックセットアルゴリズムにおける誤検出を抑制するという検出手法である．ただし，Hybrid 競合検査においては Happens-Before に制限を付けてスレッドの Fork/Join のみを取得する．

2.1.4 競合検査の並列化

競合検査の並列化における最新の手法として Parallel FastTrack[6] がある．これは，Happens-Before 関係を用いて効率的に競合検査を行う手法である FastTrack[4] を並列化したものである．アプリケーションスレッドから検査スレッドを分離することで，アクセスが発生するたびに検査のためにアプリケーションスレッドを停止させることがなくなり，かつ検査を複数スレッドで行うことで検査時間のオーバーヘッドを削減している．ParallelFastTrack におけるアクセス分散はアドレス $addr$ は，$tid = (addr >> C)\ modn$(定数 C, スレッド数 n) で決められるスレッド tid によって検査される．

2.2 動機

Parallel FastTrack の方式で各検査スレッドに検査対象アドレスを割り振った場合，検査スレッドごとの検査にかかる負荷が偏る場合がある．実際にマルチスレッドライ

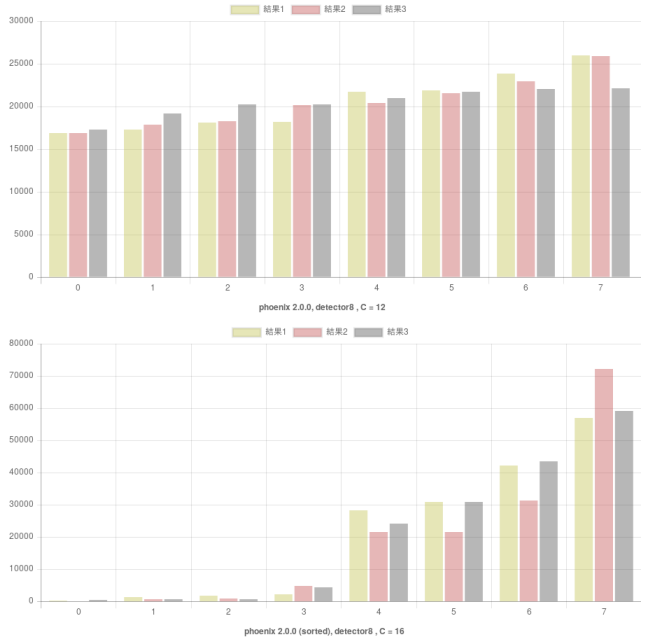

図 1 Parallel FastTrack 方式: 検査スレッドの管理するアドレス数

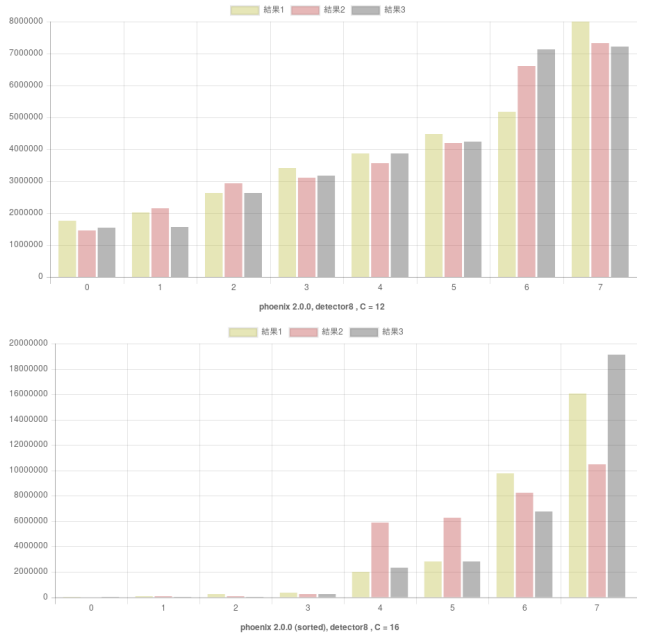

図 2 Parallel FastTrack 方式: 検査スレッドの検査するアクセス数

ブラリ Phoenix を用いた行列演算プログラムのアクセスを Parallel FastTrack の方式で各検査スレッドに検査対象のアドレスを割り当てた場合，各検査スレッドの検査するべきアドレス数と，アクセス数は**図 1**, **図 2** のような結果になる．図 1 は Parallel FastTrack 方式でアクセスを振り分けたときに各検査スレッドがいくつのアドレスを管理することになるかを表した図であり横軸は検査スレッドを，縦軸は管理するアドレス数を示している．図 2 は Parallel FastTrack 方式でアクセスを振り分けたときに各検査スレッドがいくつのアクセスを処理するかを表した図であり，横軸は検査スレッドを，縦軸は処理するアクセス数を示している．また，図 1, 図 2 の上段と下段はそれぞれ定数 C の値を 12, 16 にした場合にスレッド毎の検査アドレス数とアクセス数をソートした結果である．またそれぞれにおける結果 1,2,3 は 3 同じ検査対象プログラムに対して 3 回測定した結果である．この結果から，Parallel FastTrack の方式でアクセスを分散した場合は検査スレッド毎に負荷が偏る場合があることがわかる．

競合解析手法の性能は検査時間のオーバーヘッドが最も大きい検査スレッドに依存する．そのため，検査スレッド間の負荷の偏りが大き場合，負荷の大きい検査スレッドは検査を行っているが負荷の小さい検査スレッドは既に検査を終了している状態になる．そこで，検査スレッド間の負荷の偏りを解消することで負荷の大きい検査スレッドの検査時間のオーバーヘッドを小さくすることで競合解析全体の性能を向上させることができると考えられる．

2.3 取り組み

検査スレッド間の負荷を分散する簡単な方法として Parallel FastTrack で用いられている分散式の定数 C を小さくすることが考えられる．しかし，Song らの研究 [8] において，隣り合うメモリのベクトルクロックは一致することが多いため競合検査のメタデータを共有するという最適化が提案されており，この最適化を利用するためには共有するメタデータは同じ検査スレッドが管理する必要がある．そのため，近いアドレスは同じ検査スレッドによって管理されるようにするために，ある程度大きい領域の検査を同じ検査スレッドがするべきである．この理由により検査スレッド間の負荷の分散のために Parallel FastTrack の分散方式における定数 C を小さくするという手法を選択するべきでない．

そこで，この論文では Parallel FastTrack で使用されている分散方式の定数 C を小さくする以外の方法でスレッド間の負荷を分散する方法を提案する．

3. 提案手法

本研究では既存の競合検査並列化に対し検査精度を向上させるために，ハイブリッド競合検出を並列化する．また，既存の競合検査並列化に対し検査効率を向上させるために，競合検査を並列化する際に検査スレッド間に生じる負荷の偏りを解消する．並列化された検査スレッドの負荷の偏りの分散のために，本研究ではオフライン解析を行う．オフライン解析を行うので，分散は競合解析前に管理アドレスを計算することができるので検査スレッド間の負荷の偏りの解消は 1 度でよい．

オフライン解析を行うために，アプリケーションプログラムを 1 度実行し，**図 3** のようにアプリケーションプログラムのメモリアクセスを記録したトレースファイルを生成する．

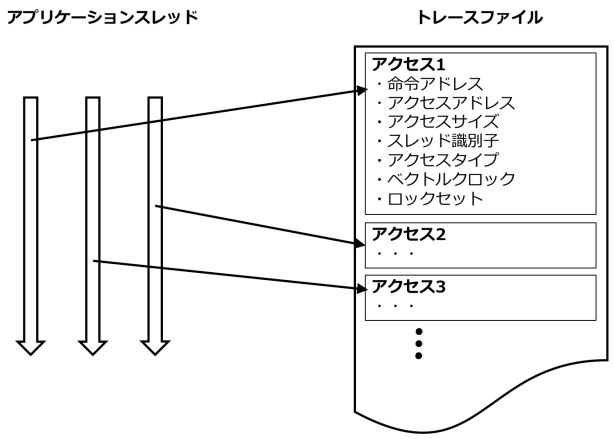

図 3 トレースファイルの生成

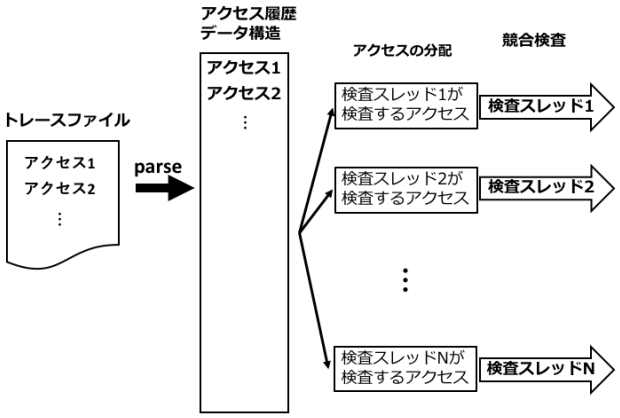

図 5 提案手法概要

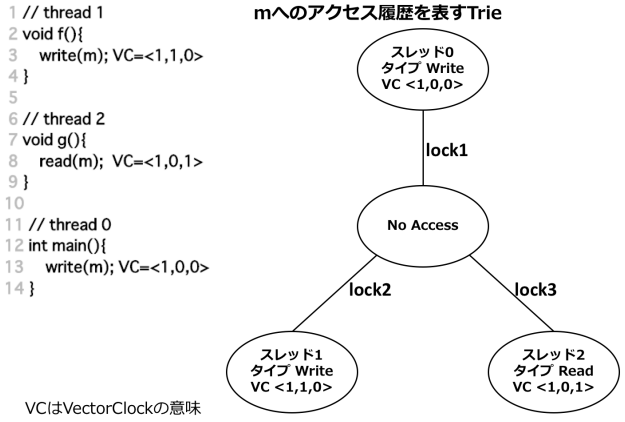

図 4 バイブリッド競合検出で用いる Trie

競合検査はアプリケーションスレッドの実行で生成されたトレースファイルに対してハイブリッド競合検査手法 [5] を検査スレッド間の負荷を分散するように並列化して適用する．ハイブリッド競合検査の適用には参考文献 [3] で提案されているような Trie を用いる．この Trie は**図 4** のようなハイブリッド競合検査に必要なデータとして，辺のラベルとしてロック識別子を使用し，頂点にはアクセスしたスレッドの識別子・アクセスタイプ・アクセス時の VectorClock が記録される．

ハイブリッド競合検査に用いる VectorClock は Choi が提案したハイブリット競合検査 [5] で使われているものではなく，ベクトルクロックの比較と更新に単一要素ではなく全要素を用いるような VectorClock を使用する．これは Hybrid Dynamic Race Detection で使用されるような単一要素で比較する VectorClock を用いた場合，実際には順序関係 (Happens-Before 関係) が存在する 2 つのアクセスに対する順序関係が消えてしまいアクセスの順序関係を見逃してしまうので検出精度の向上のためには全要素についてのベクトルクロックの比較，更新が必要になるからである．

この提案手法では競合検査を並列に行う際に検査スレッド間のアクセス負荷の偏りを減らすことを目的とする．検査スレッド間の負荷の偏りを解消するために，**図 5** のようにまずトレースファイルをパースした後に全てのアクセスを走査しアクセスを分配してから，分配されたアクセスを検査スレッドが検査していく．トレースファイルのパースは 1 スレッドで行い，以降ではパースされたアクセス履歴について考える．

3.1 アクセス履歴をアドレス毎に分ける

アクセス履歴からアクセスを分配する段階も並列に行う．アクセスを分配するためのスレッド数は検査スレッド数と関係は無いが，本研究では検査スレッド数と同じにする．このアクセス履歴からアクセスを分配するスレッドを分配スレッドと呼ぶことにする．

アクセスを分配するために，アクセス履歴を分配スレッド数 N で分割し，それぞれ分配のスレッドの担当区域にあるアクセスをアドレス毎に分割する．検査するアドレスは対応するアクセス履歴格納バッファを分配スレッド数である N 個保持する．tid 番目の分配スレッドは担当するトレース領域でアドレス $m1$ へのアクセスを見つけたら，アドレス $m1$ に対応する N 個のアクセス履歴格納バッファのうち tid 番目のバッファにアクセスを格納する．

図 6 は分配スレッド 2 がアドレス $m1$ へのアクセスを検出し，アドレス $m1$ の持つアクセス履歴格納バッファのうち分配スレッド 2 に対応するバッファにアクセスを保存する例である．この方法を全てのアクセス履歴に対して行うことで，各アドレスに対応するアクセス履歴がそのアドレスに対応するアクセス履歴格納バッファに保存される．アドレス $m1$ に対する検査は対応するアクセス履歴格納バッファ N 個を前から順番にアクセスしていくことで，アドレス $m1$ をアクセスの先頭アドレスにもつアクセス履歴はトレースファイルにおける順序が保存される．

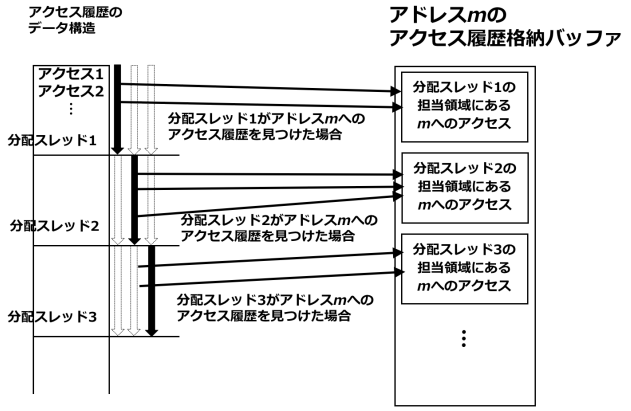

図 6 トレースファイルからアクセスを分配

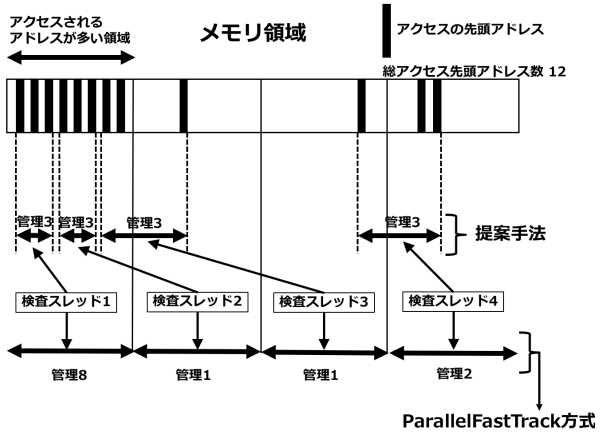

図 7 提案手法の分散方法と ParallelFastTrack 方式の分散方法

3.2 負荷の偏りを解消するアクセス履歴の分配

上のようにアクセスをアドレスごとに分けることで，アクセス履歴からアクセスの先頭アドレスが m であるアクセスのリストが生成される．本研究では各検査スレッドの検査管理するアクセスの先頭アドレスの数を負荷とする．この負荷を分散するために，アクセス履歴からアクセス履歴格納バッファにアクセスを保存する際に，同時にアクセス履歴の全てのアクセスの先頭アドレスを保存することで，全アクセスの先頭アドレスの集合を計算する．本研究では，この実際にアクセスがあったアクセス先頭アドレスの数を基準にアクセスを分散する．Paralell FastTrack 方式ではアクセスの有無に関わらず，領域に対して担当スレッドを決定していた．しかし，**図 7** のように特定の検査スレッドの担当領域に実際にアクセスされるメモリアドレスが偏っていた場合に，Parallel FastTrack 方式で割り当てた場合に検査スレッド 1 の負荷が大きいのに対して，本研究の提案手法ではアクセスの先頭アドレスの数で分散されるため全ての検査スレッドが同じ数のアクセス先頭アドレスを管理することになり検査スレッド間の負荷の偏りが解消される．

また，本研究の提案手法では各検査スレッドは近いアドレスを管理することになる．実際に図 7 では検査スレッド 1 は前から 3 つの連続したアクセスアドレスを管理している．このように近いアドレスを同じ検査スレッドが管理することで，近いメモリ領域における競合検査に必要なデータを共有するという最適化 [8] を適用することも可能になる．ただし本研究ではこのような最適化は行っていない．

3.3 負荷の捉え方

本研究のスレッド間の負荷の偏りの解消方法は各検査スレッドが管理するアクセスの先頭アドレス数を負荷と捉え，アクセスの先頭アドレス数が平等になるように分配した．しかし，負荷の捉え方によっては異なる負荷の分散方法がある．例えば，各検査スレッドが管理するアクセスの先頭アドレス数を負荷として捉えるのではなく，全てのアクセスのアドレス数や，検査するアクセス数を負荷として捉えることで別の基準で負荷を分散することもできる．本研究では各検査スレッドがアクセスの先頭アドレスを負荷の基準として扱う．これは先頭アドレスの数で負荷の偏りを解消することで，最も簡単にかつ負荷の偏りの解消のコストが比較的小さいと予想したからである．

4. 実装

4.1 トレースファイルを生成する Pintool

トレースファイルの生成には Intel の Pin[9] という動的バイナリ計装フレームワークを使用する．これは実行ファイルに対して，動的に解析を行うことができるツールである．このフレームワークを用いることで指定した関数の直前直後に命令を挿入，指定した関数を別の関数で置換，メモリへのアクセスの取得などを行うことができる．

本研究ではこのフレームワークを用いて，ロックの取得と解放に関わる関数を置換することでロックセットを計算し，スレッドの生成と終了に関わる関数を置換することで VectorClock を計算した上で，アプリケーションプログラムの全てのメモリアクセスの直前に競合検査に必要な情報をトレースファイルに記録する Pin tool を作成した．

この Pin tool はスレッド Fork/Join とロックの取得/解放関数におけるアクセスはトレースしない．

また，スタック領域における変数はスレッドローカルな変数であるためスレッド間で共有されることが珍しい．そのため，スタック領域へのアクセスは検査対象外にした．

4.2 free 関数の呼び出し

動的解析におけるテクニックとして free 関数がコールされたときにコールされた領域を管理するデータを破棄するというものがある (文献 [11], [12])．しかし，データ構造を破棄する操作はこれまでのアクセス履歴を全て解放する

ことになるので操作の時間オーバーヘッドが大きい．そこで，アクセストレース時に検査対象のプログラムにおける free 操作を実行しないように計装を行った．

5. 評価

5.1 実験環境

実験環境は全て 4 コア Intel(R) Core(TM) i5-7200U CPU 2.5GHz 上で ubuntu17.04，RAM8GB 上で行う．実験に使うサンプルとして phoenix-2.0.0[10] ライブラリを使用したマルチスレッド行列演算プログラムを対象とする．行列のサイズは 10×10 の行列と 100×100 の行列を使用する．

5.2 実験目的

この実験は，提案手法による検査スレッド間の負荷を分散することで，検査スレッド間の負荷がどの程度分散され，競合検査の速度についてどの程度向上が見込めるかを評価することを目的とする．比較対象として Parallel FastTrack と同じ方法 ($(addr >> C)\ mod\ n$ で分散させる方法) で負荷を分散させた場合を用いる．また，検査時間の比較については Parallel FastTrack 方式で分散させた場合と単一スレッドで競合検査を行った場合を用いる．本研究では，提案手法による負荷の偏りを解消することによる，検査の効率化の程度を調べるために，トレースファイルのパースと分散自体にかかる時間は計測対象に含めず，検査自体にかかる時間のみを比較する．

5.3 実験結果

5.3.1 管理アドレスの分散

提案手法で負荷を分散させた場合に，Parallel FastTrack 方式を用いた場合に比べて管理するアドレスの数の観点でどの程度負荷が分散されているかを確認する．

図 8, **図 9** はそれぞれ 10×10 の行列に対して Parallel FastTrack 方式と提案手法の方式でアドレスを分配した結果，各検査スレッドが管理しているアドレスの総数である．また，**図 10**, **図 11** もそれぞれ 100×100 の行列に対する Parallel FastTrack 方式と提案手法の方式でアドレスを分配した結果である．

負荷の偏りの指標として，管理するアドレスの最も少ない検査スレッドに対する管理するアドレスの最も多い検査スレッドの割合を考える．10×10 の行列に対しては，Parallel FastTrack 方式では最も管理アドレス数が少ない検査スレッドが管理するアドレス数は 485 個であるのに対して，最も管理アドレス数が多い検査スレッドは 23847 個のアドレスを管理している．一方で提案手法による負荷の分散を行った場合，管理アドレスが最も少ない検査スレッドは 1504 個のアドレスを管理しており，それに対して最も多い検査スレッドは 9623 個のアドレスを管理していた．結果として Parallel FastTrack の方式で分散した場合は上記の割合が 49 倍であるのに対して，提案手法で負荷を分散した場合上記の割合が 6 倍になった．また、100×100 の行列に対して同じように負荷の偏りを計算すると，Parallel FastTrack 方式で分散した場合は管理数最小検査スレッドは 206 個のアドレスを管理している一方で管理数最多検査

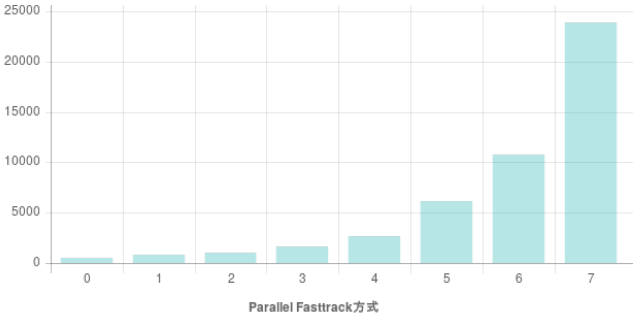

図 8 10×10 の行列計算のアクセスを Parallel FastTrack 方式で分配した場合

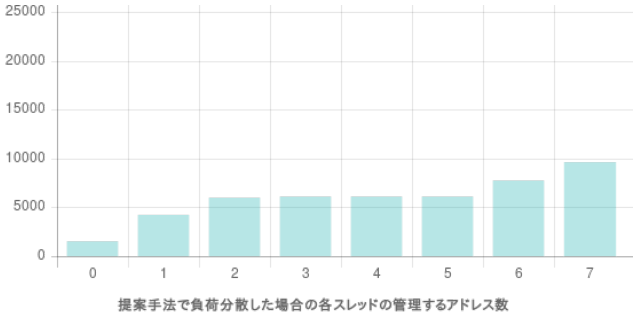

図 9 10×10 の行列計算のアクセスを提案手法で分配した場合

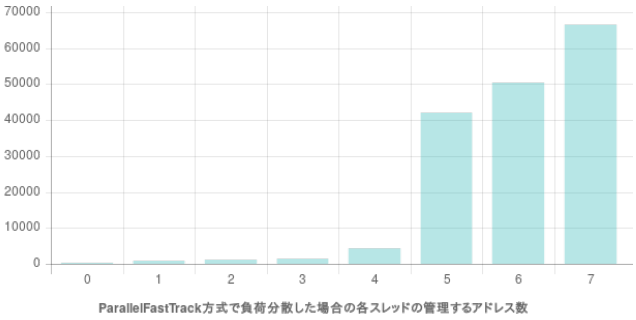

図 10 100×100 の行列計算のアクセスを Parallel FastTrack 方式で分配した場合

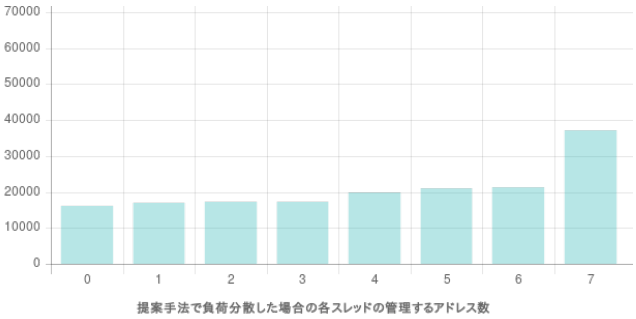

図 11 100×100 の行列計算のアクセスを提案手法で分配した場合

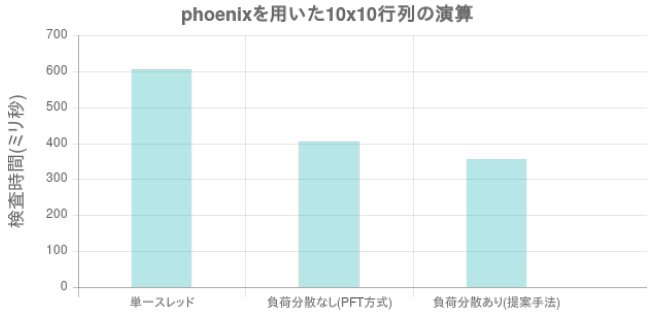

図 12 10 × 10 の行列計算の検査にかかる時間

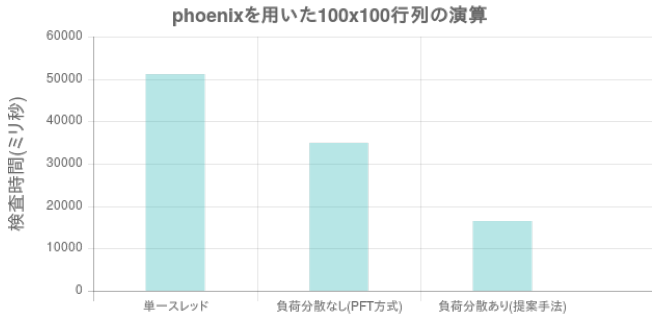

図 13 100 × 100 の行列計算の検査にかかる時間

スレッドは 66455 個のアドレスを管理しており 322.6 倍であったのに対して，提案手法で負荷の分散を行った場合は管理数最小検査スレッドは 16170 個のアドレスを管理しているのに対して管理数最多検査スレッドは 37182 個のアドレスを管理しているため負荷の偏りは 2.3 倍になっている．

これらの結果は Parallel FastTrack 方式で分散した場合と提案手法の方式で分散した場合で管理するアドレスの合計数は一致していた．

図 9, 図 11 において提案手法で各検査スレッドが同じ個数のアドレスを管理しようとしたにもかかわらず，負荷の偏りが残っている．これは提案手法が分散し均一化しているのはアクセスの先頭アドレスであるが，アクセスはアクセス毎にアクセスサイズが異なるのが原因であると考えられる．Pin はアクセスのサイズとして 1,2,4,8 バイトを観測するので，提案手法のようにアクセスの先頭アドレスを均一化するように負荷分散した場合，最大で 8 倍の負荷の偏りが残る．

5.3.2 検査速度

提案手法を用いて負荷を分散した上で競合検査を行った場合と，Parallel FastTrack 方式を用いて競合検査を行った場合, 単一スレッドで競合検査を行った場合の検査時間を比較する．この結果の時間には純粋に競合解析を行っている時間だけを計測しており, アクセスを分散するのにかかる時間は含めていない．また，この結果は同じトレースファイルに対して 10 回計測し，その結果から最も時間のかかった結果と最も時間のかからなかった結果を除いた上で平均を求めた結果である．

図 12, **図 13** はそれぞれサイズが 10 × 10,100 × 100 の行列の掛け算を行ったアクセストレースの競合検査を行った場合の時間である．10 × 10 のサイズの行列に対して負荷を分散した場合，提案手法で分散した場合に競合解析にかかる時間は Parallel FastTrack 方式で分散した場合に比べて 88.4%, シングルスレッドで解析した場合の 58.9%になる．また，100 × 100 のサイズの行列にたして負荷を分散した場合，提案手法は Parallel FastTrack 方式で分散した場合に比べて 47.0%, シングルスレッドで解析した場合の 32.1%の時間になる．

100 × 100 のサイズの行列計算のアクセストレースに対する検査について，図 10, 図 11 から Parallel FastTrack 方式の場合に対して提案手法の最大管理検査スレッドの管理するアドレス数が約半分になっていることがわかる．一方で図 12, 図 13 より検査にかかる時間も約半分になっていることがわかり，管理アドレス数が競合解析時間に影響を与えていると考えることができる．

10 × 10 のサイズの行列計算のアクセストレースに対する検査が，図 8, 図 9 から管理アドレスの最も多い検査スレッドの管理するアドレス数は提案手法では半分以下になっているが，検査にかかる時間は 88.4%程度にしかなっていない．この原因は 2 つ考えられる．1 つめはレイテンシなどの競合検査とは直接関係がない部分の影響が大きい可能性がある．2 つめは検査するアクセス数の偏りを解消できていない可能性がある．これは提案手法がアクセスの先頭アドレス数を基準に負荷を分散しているが，特定のアドレスに対してアクセスが集中していた場合，アクセス数の観点から見た負荷の偏りが解消できていない可能性がある．

6. 関連研究

Eraser[2] はロックセットアルゴリズムを用いた競合検出器である．Eraser はメモリがロックによって保護されているというロック規律を前提として競合を検査する．一方で，Fork や Join によるアクセスの順序関係 (Happens-Before 関係) を考慮しない．しかし，マルチスレッドプログラムは Eraser の提案するロック規律を侵害しているが，アクセスの明示的な順序関係により競合が発生しないようになっていることが多いので，誤検出が発生してしまう．検査はメモリアクセスの度に行うが，アプリケーションスレッドと検査スレッドが分離されていないので検査の度にアプリケーションを停止させてしまう．

Choi は静的解析と動的解析を組み合わせて競合検出の精度を上げている [3]．join 操作を擬似ロックを用いてモデル化し,start を所有権を用いてモデル化することで誤検出を減らしている．また過去のアクセス履歴を Trie を用いて表現することで効率的にアクセス履歴を表現し，過去のアクセスの検索を可能にしている．検査はメモリアクセスの度に行うが，アプリケーションスレッドと検査スレッド

が分離されていないので検査の度にアプリケーションを停止させてしまう．

ハイブリッド競合検出 [5] はロックセットアルゴリズムをベースにした上で，ロックセットアルゴリズムの誤検出を Happens-Before 関係を用いて抑制する．この Happens-Before 関係は純粋な Happens-Before 関係ではなくスレッドの Fork と Join だけをメッセージイベントとして捉えた制限付き Happens-Before 関係である．ただし，この Happens-Before 関係を計算する VectorClock は近似されたものであるので高速ではあるが十分に誤検出の抑制が行われない．検査は 1 スレッドで行われる．

FastTrack[4] は Happens-Before 関係を用いて，2 つのアクセスイベントに順序関係が存在しない場合に競合として扱う．この順序関係のトリガーにはスレッドの Fork・Join の他にロックの取得・解放をイベントとして扱う．FastTrack で検出されたデータ競合には誤検出が含まれない．しかし，検出結果には異なるスレッドのインターリービングで発生するデータ競合が含まれない．検査はメモリアクセスの度に行うが，アプリケーションスレッドと検査スレッドが分離されていないので検査の度にアプリケーションを停止させてしまう．

FastTrack は 1 スレッドで検査を行っており，アプリケーションスレッドと検査スレッドが同じスレッドだった．そのため，アクセスに対して検査を行う度にアプリケーションの実行が中断されてしまっていた．それに対し，Parallel FastTrack[6] はアプリケーションスレッドと検査スレッドを分離し，検査を並列に動かすことで検査がアプリケーションの実行を妨げないようにし，高速化を図った．検出される競合は FastTrack と一致するので誤検出はないが検出漏れが存在する．

7. 結論

データ競合の動的検査を並列化する場合に Parallel FastTrack の方式では検査スレッド間に負荷の偏りが存在し，特定の検査スレッドに負荷が集中した場合に十分に検査時間のオーバーヘッドを削減できないことを発見した．この負荷の偏りは競合検査のオーバーヘッドを大きくしてしまう．そこで本研究では，ハイブリッド競合検査を検査スレッド間の負荷の解消をするように並列化する，高精度かつ高性能なオフライン競合検査の手法を提案した．Phoenix ベンチマークを用いた実験により，従来の Parallel FastTrack が提案する分散方法で負荷を分散した場合にくらべて最大で 2 倍程度の検査の高速化が確認された．

謝辞 本研究は JSPS 科研費 16K00093「並行ソフトウェアの正確かつ高速な実行時検査」の助成を受けたものです．

参考文献

[1] Leslie Lamport: Time,Clocks, and the Ordering of Events in a Distributed System. Commun. ACM,Vol21,No.7,558-565,1978
[2] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson: Eraser: A Dynamic Data Race Detector for Multithreaded Programs. TOCS,Vol15,No.4,391-411,1997
[3] Jong-Deok Choi,Keunwoo Lee,Alexey Loginov,Robert O'Callahan,Vivek Sarkar,Manu Sridharan: Efficient and Precise Datarace Detection for Multithreaded Object-Oriented Programs. PLDI'02,285-297
[4] Cormac Flanagan , Stephen N. Freund: FastTrack: Efficient and Precise Dynamic Race Detection. PLDI'09,121-133
[5] Robert O' Callahan and Jong-Deok Choi: Hybrid Dynamic Data Race Detection. PPoPP'03,167-178
[6] Young Wn Song and Yann-Hang Lee: A Parallel FastTrack Data Race Detector on Multi-core Systems. IPDPS2017,387-396
[7] Friedemann Mattern: Virtual Time and Global State of Distributed Systems. In Proceedings of the International Workshop on Parallel and Distributed Algorithms(1988)
[8] Young Wn Song and Yann-Hang Lee: Efficient Data Race Detection for C/C++ Programs Using Dynamic Granularity. IPDPS'14,697-688
[9] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. PLDI, Chicago, IL, June 2005, pp. 190-200.
[10] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski,Christos Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. HPCA, Phoenix, AZ, February 2007
[11] Frank Ch.Eigler: Mudflap: Pointer User Checking for C/C++. GCC Developers Summit2003,57-70
[12] Olatunji Ruwase: A Practical Dynamic Buffer Overflow Detector. NDSS Symposium2004.