平成27年度　修士論文

GPGPUを用いた並列テキストマッチングについての研究

電気通信大学 大学院情報システム学研究科
情報システム基盤学専攻

1453003　岩崎　一輝

指導教員
本多 弘樹 教授
三輪　忍　准教授
古賀　久志 准教授

平成28年2月10日

目 次

第1章

序論

1.1 研究背景

近年ではネットワーク上の有害なパケットの種類と内容が増加しているため，不正アクセスを検知するためのネットワーク侵入検知システム（Intrusion Dtection System，略称 IDS）は，今後更にリアルタイムで多数のパターンを検出する必要性が高まることが予想される．ネットワーク層におけるファイアウォールを越えてくる有害なパケットに対して，IDS は有害なパケットを検出し，不正アクセスからシステムを保護する必要がある．様々な有害なパケットの増加に伴い，IDS が不正アクセスを検知するために使用されるルールの数も同様に増加している．

IDS では，入力されてくるパケットに対して，有害なパターンを検出するために文字列照合プログラムが実行されている．中でも一般的なものが，決定性有限オートマトン（Deterministic Finite Automaton，略称 DFA）ベースのプログラムで，あらかじめ定められたシグネチャと呼ばれる「不正アクセスと判断される文字列パターン」からステートマシンを作成し，入力されるパケットがステートマシン内の状態間を遷移することによってマッチングを行う．この DFA ベースの文字列照合プログラムは，メモリ要件はステートマシンの作成する状態数と入力パケットの大きさに比例するものの，時間計算量が少なく，規則性と拡張性の両方の面で優れている [1].

DFA ベースの文字列マッチングで最も一般的なものとして，Aho-Corasick アルゴリズムが知られている [2]．そして，Aho-Corasick アルゴリズムを GPU 上での並列テキストマッチングのために改良されたアルゴリズムとして Parallel Failureless Aho-Corasick（PFAC）も提案されている [3]．PFAC は GPU の高い並列度を活か

し，多数のスレッドを使用して入力文字列の文字一つ一つにスレッドを割り当てる．入力文字列の検査が文字単位で並列に行われるため GPU の性能に依存する部分が大きいが，膨大なパケットとパターンに対してテキストマッチングを行う際は Aho-Corasick と比較して大幅に計算時間を短縮できる．

また，PFAC はマルチ GPU による実装も行われている [4]．従来の手法では入力文字列を各 GPU に分割する事で 1 つの GPU が検査する文字列の量を減らし，更なる高速化を実現している．この入力文字列を分割する手法とは別に，ターゲットパターンを分割する事で高速化する方法も考えられるが，従来の研究では PFAC でパターンを分割する手法は報告されていない．

1.2　研究目的

本論文では，大量の入力文字列と複数パターンとのテキストマッチングの更なる高速化を目指す．具体的には，マルチ GPU 環境下においてパターンを各 GPU に分割する手法を提案し，更に，より良いパターンの分割手法も提案する．提案手法を PFAC アルゴリズムに実装した際の処理性能を評価するために，PFAC の従来のマルチ GPU 化手法である入力文字列を分割する手法と比較を行い，提案手法の有効性を示す．

1.3　本論文の構成

以下に本論文の構成を述べる．

第 2 章

本研究が対象としているテキストマッチングのアルゴリズムである，PFAC について述べる．PFAC は Aho-Corasick を改造したものであるため，Aho-Corasick についても述べる．またテキストマッチングの関連研究についても述べる．

第 3 章

本研究の提案手法であるパターン分割について述べ，従来手法である入力分割との違いや本研究でのパターンの分割手法について述べる．

第4章

本研究で提案した手法を様々なデータセットに対して評価実験を行い，その有効性を示す．

第5章

関連研究として，侵入検知システムにおけるパターンマッチングの高速化について参考にした論文を紹介する．また，パターン分割が用いられている従来の研究についても述べる．

第6章

本研究のまとめと考察及び今後の課題について述べる．

第2章

テキストマッチング

本章では，本研究で用いているテキストマッチングアルゴリズムである Parallel Failureless Aho-Corasick(PFAC）アルゴリズムについて述べる．PFAC は，テキストマッチングに用いられるアルゴリズムとして広く知られている Aho-Corasick アルゴリズムを GPU 上での並列テキストマッチングのために改良したアルゴリズムである．そのため，まずは Aho-Corasick 法について述べる．

2.1 Aho-Corasick

任意の文字列中から複数のパターンに一致する文字列を高速検索するアルゴリズムとして，Aho-Corasick 法が広く知られている．Aho-Corasick 法は Alfred V. Aho 氏, Margaret J. Corasick 氏の 1975 年の論文 [2] で提案されたもので，複数のパターン文字列からなる集合と長い文字列が与えられた際に，長い文字列に対してマッチするパターン文字列全てを求める事のできるアルゴリズムである．

Aho-Corasick 法ではあらかじめ与えられたパターン集合からパターンマッチングを行うステートマシンと呼ばれる DFA を構築する．このステートマシンに対して入力文字列を一回走査するだけで複数のパターンを同時に検索できるため，入力文字列に対して線形な計算時間を実現する高速な逐次実行のアルゴリズムである．侵入検知システム以外にも細胞生物学や遺伝子配列の検索など幅広い分野で用いられるアルゴリズムで，多くの関連研究が存在する [5,6]．

図 2–1: Aho-Corasick ステートマシン

Aho-Corasick 法で構成されるステートマシンの例を図 2-1 に示す．図 2-1 は，4 つのパターン ”AB”，”ABG”，”BEDE”，”EF ”から構成されるステートマシンを表しており，二重線で表された状態 1，2，3，4 は，入力文字列とパターンが一致した状態を表す．また，点線で表された矢印は遷移に失敗した時のバックトラック先である．入力文字列が”ABEDEDABG” だった場合，初期状態 5 から文字 A によって状態 6 へ遷移し，文字 B によって状態 6 から状態 1 へ遷移する．状態 1 においてパターン”AB” とマッチングするが，次の入力文字”E” と次の遷移先”G” が一致しない．通常，不一致が起きた場合には状態 5（初期状態）から走査し直すのだが，同一の文字から始まる状態がある場合には直接その状態へ遷移し，状態 5 に戻って状態 7 まで走査する手間をショートカットする事ができる．つまり，点線矢印は不一致時のバックトラック先を表すポインタである．これによってシングルスレッド処理でも効率的なテキストマッチングが可能である．

Aho-Corasick アルゴリズムは複数の文字列を一度の走査で高速に検索が可能な点から，このアルゴリズムを用いた研究は数多く存在する．通常，Aho-Corasick 法は逐次計算用のアルゴリズムとして用いられるが，OpenMP や CUDA を用いて並列化を行った先行研究も存在する．多くの先行研究では Aho-Corasick 法を並列化する際には，入力文字列を等間隔に分割し，各スレッドが分割された部分入力文字列に対する検索を並列に行う方法を取っている [5]．図 2-2 に Aho-Corasick をスレッド毎に並列化した時の例を示す．

図 2–2: Aho-Corasick の並列化

Aho-Corasick 法を並列化する際には図 2-2 のように入力文字列を等間隔に分割して各スレッドが割り当てられた入力文字列に対して並列にマッチング処理をする方法が一般的である．しかし，図 2-2(a) に示すように 1 つのスレッドに対して 4 つの文字ずつ割り当てて処理を行うような並列化をした場合，スレッド 3 とスレッド 4 の境界をまたぐ文字列 ”AB ” を検出できなくなり，正しい結果を得られなくなってしまう．

そこで，図 2-2(b) のように各スレッドがスレッド間の境界を越えてオーバーラップスキャンをする手法が取られている．この手法により図 2-2(b) において，スレッド 3 が文字列 ”AB ” を検出できるようになる．各スレッドが ”最長パターン長-1 文字 ”のオーバーラップ演算をする事で正しいマッチング結果を得られるようになり，こうして Aho-Corasick 法を並列化しているが，重複計算によるオーバーヘッドも増大してしまう．

Aho-Corasick は逐次計算においては，ステートマシンによって一度に効率よくパターンマッチングを行う事が可能である．しかし，ステートマシンの特性上，アルゴリズム自体の並列化は困難であるため，テキストマッチング処理を並列化する際には他のアルゴリズムが用いられる事が多い．

2.2 Parallel Failureless Aho-Corasick

図 2–3: PFAC ステートマシン

PFAC アルゴリズムは，Aho-Corasick をベースとして，GPU で高速に並列テキストマッチングを行うために開発されたアルゴリズムである．提案者らの手によってこのアルゴリズムを CUDA 実装した PFAC ライブラリがオープンソースで公開されている [4].

Aho-Corasick と同様に，検出したいパターンのリストからステートマシンを構築し，入力文字列がステートマシンの状態間を走査する．前項の例と同様に，“AB,ABG,BEDE,EF” の 4 つのパターンから構築されたステートマシンを図 2-3 に示す．図 2-1 の Aho-Corasick 法のステートマシンと比較すると不一致時のバックトラック先を表すポインタである点線矢印が存在しない．PFAC では入力文字列が状態間の遷移に失敗した時点でバックトラックを行わずにその場でスレッドを終了する．また，PFAC では入力文字列中の全ての文字をパターンのスタート文字とみなして走査を行うため，一般的には多くのスレッドが早い段階で遷移に失敗する．多くのスレッドが早期に計算を終了するために PFAC では効率的にテキストマッチングを行うことができる [3].

図 2–4: PFAC のバイト毎のスレッド割り当て

PFAC はスレッドの開始位置から始まる任意のパターンを識別するために入力文字列の各バイトに個々のスレッドを作成する．図 2-4 に示すように，PFAC によって作成されるスレッドは入力文字列の文字一つ一つに割り当てられる．

例えば図 2-4 のように入力文字列が ”ABEDEDABG ” という文字列を含んでいた場合には PFAC は【$t_0, t_1, \cdots, t_8$】のスレッド ID を持つ 9 つのスレッドを呼び出す．この時スレッド t_{j} は，位置 j で始まる部分をチェックする．つまりこの場合 t_1 は”BEDE…” と文字列をチェックしていき，パターン 3”BEDE” を見つけ，マッチングする．すべてのスレッドは，初期状態から開始する．その次の状態が，現在の状態と入力文字には有効な遷移がないことを示すトラップ状態であれば，PFAC ではスレッドが終了する．

入力文字列の各文字に個々のスレッドを割り当て，遷移に失敗したスレッドから処理を終了する．PFAC の各スレッドがそれぞれ唯一の開始位置から始まるため，パターンを識別する際に非常に効率的に，計算を行う事が出来る．図 2-5 に示すようにスレッド t_3 が初期状態で文字”D” をとるが，初期状態では”D” には有効な遷移が存在しないので，スレッド t_3 は早くも初期状態で終了する．スレッド t_5, t_7, t_8 も同様である．一方，スレッド t_6 は”AB”,”ABG” の 2 つのパターンに一致することができる．この場合”AB” は”ABG” の接頭辞であるため， PFAC は最長のパターンを検出するために，スレッドを終了せずに走査を続けることになり，次の遷移で”ABG” を検出する事になる．

第 4 章で詳述するが，PFAC を含む DFA ベースの文字列マッチングの処理では，この状態遷移を行う回数が多いほど計算量が大きくなり，計算時間も長くなる．このため，遷移回数が少ないマッチング処理を行う事が高速化につながり，多くのスレッドで遷移が早期に終了する PFAC は高速なマッチング処理を実現している．

図 2–5: 入力文字列 ”ABEDEDABG ”への 9 スレッド並列処理

図 2–6: GPU での PFAC 実行モデル

図 2-6 に示すように，PFAC では 1 つのスレッドブロックが 1024byte(1024 文字) の処理を行い，最大で 256 のスレッドが 4 回ずつマッチングを行っている．また，PFAC が処理できる最長のパターン長は 512 である [7,8].

図 2–7: PFAC の処理ステップ

PFAC 実行時の処理ステップを図 2-7 に示す [11]．ホスト側でパターンファイルからステートマシンを構築し，デバイスへコピーするため，パターンファイルが膨大なサイズの時はステートマシンの構築に時間を取られてしまう場合もある．

また，ステップ 10 で CPU 側でも PFAC を実行しているが，これは GPU 側で PFAC を実行している間に CPU 側でもコピー元のステートマシンと入力文字列を用いて同様に PFAC を実行している．これはステップ 13 で CPU 側と GPU 側のマッチング結果と照合し，ミスマッチがないか確認するために行っている．しかし，膨大な入力データとパターンファイルのマッチング処理を行う場合には，ホスト・デバイス間通信とデバイス・ホスト間通信の時間を含めても GPU 側での PFAC 実行の方が早期に結果を返すため，CPU 側とのマッチング結果の照合が行われないまま GPU での実行結果のみが出力される．

2.3　その他の並列化手法

図 2–8: 複数 GPU 環境下での入力分割手法

現在公開されている PFAC ライブラリでは，複数の GPU を用いて PFAC を実行する事も出来る．用いられている並列化手法は，CPU 側で OpenMP により PFAC を実行する GPU の台数と等しい数のスレッドを生成し，入力文字列を各 GPU に分割している．この入力文字列を分割する並列化手法（以降は**入力分割**と呼ぶ.）では，検索したい文字列パターンのリストは各 GPU が同様のものを持つ．

PFAC において 4 つの GPU にマルチ GPU 実装する際に入力分割を適用した例を図 2-8 に示す．

この手法では各 GPU が同じパターンセットを持ち，同じステートマシンを構築し，それぞれに割り当てられた入力文字列に対してマッチング処理を行う．この手法を用いると，各 GPU が処理する入力文字列の数は 4 分の 1 になるので大幅な高速化が可能になるが，入力データを分割すると Aho-Corasick 法を並列化する時と同様に入力文字列を分割した際の境界面でパターンを発見できなくなる恐れがある．そのため，図 2-2(b) のように ”最長パターン長ー 1 文字 ” のオーバーラップ演算で境界面での処理を行う事により，この問題を解決している．

第3章

提案手法

現在，マルチ GPU 環境下で PFAC を実行する際の並列化手法は入力分割が用いられている．また，従来の研究では，入力文字列ではなくパターンリストを分割する並列化 (パターン分割) を行い，マッチング処理を高速化する試みも研究されている [9, 10]．

しかし，マルチ GPU 環境下で PFAC を実行する際にパターン分割を行う先行研究は存在していない．そこで，本研究では各 GPU に対してより良いパターン分割を行う手法を提案する．

3.1 手法の着眼点

複数台の GPU を活用できる環境下で PFAC を実行する際に用いられる際の並列化手法として，従来手法では入力分割による並列化が行われているが，本研究では入力文字列は各 GPU に分割せずに転送し，検索するパターンのリストを各 GPU に分割する並列化手法を提案する．

パターン分割の概要の例として，PFAC を 4 つの GPU にマルチ GPU 実装する際にパターン分割を適用した時の概要を図 3-1 に示す．この手法では各 GPU が検索するパターンリストが違うのでそれぞれ異なるステートマシンを構築する．この時，1 つの GPU が検索する文字列パターンは入力分割と比較すると単純計算で 4 分の 1 であるが，入力分割が各 GPU に入力文字列が 4 つに分割されるのに対し，パターン分割では各 GPU が同じサイズの入力文字列を計算する必要がある．

図 3–1: マルチ GPU 環境下でのパターン分割手法

このパターン分割手法では，入力文字列を分割しないので境界面が発生する事によるパターンの検知漏れを防ぐためのオーバーラップ演算が発生しない事が利点の一つに挙げられる．

パターン分割による狙いは，パターン数が少なくなることにより早い段階で遷移を失敗させるスレッド数を増やす事である．ステートマシン内の状態間の遷移に失敗した瞬間にスレッドが計算を終了するという PFAC の特性を利用する．つまり，入力文字列がパターンリストから構成されるステートマシン内の状態間を遷移する回数が多くなるほど，パターンリストを分割する事で削減される遷移回数も多くなることが期待できる．

3.2　パターン長を考慮した分割手法

マルチ GPU 環境下において，パターン分割では最も処理の長い GPU の計算時間がシステム全体の実行時間として表れる．各 GPU の計算時間の差異が大きい場合には，早く処理を終えた GPU は他の GPU が計算を終えるまで何も行わない時間が生まれてしまい，効率的ではない．

そこで，本研究では長大なパターンが 1 つの GPU に偏る事により，入力文字列にマッチする際に遷移回数が多くなり，結果として計算時間が増えてしまう事に注目した．具体的には，長さが異なるパターンを含むパターンファイルを分割する場合は，各 GPU に転送するパターンの長さに偏りがないように，前項で述べた手法にシェルスクリプトによる処理を付け加えて 1 つのパターンファイルを 4 つに分割した．

図 3-2 にパターン数のみを考慮し，パターン長を考慮しないパターン分割手法の簡単な例を示す．図では，各パターンの右の数字はパターン長を示しているものとする．図の例では，4 つの GPU に対してそれぞれ 2 つのパターンを転送しているが，GPU2 にはパターン長 4 のパターンが 2 つ転送される一方で，GPU3 にはパターン長 8 のパターンが 2 つ転送されている．DFA ベースのマッチング処理では遷移回数が計算時間に大きく影響する事を考慮すると，これらのパターンがそれぞれ入力文字列にマッチする場合，GPU2 は他の GPU と比べて計算が早く終わり，効率的な計算が行えない事が予測される．

同様に，図 3-3 に各 GPU のに転送されるパターンファイルのパターン長を考慮したパターン分割手法の簡単な例を示す．図 3-3 ではパターンリストは長いパターンから降順にソートされている事が分かる．このパターンリストから，パターン長 8 とパターン長 4 の組み合わせを各 GPU に転送することによりパターン長の偏りは無くなる．図 2-6 に示したように，長大な入力文字列を処理する際には PFAC では各スレッドが複数回のマッチング処理を行うため，パターンファイルと入力文字列のサイズが膨大になった時には図 3-2 の例と比較した際の明確な高速化が期待できる．

そこで，本研究では，図 3-3 のように各 GPU に転送されるパターンの長さの合計がほぼ同等になるようなパターン分割手法を提案する．

図 3–2: パターン長を無視したパターン分割の例

図 3–3: パターン長を考慮したパターン分割の例

3.3　パターン分割の実装

図 3–4: ソースコード編集部分

本研究では，実装のプラットホームとして PFAC ライブラリ内のサンプルプログラムである”OMP_PFAC.cpp” を用いる．”OMP_PFAC.cpp” では CPU 側で OpenMP により PFAC を適用する GPU の台数に等しい数のスレッドを生成し，パターンファイルは 1 つのものを各 GPU にコピーする．一方入力文字列は 1 つの GPU に最大 256MB まで転送し，入力文字列のサイズが 256MB を超えている場合には 2 つめの GPU に超過分の入力文字列を転送する入力分割の手法を取っている．

このサンプルプログラムに，各 GPU に対してパターンファイルを転送できるように機能を追加した．図 3-4 にソースコード編集部分を示す．本研究は 4 つの GPU を用いる環境で行ったため，1 つのパターンファイルと更にそれを 4 分割した”patternFileA”，”patternFileB”，”patternFileC”，”patternFileD” をあらかじめ用意し，4 つのパターンファイルを 4 つの GPU に対してそれぞれ転送する事でパターン分割を行った．”tid” は OpenMP によって生成された CPU のスレッドの ID を表しており，この各スレッドが同じ ID を持つ GPU にパターンファイルを転送する．よってこの生成されたスレッドの ID 毎にあらかじめ用意した 4 つのパターンファイルを転送する処理をする事により，パターン分割を行う事が出来る．

図 3–5: パターン分割を行うシェルスクリプト

PFAC では入力文字列がステートマシンを走査する遷移回数によって計算時間が決定するが，パターン分割ではステートマシンのサイズが小さくなる分，入力文字列は分割されていない元サイズのデータが各 GPU に割り当てられる．入力分割よりも計算時間を短くするためには,「元サイズの入力文字列が分割されたステートマシンを走査した時の遷移回数」（パターン分割での遷移回数）が,「分割された入力文字列が元サイズのステートマシンを走査した時の遷移回数」（入力分割での遷移回数）よりも少なくなれば良い．

また，パターンファイルをパターン長の偏りが無いように分割するための自動化処理をシェルスクリプトにより行った．図 3-5 にスクリプトの内容を示す．前処理として，分割する前のパターンファイルをソートし，パターン長の長いパターンから降順に並び替えておく．パターンファイルでは一行に 1 つのパターンが格納されている．本研究では 4 台の GPU を用いて実験を行ったため，行番号を 4 で割った時の余りの値を分割するパターンファイル 1〜4 に照らし合わせて分割を行った．完全に各パターンファイルのパターン長の偏りを無くす事はできないが，この分割方法により，ある程度偏りを軽減する事ができる．

第4章

評価実験

本章では提案した手法を実装し，様々なデータセットに対して，入力分割とパターン分割による PFAC の計算時間の比較を行い，二つの手法とデータセットの関係性を明らかにする．また，パターン分割が入力分割と比較して有効となるデータの条件を示す．

また，本研究では入力文字列は 256MB のデータで固定し，パターンリストの変化から，実行時間の変化を検証する．

使用するパターンリストのデータセットを表 4-1 に示す．また，マッチング率を「マッチする回数/入力文字列の長さ」と定義し，指定した時以外は，全てのパターンの長さは等しいものとする．

実験環境は表 4-2 に示す通りである．

本研究で用意する GPU の台数は 4 つで，表 4-2 に示す Tesla K20 と全て同様のものである．また，本章に示す計算時間のデータは全て GPU のカーネルの実行時間とし，転送時間は考慮しないものとする．

表 4–1: データセット

NO_MATCH	入力文字列に 1 つもマッチしない入力文字列とパターンの組み合わせ
ONE_MATCH	入力文字列に 1 つのパターンがマッチする入力文字列とパターンの組み合わせ
ALL_MATCH	入力文字列に全てのパターンが等しくマッチする入力文字列とパターンの組み合わせ

表 4–2: 使用計算機

	Dell Precision-T5610
OS	Ubuntu 14.04
CPU	Intel Xeon E5-2630 2.60GHz
GPU	Nvidia Tesla K20
CUDA core	2496core
GPU memory	5GB
CUDA version	7.5

4.1　シングル GPU における評価

前述した通り，PFAC の計算時間は入力文字列がステートマシン内の状態間をどれだけ遷移したかにより決まる．まずは単一の GPU で PFAC を実行した際の実験結果から，遷移数による計算時間の変化を示す．PFAC ライブラリ内のサンプルプログラムファイル”simple_example.cpp” を用いて実験を行った．

図 4-1 は同じ 256MB の入力文字列に対して，パターンファイルは 10 文字のパターンただ一つのみで検索を行ったものである．この時，二つのデータセットを用意し，一方はパターンと入力文字列が 1 つもマッチしない組み合わせ（**NO_MATCH**）を用い，もう一方はパターンと入力文字列が必ず等しくマッチする組み合わせ（**ALL_MATCH**）を用いる．つまり NO_MATCH は早い段階で遷移を終了し，ALL_MATCH ではパターンの長さの分だけステートマシン内の状態間を遷移する事になる．当然計算時間は遷移数の多い ALL_MATCH の方が長くなる．

次に，NO_MATCH と ALL_MATCH それぞれにに対してのパターン長を変更した時の計算時間の変化を図 4-2 に示す．各パターンファイルのパターン数は 1 である．

図 4-2 では，NO_MATCH はパターン長の増減による実行時間の変化はほとんどなく，ALL_MATCH ではパターン長に比例して実行時間が増加している事が分かる．PFAC では遷移が失敗した時点でスレッドが計算を終了するので，一般に遷移が早期に失敗する NO_MATCH ではパターン数やパターン長の増減に関係なく計算時間は短くなる．ALL_MATCH のパターン長を変更した時の計算時間の変化はパターンの長さの分だけ遷移数が増えるため比例関係になっている．

図 4–1: NO_MATCH と ALL_MATCH の計算時間の比較

次に，パターン長を 10 に固定し，パターン数を変化させる事による計算時間の変化を図 4-3 に示す．図 4-3 を見ると，NO_MATCH は前述したようにパターン数による計算時間の変化は見られない．ONE_MATCH は，1 つのパターンのみが遷移数に大きく関わるとみてよいので，パターン数によって計算時間はあまり変化しない．ALL_MATCH のみがパターン数が増加するにつれ大きく計算時間が伸びている．

PFAC において，マッチング処理は各スレッドで並列に計算されるため，パターン長が変わらなければパターン数が増えても実行時間に変化はないように考えられるが，マッチングするパターンの数を増加させれば実行時間も増加する事が分かった．第 2 章で前述したが，PFAC では 1024 の文字に対して 1 つのスレッドブロックが割り当てられる．最大 256 のスレッドがそれぞれ 4 回ずつ 4 つの文字をスタート地点として走査を行うため，マッチするパターンの数が増えると，その分一つのスレッドが行う遷移回数が増える事が考えられる．

図 4–2: パターン長の変化による実行時間の変化

図 4–3: パターン数の変化による実行時間の変化

図 4–4: SM 1 つあたりのスレッドブロック数の変化による計算時間の変化

ここで，図 2-6 に示すように，PFAC で SMX1 つあたりに割り当てられるスレッドブロック数の観点からパターン数の増加による影響について考察を行う．

本研究では，GPU に Tesla K20m を用いており，Tesla K20 には SMX が 13 機搭載されている．この GPU で SMX 1 つあたりにスレッドブロックが 1 つ割り当てられている時に走査できる文字列の長さは 256*4*13 = 13312 文字である．そこで，ONE_MATCH と ALL_MATCH について，入力文字列が 13312 文字の時と 26624 文字の時とで比較し，SMX1 つあたりのスレッドブロック数が変化する事による計算時間の変化について調査する．パターンはそれぞれパターン数 4，パターン長 10 の ONE_MATCH と ALL_MATCH を用意する．図 4-4 を見ると，入力文字列が 13312 文字の時は ONE_MATCH と ALL_MATCH がほぼ変わらず，26624 文字になると ALL_MATCH の方が計算時間が長くなっている．

この結果から，1 つのスレッドが複数回マッチング処理を行う場合，つまり SM1 つあたりに割り当てられるスレッドブロックの数が増えるとマッチする数が多いほど計算時間が延びるという事が考えられる．この事から各スレッドが複数回処理を行う必要のある膨大な入力文字列を処理する事を想定している本研究では，パターン数が増えるほど計算時間が長くなると考えられる．

4.2　マルチ GPU における評価

複数の GPU で PFAC を実行した際の入力分割とパターン分割の計算時間を比較する．シングル GPU における評価と同様に，入力文字列を 256MB のものに固定し，NO_MATCH，ONE_MATCH，ALL_MATCH の 3 種の方法で作成したパターンファイルを用いて評価を行う．また，NO_MATCH と ONE_MATCH についてはパターン数を増減させても遷移数に変化はないことがシングル GPU での評価により分かっているのでパターン数による計算時間の変化は省略する．また，グラフに示す計算時間は二つの分割手法における各 GPU で最も時間が長かったものとする．また，青のグラフは入力分割を，橙色のグラフはパターン分割を表すものとする．

まず，図 4-5 は NO_MATCH についてパターン数を 100 に固定し，パターン長を変化させた時の入力分割とパターン分割について比較したものである．前述したが，パターン分割は各 GPU が入力文字列を 256MB，入力分割の 4 倍のデータを処理する必要がある．そのため，遷移がほぼ起こらない NO_MATCH では単純に入力分割の 4 倍分の処理を各 GPU がする事になってしまい，実行時間も約 4 倍となっている．

次に，最も遷移の回数が多くなるデータセットであるパターンリスト ALL_MATCH のパターン長とパターン数による計算時間の変化を示したものが図 4-6，4-7 である．図 4-6 ではパターン数を 2000 に固定し，図 4-7 ではパターン長を 20 に固定している．また，どちらもマッチング率は 25 %とした．図 4-6 を見るとパターン長が 20 の時には，パターン分割が入力分割よりも計算時間が短くなっていることが分かる．更に，パターン長を 30 以上にした時にはパターン分割の方が明確に計算時間が短くなっている．また，図 4-7 ではパターン数による比較だが，パターン分割で各 GPU に転送されるパターン数はグラフ中に記載されているパターン数の 4 分の 1 となる．図 4-6 と同様に，パターン数からなる遷移数が増えてくるにつれて，パターン分割の有効性が高まることが分かる．また，ALL_MATCH のパターン数 2000，パターン長 20 における各 GPU の計算時間のデータを図 4-8 に示す．図 4-8 を見ると入力分割もパターン分割も全ての GPU がほぼ同時に処理を終えている事が分かる．これはマルチ GPU 環境下での計算において理想的な状況である．

図 4–5: マルチ GPU における NO_MATCH の実行時間

図 4–6: ALL_MATCH のパターン長変化

図 4–7: ALL_MATCH のパターン数変化

図 4–8: 各 GPU の計算時間

マルチ GPU での計算において，1 つの GPU に計算負荷がかかってしまうと大幅に全体の処理時間が長くなるので，図 4-8 のように計算時間がほぼ同等となるようにが好ましい．

PFAC では遷移回数によって計算時間が決まるため，転送するデータサイズは同じであっても 1 つ 1 つの GPU で計算時間が大きく違う事は十分にあることである．入力分割は入力文字列を分割する際に 1 つの GPU に計算負荷がかかる事を予測するのは難しい．従来手法の入力分割でも入力文字列の先頭から GPU の数と等しい数に分割する手法を用いている．一方，侵入検知システムにおいてはどのパターンが多くマッチするかはある程度既知であることが多い．また，マッチする事が分かっているパターンが複数個ある場合，パターン長とパターン数からある程度の遷移回数を予測し，各 GPU の計算時間をできるだけ揃える事も不可能ではない．

ここで，入力文字列 256MB のうち，パターンとマッチする文字列が先頭の 64MB に偏ったと仮定した時の入力分割とパターン分割の計算時間についてのデータを取得するため，入力文字列を変更する．先頭の 64MB についてはターゲットパターンを多く含み，残りの 192MB はランダムに生成された文字列とする．この時，先頭の 64MB の入力文字列のマッチング率を変化させ，パターン数とパターン長は共に 100 で各 GPU の計算時間の変化を計測したものを図 4-9，4-10 に示す．

図 4-10 を見ると入力分割において，GPU0 にパターンを多く含む入力文字列が転送されたため，GPU 0 のみ大幅に計算時間がかかり，マッチング率が 10 %でもパターン分割の方が計算時間が短くなった．

パターン分割では各 GPU は 256MB の入力文字列に対してマッチングを行うが，この実験では 256MB のうち 192MB はランダムに生成された入力文字列なため，192MB 分の処理を行う間は遷移回数が少なく，計算時間が短くなる．入力分割では GPU0 以外の GPU はランダムに生成された 64MB の入力文字列を処理したため，早期に処理を終えたが，GPU0 が計算を終えるのをしばらく待機する状態となった．マッチング率が 10 %の時点でパターン分割の計算時間が入力分割よりも短くなる結果になった．

図 4–9: 各 GPU の計算時間 2

図 4–10: 各 GPU の計算時間 3

最後に，前章の 3.2 で述べたように各パターンのパターン長が異なる場合についての評価を行う．実験で用いるデータセットの種類は ALL_MATCH だが，パターン長は｛20，15，10，5｝の 4 つの長さのパターンが等しく混在している場合を想定して計算時間を測定する．

提案手法の有効性を示すため，各パターンファイルのパターン長に偏りがある前章の図 3-2 のような分割手法を「パターン分割 1」，提案手法である前章の図 3-3 のようにパターン長の偏りを軽減する分割手法を「パターン分割 2」として実験を行った．パターン分割 1 ではパターン長 20 のみのファイル，15 のみのファイル，10 のみのファイル，5 のみのファイルに分割するものとし，パターン分割 2 では 4 つの長さのパターンがそれぞれ各 GPU に均等に振り分けられるようにした．また，マッチング率は 25 %とした．

図 4-11 にパターン数 2000(各パターン長のパターン数が 500 ずつ) の時のパターン分割 1 とパターン分割 2 の計算時間の比較を示す．この図は，パターンと入力文字列がマッチする場合には各 GPU に分割するパターンファイルのパターンの長さに偏りを持たせない事で計算時間が削減される事を示している．

パターン分割 1 の計算時間に注目すると，図 4-7 に示す全てのパターン長が 20 の時とほぼ同等の値になっている．図 4-12 に示す各 GPU の計算時間から分かるように，最も計算時間の長い GPU が図 4-11 での計算時間として表れている．つまり，今回の場合，パターン長 20 のみのパターンファイルを処理した GPU1 の計算時間が最も長くなり，同時に GPU1 においては全てのパターン長を 20 とした図 4-7 と同条件となったためであると考えられる．

一方，パターン分割 2 においては各 GPU の処理量が等分割された事により計算時間もほぼ等しくなり，パターン分割 1 と比較して計算時間が短くなった．この結果から，パターンの長さを考慮して各 GPU に振り分ける事により効率的にマルチ GPU での並列化が行われたと言える．

図 4–11: パターン分割手法による計算時間の比較

図 4–12: 各 GPU の計算時間 4

図 4–13: 分割手法による計算時間の比較

図 4-13 は，図 4-7 と同様にパターン数の変化による入力分割，パターン分割 1，パターン分割 2 の計算時間の比較を示している．全てのパターンの長さが 20 の時と比べて入力分割においても計算時間が短くなっているため，パターン分割 2 が入力分割よりも計算時間が短くなるにはパターン数が約 3000 以上の時になっている．しかし，パターン分割 1 と 2 を比較するとパターン数が増える毎に計算時間の差が開いている事が分かる．これは，パターン数が増える程パターンの長さが 20 のみのパターンで構成されたファイルと各パターン長を均等に分割したファイルとの間で，マッチング処理を行う際の遷移回数の差が開いていく事を示しており，提案手法の有効性を確認できる．

第5章

関連研究

本章では，侵入検知システムにおけるテキストマッチング処理の高速化に焦点をあてた関連研究について述べる．また，実装するテキストマッチングアルゴリズムや用いるデバイスに違いはあるが，本研究の提案手法であるパターン分割を用いた研究も存在するので同様に述べる．

5.1 侵入検知処理高速化に関する研究

近年では，Snort [12] を用いたセキュリティシステムの侵入検知処理高速化が注目されており，種々の研究が進んでいる．Snort は他の IDS ツールと比較して，高トラフィックやフラグメント化された検知しにくいパケットによる侵入攻撃に対しても，高い検知能力を有することが報告されている [13]．また，最新の侵入被害に対しても対応が早いことから侵入検知システムに関する研究で用いられる事が多い．

Snort ではパケット識別によるフィルタリング処理，文字列のパターンマッチング処理，プロトコルの処理に分けられるが，その中でも最も処理負荷が大きくなる文字列パターンマッチング処理であるため，処理を高速化するためにパターンマッチングアルゴリズムの高速化と，パターンマッチング処理の負荷削減が現在も検討されている．

林 經正氏らの論文 [14] では，Snort で用いられている Boyer-Moore アルゴリズム（BM アルゴリズム）[15] は，パターンリスト内の全ての検索文字列とパケット内データとの間でパターンマッチング処理を行うもので，検索文字列数が多くなるとマッチング処理量が非常に多くなり，計算負荷が大きくなってしまう問題について触れている．そこで Aho-Corasick アルゴリズムと BM アルゴリズムを応用し，共通文字列を探索木の親ノードとする Aho-Corasick Boyer-Moore(ACBM) アルゴリズム [16] を用いることにより，BM アルゴリズムよりマッチングする処理数自体を減らし，高速処理が実現する事を明らかにしている．

また，同論文ではパターンマッチング処理の前に前処理としてあらかじめ処理する必要がないと判断できるパケットをフィルタリングする事により IDS で処理するパケット数を削減する手法も取られている．パケットフィルタリングをハードウェアで，パターンマッチ処理をソフトウェアで処理するハードウェア/ソフトウェア協調処理によって高負荷状態でもマッチング処理が可能であることを明らかにしている．

Nhat-Phung Tran 氏らの論文 [5] では，並列化した Aho-Corasick アルゴリズムを GPU で実装する際に，入力文字列データをブロックに分け，GPU のスレッドブロック毎に割り当て、グローバルメモリへのアクセスを合体させる事によりグローバルメモリの負荷の数を最小限にする最適化手法について述べている．

5.2　パターン分割に関する研究

前章では，PFAC ライブラリでは PFAC をマルチ GPU 実装する際に用いられる並列化手法は入力分割である事を述べたが，パターンマッチング処理を高速化させるためのアプローチとして，入力文字列に焦点をあてたアプローチとは対照的に，パターンに焦点をあてた先行研究も存在する．

S.Arudchutha 氏らの論文 [9] では，マルチコア CPU 環境下で，並列化した Aho-Corasick 法を実行する際に各コアに対してパターン分割を行い，高速化を実現している．

同じく，X.Bellekens 氏らの 2013 年の論文 [10] では，文字列検索アルゴリズムの一種であるクヌースーモリスープラット法（KMP 法）[17] を単一の GPU で実行する際に各コアに対してパターン分割を行っている．

第6章

結論

6.1 まとめ

本論文では複数ＧＰＵ環境下におけるＰＦＡＣアルゴリズムの並列化手法を提案した．従来手法と様々なデータセットに対して比較を行い，ある条件のデータセットにおいて提案手法の有効性を示した．

同時に，提案手法が従来手法と比較して有効となるデータセットの条件を示した．入力文字列のステートマシン内の遷移回数が多いほどパターン分割の方が計算時間が短縮する事が明らかになったが，マッチング率 25 %という高確率でマッチする条件で実験を行ったにも関わらず，パターン分割が入力分割よりも高いスループットが発揮されるまでにパターン長 20 の時にパターン数が約 2000 以上マッチする必要がある事が分かった．この結果は現在の侵入検知システムの多くの場合において入力分割がパターン分割よりも高いスループットを提供する事を示しているが，第 1 章で述べたように今後は有害なパケットが増え，それに伴って侵入検知システムがマッチングを行うパターンの数も増加していく事が予想される．パターン数が増え，マッチング率も増えていくとパターン分割のスループットが入力分割を上回るので，本研究では提案手法であるパターン分割の将来的な有効性が示す事ができたと言える．

また，パターンの分割手法として，パターン長の長いパターンが一つの GPU に偏らないための分割法を提案し，有効性も確認した．

6.2　今後の課題

本来，本研究では実際の Snort で用いられる入力文字列とパターンの組み合わせのデータで実験をする事が理想であったが，技術的な問題と時間的な問題が重なり断念する結果となった．代わりのデータとして入力文字列とパターンは自作のプログラムにより生成したデータを用いたが，マッチング率やパターン長など手探りで進めた点が多い．第一の今後の課題としては，実際の侵入検知システムで用いられる入力文字列とパターンで実験を行い，本研究の提案手法がどの程度のスループットを提供できるかを調査する．

提案手法であるパターン分割が，特定の条件下では従来手法の入力分割と比較して計算時間が短くなることは示すことができたが，現実的ではないデータセットでの結果であるため，実用性があると主張できるほどの結果を残すことはできなかった．

また，今回の研究ではパターン長の長いものほど遷移回数が多いことを定義したが，短いパターンでも何回も繰り返しマッチするような場合には遷移回数が増えるため，そういったパラメータについても考えてパターンを分割する必要がある．

本研究では長いパターン長のパターンが一つの GPU に偏らない事を考慮した分割手法を提案したが，パターン分割は入力分割と比べて各 GPU への分割方法が多く考えられる．マルチ GPU において目指すべき分割方法は計算時間を揃える事のため，計算量に密接に関係する遷移回数をできるだけ等分割する手法の提案が課題である．また，以前調べたことはあるが，実際の侵入検知システムにおいてよく検知するパターンについて更に詳しく調査する必要がある．

謝辞

本研究を行うにあたり，研究の場を与えていただき，なおかつ適切な御指導，御助言をして頂いた本多弘樹教授と三輪忍准教授並びに古賀久志准教授に心より深く感謝致します.

そして，本研究を御支援，御協力下さった高性能コンピューティング学講座の本多・三輪研究室の皆様に御礼申し上げます.

平成 28 年 1 月 28 日

參考文獻

[1] P.-C. Lin , Y.-D. Lin , T.-H. Lee and Y.-C. Lai"Using string matching for deep packet inspection"IEEE Computer, vol. 41, no. 4, pp. 23-28, 2008.

[2] Alfred V. Aho, Margaret J. Corasick, "Efficient string matching: an aid to bibliographic search" Communications of the ACM, Volume 18 ,pp. 333-340 June. 1975

[3] Cheng-Hung Lin, Sheng-Yu Tsai, Chen-Hsiung Liu, Shih-Chieh Chang, and Jyuo-Min Shyu, "Accelerating String Matching Using Multi-threaded Algorithm on GPU," IEEE GLOBAL COMMUNICATIONS CONFERENCE (GLOBECOM), Miami, Florida, USA, pp.1-5, December 2010.

[4] L.Chien et al., "pfac - PFAC is an open library for exact string matching performed on NVIDIA GPUs", http://code.google.com/p/pfac/.

[5] Nhat-Phuong Tran,Myungho Lee, Sugwon Hong, Jeeyoung Choi "High Throughput Parallel Implementation of Aho-Corasick Algorithm on a GPU" Parallel and distributed Processing Symposium Workshops and PhD Forum(IPDPSW), 2013 IEEE 27th International.pp1807-1816 May 2013

[6] Shima Soroushnia1, Masoud Daneshtalab1, Juha Plosila1, Tapio Pahikkala1 and Pasi Liljeberg1, "High Performance Pattern Matching on Heterogeneous Platform" Journal of Integrative Bioinformatics, Oct 2014

[7] Cheng-Hung Lin, Chen-Hsiung Liu, Lung-Sheng Chien, Shih-Chieh Chang, "Accelerating Pattern Matching Using a Novel Parallel Algorithm on GPUs," IEEE Transactions on Computers, vol. 62, no. 10, pp. 1906-1916, Oct. 2013

[8] Cheng-Hung Lin, Chen-Hsiung Liu, Lung-Sheng Chien, Shih-Chieh Chang, Wing-Kai Hon "PFAC Library:GPU-Based String Mathing Algorithm" http://on-demand.gputechconf.com/gtc/2012/presentations/S0054-PFAC-Library-GPU-Based-String-Matching-Algorithm.pdf

[9] S. Arudchutha, T. Nishanthy, R. G. Ragel ”String Matching with Multicore CPUs: Performing Better with the Aho-Corasick Algorithm” IEEE International Conference on Industrial and Information Systems (ICIIS), pp. 231-236, 17-20 Dec. 2013

[10] X Bellekens, I Andonovic, RC Atkinson ”Investigation of GPU-based Pattern Matching” Post Graduate Symposium on the Convergence of Telecommunications, pp.1-5 2013

[11] 高橋　亮介, 井上　潮　”GPGPU を用いた並列テキストマッチング”　DEIM Forum 2012 pp.1-5

[12] The Open Source Network Intrusion detection System http://www.snort.org

[13] 伊藤良孝, 平井伸幸, Ids の検知能力を検証する. Nikkei Internet Technology, 2002.9.

[14] 林 經正, 横山 幹, 高原 厚, 岩橋 政宏, Snort を用いた侵入防止システムの構築と侵入検知処理高速化の検討, 社団法人情報処理学会研究報告　2003.5

[15] BOYER R.S. and MOORE J.S. A fast string matching alorithm. Communications of the ACM. No.20,1979

[16] B. Commentz-Walter. A string matching algorithm fast on the average. 1979

[17] Donald Knuth; James H. Morris, Jr, Vaughan Pratt (1977 年). “Fast pattern matching in strings”. SIAM Journal on Computing 6 (2): pp323-350.

図一覧

表一覧