社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

Random Walker 単位の非同期処理における Random Early Detection を用いた自律的輻輳制御機構

滝沢　　駿† 　金子　晋丈††,†††

† 慶應義塾大学大学院理工学研究科
†† 慶應義塾大学理工学部
††† 慶應義塾大学デジタルメディア・コンテンツ統合研究センター
E-mail: †{takishi,kaneko}@inl.ics.keio.ac.jp

あらまし　大規模グラフ解析において，Random Walk (RW) は有用であり，同期型，非同期型の分散グラフシステムに実装されてきた．しかし，動的グラフ等の均等分割が困難なグラフを複数のサーバで分散管理し，非同期に RW を実行しつつ，実行時間を短縮する場合，各サーバの Random Walker (RWer) の待ち行列である Random Walker Queue (RQ) のキューイング遅延が課題となる．そこで本研究では，非同期に RW を実行するとともに，各サーバが RWer の流量を自律的に調整する機構として，自律的 RWer 輻輳制御機構を提案する．本機構は，ルータなどで採用されているキュー管理アルゴリズム Random Early Detection (RED) を適用し，RWer の破棄を許容しながらサーバの RQ を管理する. RED を用いたキュー管理と RWer ロス検知による輻輳の初期検知，そして sleep を用いた RWer 生成速度調整を行い，バースト的な RWer 生成を防ぐ. 評価の結果，キュー管理をテールドロップにした場合と比べ，実行時間が約 1 秒早くなり，RW の追加実行数が約 84 % 少なくなった.

キーワード　Random Walk，グラフ解析，非同期処理，分散処理，Random Early Detection

Autonomous Congestion Control Mechanism using Random Early Detection in Asynchronous Random Walker Processing

Shun TAKIZAWA† and Kunitake KANEKO††,†††

† Graduate Schooll of Science and Technology, Keio University
†† Faculty of Science and Technology, Keio University
††† Research Institute for Digital Media and Content, Keio University
E-mail: †{takishi,kaneko}@inl.ics.keio.ac.jp

Abstract　In large-scale graph analysis, Random Walk (RW) is useful and has been implemented in both synchronous and asynchronous distributed graph systems. However, when a graph such as a dynamic graph, which is difficult to divide equally, is distributed across multiple servers and RW is executed asynchronously, queuing delays occur in the Random Walker Queue (RQ), which is a queue of Random Walker (RWer) on each server. Therefore, we propose an autonomous RWer congestion control mechanism that executes RW asynchronously and allows each server to adjust the RWer flow rate autonomously. This mechanism applies the queue management algorithm Random Early Detection (RED), which is used in routers, to manage RQ on server while allowing RWer discards. The evaluation results show that the execution time is about 1 second faster and the number of additional RW executions is about 84 % less than the case with tail-drop queue management.

Key words　Random Walk，graph analysis，asynchronous processing，distributed processing，Random Early Detection

1. はじめに

Random Walk (RW) はグラフ解析において広く利用されている．RW を用いたグラフ演算の例として，コミュニティ検出，リンク予測，類似性推定，推薦システムなどが挙げられる．これらのグラフ解析はそれぞれ用途が異なるため，グラフの利活用が盛んになっている現代では，一つの保存されたグラフに対して，RW を使った様々なアプリケーションが実行されること

になる．

これまで，大規模グラフにおける RW を複数サーバで実行するときは，1. グラフを均等に分割しそれぞれのサーバに配置する．2. 全サーバで頻繁に進捗度合いを合わせる同期処理によって RW を実行する． といった手順を踏んでいた．グラフを均等に分割することで， 同期処理で問題となる負荷不均衡を軽減することができる．

しかし，現実のグラフは時間とともに変化するため，例え初めに均等に分割したところで，次第に不均等になり， 同期処理における負荷分散が困難になる．さらに， 複数のユーザが同じグラフデータを使ってそれぞれ別々のアプリケーションを実行する場合，演算， メッセージ送信， 同期タイミングがそれぞれ異なるため，Master-Worker 型の同期処理を同時に行うことは困難である．また， 同期型のシステムの欠点を補うため開発された非同期型の分散グラフ処理システムでは，早く演算が終わったサーバは， 遅いサーバを待つことなく，次の演算に取り掛かる．これにより同期の待ち時間の発生を防ぐことができるが，メッセージをまとめて送信するため， サーバ内でメッセージ単体に対し， 送信待ちが発生する．

そこで本研究では，グラフ上で RW をする主体である Random Walker (RWer) を単位とした演算， 送受信を， サーバ間， サーバ内で独立して行う RWer 単位の非同期処理の実現を目指す．非同期に RW を実行することで，同期処理での課題である，負荷不均衡による待ち時間の増加を解消できることに加え，サーバ内での演算，送受信に関して，それぞれの RWer がお互いを待つことがなくなる．RWer 単位の非同期処理の課題は， RWer がある一つのサーバに集中してしまうことである．なぜなら，それぞれのサーバが保持しているグラフは不均一なものであり，かつそれぞれのサーバが生成する RWer の量も一定ではないからである．その結果， それぞれのサーバが保持している RWer の待ち行列である Random Walker Queue (RQ) のキューイング遅延が増加する．また， RWer の破棄を許容している場合は，追加実行の発生とそれによる実行時間の増加が問題となる．

本研究ではこの問題点に対し， RWer 単位の非同期処理において，各サーバが RWer の流量を自律的に調整する機構として，自律的 RWer 輻輳制御機構を提案する．自律的 RWer 輻輳制御機構に求められることは，RWer の集中によるキューイング遅延と RWer 破棄の低減，そして RWer 間の公平性を保つことである．この要件に対し本機構は，ルータなどで採用されているキュー管理アルゴリズムである Random Early Detection [1] のアイデアを利用し， RWer の破棄を許容しながらサーバの RQ を管理する．そして， RWer の破棄を検知したサーバは RWer の生成スピードを落とす．この RED を用いたキュー管理を導入することで， RWer の集中 (輻輳) の初期検知ができるだけでなく，複数のサーバが同時に RWer の生成スピードを下げることで， 再び同時に生成スピードが上昇してしまうグローバル同期を防ぐことができる．

最後に， 以降の本論文の構成を示す．2 章では関連研究について述べる．3 章では提案手法を述べ，4 章で提案手法に対する評価を行い，5 章で本論文の結論を述べる．

2. 関 連 研 究

2.1 分散グラフ処理システムとグラフ分割

大規模なグラフの演算のために，様々な分散グラフ処理システムが開発されてきた．[2]， [3] のような同期型の分散グラフ処理システムでは， 全サーバの進捗度合いを合わせるための同期が頻繁に発生する． そのため， 早く演算が終わったサーバはボトルネックとなるサーバの演算が終わるまで待機する必要がある． この待ち時間は， グラフ分割の質に依存することがわかっている．[4] そこで既存の同期型の分散グラフ処理システムでは通常， 均等なグラフ分割を前提としている．

同期型のシステムの欠点を補うために開発されたのが， [5]，[6]， [4] のような非同期型の分散グラフ処理システムである．これらのシステムでは， 早く演算が終わったサーバは， 遅いサーバを待つことなく，次の演算に取り掛かる． これにより，同期型の問題点である， 負荷不均衡による待ち時間の発生を防ぐことができる．しかし， こちらも同期型のシステムと同様，均等なグラフ分割を前提としており，本研究が想定する不均等なグラフ分割でのグラフ演算の実行を考慮していない．

基本的に分散グラフ処理システムでは，元グラフをパーティショニングアルゴリズムにより分割する．様々なパーティショニングアルゴリズムがあるが，評価する際， 最も不均等な分割として上げられるのが，それぞれのパーティションで頂点数が同じになるようにランダムに分割する手法である．本研究では， 動的グラフを想定しており，時間の経過とともにそれぞれのパーティションの頂点数すら不均等になっていくため，通常のランダム分割よりも， より不均等なグラフ分割となる．このようなグラフ分割の場合，既存手法では， 想定していた性能が出なくなると考えられる．

2.2 キュー管理アルゴリズム

ネットワークの輻輳を防ぐための輻輳制御という技術があるが，そこで重要な役割を担っているのが， ルータでのキュー管理である．輻輳状態になると， ルータに蓄積するデータ量が増えていく．ルータはキュー管理アルゴリズムに則り， そのデータを処理し，適切な輻輳制御のサポートをする．

最も単純なキュー管理アルゴリズムであるテールドロップでは，ルータのバッファが満杯になり， 入りきらなくなったパケットを廃棄する．このとき， 複数の送信者が同時にパケットのロスを検知し， ウィンドウを減らし，スロースタートすることによって TCP グローバル同期が発生する．

DECbit [7] では， 平均キュー長がある値以上のとき，パケットのヘッダに輻輳の印をつけ， 送信側が輻輳の印を一定割合以上観測した場合にウィンドウを減らす． この手法では， バースト的なトラフィックが到着したときに多くのパケットに輻輳の印をつけることになり，偏りが発生する．

Random Early Detection (RED) [1] では，平均キュー長が閾値を超えたとき， ルータは到着する各パケットをある確率でドロップする．これにより， バッファが満杯になる前から一部の送信者がパケットのロスを検知し， スロースタートをすることができるため，キュー長の安定化につながる．さらに， ランダ

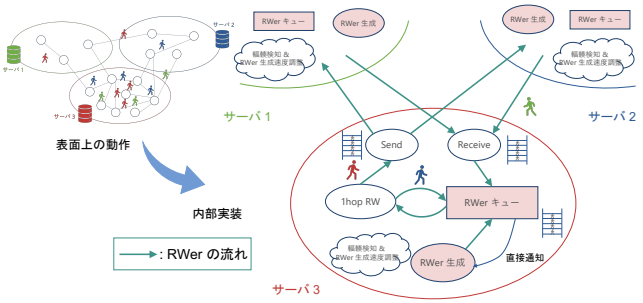

図 1: 自律的 RWer 輻輳制御機構の概要

ム性により，多くのパケットが一斉にドロップすることがないため，TCP グローバル同期を防ぐことができる．

Deep Buffer [8] はメモリの価格が下がったことにより普及するようになった巨大なバッファである．キューイング遅延は増えるが，大きなトラフィックをパケットのドロップなしで捌き切ることにより，スロースタートによるスループットの一時的な低下を防ぐことができる．

本研究が提案する自律的 RWer 輻輳制御機構では，RWer 間の公平性のために， RWer 間の待ち時間の差を削減したいことに加え，RWer 単体の価値があまり高くないことから， 破棄を許容できるため，Deep Buffer ではなく， RED を選択した．

3. 自律的 RWer 輻輳制御機構

3.1 概　　要

図 1 は本機構の概要を示す．自律的 RWer 輻輳制御機構は，サーバ毎に独立して動作する．ある頂点から RW を実行する場合，その頂点を保持しているサーバが RWer を生成し，そのサーバの Random Walker Queue (RQ) に格納する．そして， RQ に格納されている RWer を取り出し，一定の確率で終了させるか， ランダムな隣接頂点へ遷移させる．その後， RWer の滞在頂点を保持しているサーバが元のサーバと変わらない場合，再び RQ に格納され，変わる場合は， そのサーバへ RWer を送信する．動的グラフの不均等さと多様なアプリケーションに対応するため，自律的 RWer 輻輳制御機構では， 各サーバが独立して RW 演算と RWer の送受信を行う非同期処理を採用する．

3.2 設 計 指 針

設計指針では， RWer と，RQ の設計について述べる．

まず， RWer の設計について述べる． 本機構では， 待ち時間削減という非同期処理の利点を活かすため，1 RWer を 1 メッセージとし， 通信プロトコルとして UDP を使用する．TCP ではなく， UDP を使用する理由は， 以下の 3 点，(1) TCP はサーバ間を跨ぐ通信のたびに 3-way handshake 等のプロトコルオーバヘッドが大きいこと，(2) 各 RWer が運ぶデータサイズは UDP のペイロードに十分収まること，(3) 本機構が RWer のロスの検知メカニズムを有していること， である．

次に RQ の設計について述べる．RQ には，RWer 毎の実行時間の偏りを抑えるために，キューイング遅延の増加を抑えることが求められる．そこで RQ の キュー管理アルゴリズムとして RED を採用する．これにより， それぞれのサーバは輻輳を早期に検知することができ，生成スピードを落とすことによって輻輳を防ぐことができる．特に，RED のランダムな RWer の破棄は， 各サーバの送信レートを独立に低下させるため，各サーバの RWer 生成の集中を避けることができる．

なお， RQ を複数作成することでキューイング遅延を軽減する案が考えられるが，そうした場合，RWer の送信待ちのキューで再びキューイング遅延が増加する．つまり， 結局いずれかのキューでキュー管理アルゴリズムを適用することになる．そのため， 本機構では， サーバ上での RWer の数を把握しやすくするため RQ を一つに統一しキュー管理を行う．

3.3 構　　成

本機構では構成要素の内， RWer キュー (Random Walker Queue， RQ)，RWer ロス検知，RWer 生成 (Random Walker Generator， RG)，にそれぞれ下記の工夫を加えることにより， 輻輳を初期段階のうちに検知し， 回避する．

3.4 Random Walker Queue (RQ)

RQ は， Random Early Detection [1] を適用し，RQ が満杯になるよりも前から RWer をキュー長に基づいて確率的に破棄する．これにより， キューが満杯になってから破棄を始めるテールドロップに比べ，輻輳を未然に検知しやすくなる．また， テールドロップではキューが満杯のときに到着した RWer のみが捨てられるのに対し，RQ では輻輳の原因となるサーバ， つまり大量に RWer を生成しているサーバから生成された RWer の破棄率が上昇するため，RQ はテールドロップに比べて公平な制御ができる．

アルゴリズムは基本的に RED と同様である．min_{th}，max_{th}，max_p はパラメタであり，キューの長さを q としたときの RWer の破棄率を p とすると， $q < min_{th}$ のとき $p = 0$，$min_{th} \leq q < max_{th}$ のとき p は q に対して線形変化する値 (最大 max_p)，$max_{th} \leq q$ のとき $p = 1$ となる．

3.5 RWer ロス検知

RWer ロスの検知は， RWer の平均帰還時間と偏差による推定帰還時間に基づくタイムアウトによる検知と，RQ からの直接通知による検知を併用する．

3.5.1 タイムアウトによる Random Walker ロスの検知

本機構では， 生成された RWer の ID が起点頂点毎のリングバッファに格納されていく．各サーバはこの RWer ID に対して， その RWer が生成された時刻と RWer が終了したかどうかの情報を保持する．そして RWer 生成時， リングバッファを走査していき，生成から経過した時間 t が $T < t < XT$ の同じ頂点から生成された RWer を調べ， まだ終了していないものがあったとき， ロスを検知したとみなす．ここで， X はパラメタであり， T は RWer の推定帰還時間である．X が小さいと RWer ロスを見逃す確率が増加し，逆に大きいと冗長な RWer ロス検知となる．そこで本機構では $X = 5$ に設定する．T の計算には RFC 2988 の TCP タイムアウト時間を適用し，以下のように求める．

$$T = RWer \text{ の平均帰還時間} + 4 * Deviation(Deviation \text{ は偏差}) \quad (1)$$

RWer の平均帰還時間と Deviation は以下のようにして求める．

$$RWer \text{ の平均帰還時間} = (1 - 0.125) * RWer \text{ の平均帰還時間} + 0.125 * RWer \text{ の帰還時間} \quad (2)$$

$$Deviation = (1 - 0.25) * Deviation + 0.25 * |RWer \text{ の帰還時間} - RWer \text{ の平均帰還時間}| \quad (3)$$

3.5.2 RQ からの直接通知による Random Walker ロスの検知

タイムアウトによる RWer ロス検知に加えて， RQ からの直接通知による RWer ロス検知を併用する．RQ で RWer を破棄したとき， もしその RWer の起点サーバ (RW の開始頂点を管理しているサーバ) が現在滞在しているサーバと一致していた場合に限り， RWer の破棄をそのサーバにある起点頂点に直接通知する．これにより， タイムアウトによる RWer ロス検知よりも確実に， 素早く RWer ロスの検知ができる．この直接通知は， ネットワーク上でのルータから送信者への輻輳の通知と同じ役割を果たす．通常， この通知にはルータと送信者間の RTT による遅延が発生してしまうが，今回の手法ではサーバ内での通知になるため遅延が発生しない．

3.6 Random Walker Generator (RG)

RG は， RWer 生成間の sleep 時間を制御し， 起点頂点毎に RWer 生成速度調整を行う．x をパラメタとして， x 回以上連続で RWer のロスを検知しなかった場合， sleep 時間を 0 ns にし，それ以外のときは sleep 時間を最小値である 1 ns に設定する．今回， スリープの最小値を 1 ns としているが， スリープ関数の呼び出しにかかる時間が影響し，実際のスリープ時間は 1000 ns 程度となる．

4. 実装・評価

自律的 RWer 輻輳制御機構の導入によって，高スループット，低 RWer ロスが達成されることを実行時間， 総追加実行数の点から明らかにする．また， グラフトポロジを変化させ， RWer の集中度合いの影響を評価した．

4.1 実　　装

図 2 に本機構の実装を示す．各サーバは， Generate_RWer スレッドを保持しており，ここで RWer が生成され， RW が開始する．生成された RWer はまず， RQ に格納される．1hop RW スレッドでは， RQ から RWer を取り出し，一定の確率で終了させるか， ランダムな隣接頂点へ遷移させる．そしてその RWer について， 自サーバが保持する頂点へ遷移した場合，再び RQ に格納し， 逆に他サーバが保持する頂点へ遷移した場合は Send_Queue に格納する．また， 終了した RWer の生成元が他サーバの場合も，RW の終了を起点頂点に通知するため， Send_Fin_Queue に格納する．Send_RWer スレッド，Send_RWer_Fin スレッドでは，それぞれ キューから RWer を取り出し，それを元にメッセージを生成し， 他サーバへ送信する．Receive_RWer スレッドでは， 他サーバから遷移してきたメッセージから RWer を復元し， もしそれが終了した RWer でない場合，RQ に格納する．

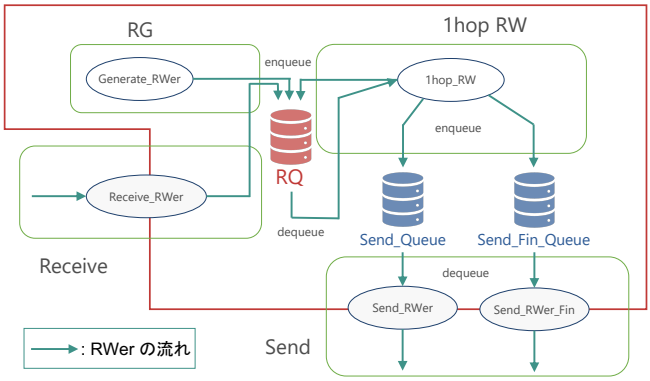

図 2: 自律的 RWer 輻輳制御機構の実装

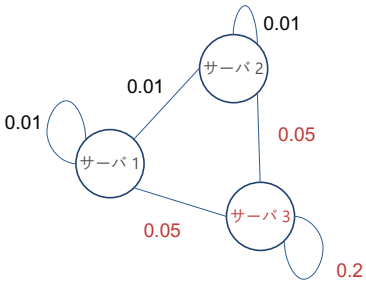

図 3: 実験で使用するグラフの生成パラメタ

4.2 評 価 環 境

サーバ 1， サーバ 2 は， Ubuntu 20.04 LTS， Intel(R) Xeon(R) CPU E5-2643 v2 @ 3.50GHz，24 コア， 32GB メモリ，サーバ 3 は， Ubuntu 20.04 LTS， Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz，8 コア， 32GB メモリを評価に用いた．サーバ 1，サーバ 2 に対して性能の低いサーバ 3 を使用することであえてサーバ 3 に負荷がかかる設定にし，意図的に輻輳を発生させる．

4.3 データセット

グラフの生成には， エッジ生成確率を制御できるアルゴリズムである， Stochastic block model (SBM) [9] を使用した．SBM では， partition 数， 各 partition 内の頂点数， partition 内の頂点間のエッジ生成確率，partition 間の頂点間のエッジ生成確率を指定し， グラフを生成することができる．本実験では， 300 頂点のグラフを 3 台のサーバで分割管理 (1 台 100 頂点) している状態を想定し，グラフを生成した． 図 3 にグラフの生成パラメタを示す．サーバ 1 の頂点間のエッジ生成確率は 0.01， サーバ 1 とサーバ 3 の頂点間のエッジ生成確率は 0.05， のように，サーバを結ぶ線上の数値がエッジ生成確率となる．今回， 意図的にサーバ 3 にエッジを集中させることにより，不均等なグラフ分割を再現しており， RW を実行したとき RWer はサーバ 3 に溜まりやすくなる．

4.4 Random Walker 生成方式の検証

それぞれのサーバが同時に RW を開始し， 全頂点から 10,000 回分，つまり 1 サーバで合計 1,000,000 回分，全サーバで合計 3,000,000 回分の RW が終了したとき，実験終了とした．このときの一番遅いサーバの実行時間を全体の実行時間とし，実験終了までに実行した RW の総量のうち，本来の必要回数 3,000,000

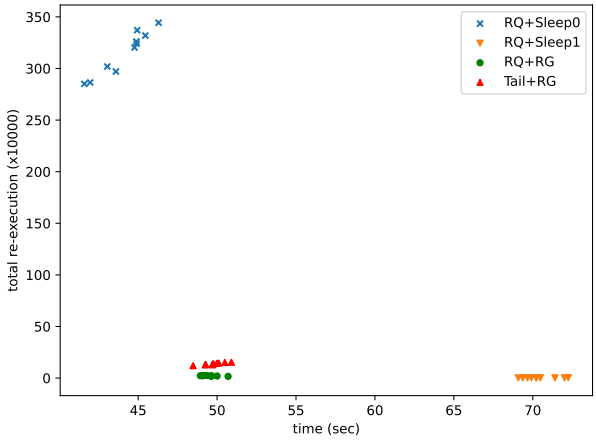

図 4: Random Walk の実行時間と総追加実行数 (自律的 RWer 輻輳制御機構とその他の比較)

回 を超えた分の回数を総追加実行数とした．

自律的 RWer 輻輳制御機構を RQ + RG として，RQ + sleep0 (RQ と RWer 生成間の sleep = 0 ns を組み合わせたもの)，RQ + sleep1 (RQ と RWer 生成間の sleep = 1 ns を組み合わせたもの) を比較した．RQ のパラメタについて，$min_{th} = 5$，$max_{th} = 100$，$max_p = 0.1$，RG のパラメタについて，$x = 3$，RW の終了確率について，$\alpha = 0.2$ とする．この先の実験でも，特に言及していない場合は同様のパラメタ設定とする．また，ロスの検知はタイムアウトによるものと直接通知によるものを併用する．

結果を図 4 に示す．横軸は実行時間，縦軸は総追加実行数を表しており，それぞれの手法ごとに 10 回実行し，結果をプロットした．平均値で見たとき，自律的 RWer 輻輳制御機構は，RQ + sleep 0 と比べて約 5 秒遅いが，総追加実行数が約 3,130,000 回少なく，RQ + sleep 1 と比べて約 20 秒早いが，総追加実行数が約 20,000 回多くなった．スリープをなくすと総追加実行数が大幅に増加し，本来の回数の 2 倍以上実行することになる．このような場合，RW 結果の信頼性が損なわれるだけでなく，サーバ 3 上の RWer 流量が常に限界の状態となっているのでシステムのロバスト性が低いことが想像できる．例えば現在のグラフに，サーバ 1，2 と同じような，サーバ 3 へ遷移する RWer が多いサーバが追加され，サーバ 3 の負担が増加した場合，新しいサーバからの RWer の大半が破棄されてしまうことになる．また，RWer 生成間に毎回 1 ns のスリープを入れた場合，RWer ロスは少なくなるが，スループットが低くなってしまうことがわかる．3.6 で言及したように，sleep 関数の呼び出しの関係で最低でも 1000 ns 程度のスリープが入ってしまうことから，毎回スリープを入れるのは適切でない．そこで自律的 RWer 輻輳制御機構は，1 ns のスリープと 0 ns のスリープを適宜使用することで，安直なスリープを使った RWer 生成速度調整手法よりもバランスの良い低 RWer ロス，高スループットを達成することができる．

4.5 キュー管理アルゴリズムの検証

3.6 と同様の実験を，RQ + RG と Tail + RG (RQ に確率的破棄を用いずテールドロップにしたものと RG を組み合わせたもの)

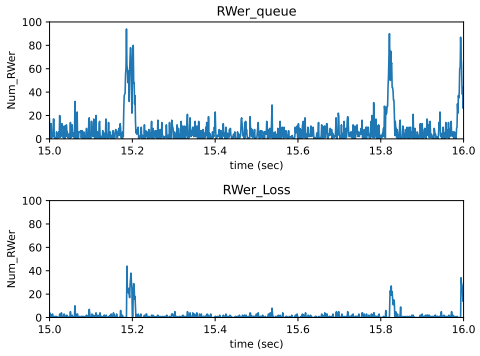

図 5: キューの状態とロスの量 (RQ + RG)

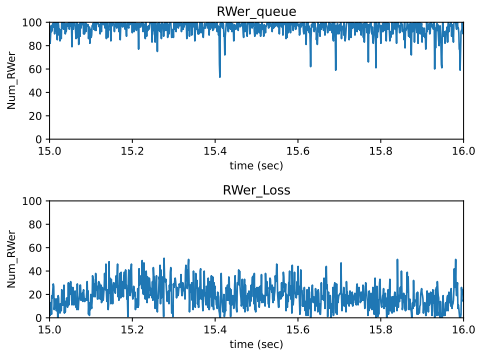

図 6: キューの状態とロスの量 (Tail + RG)

で行った．テールドロップのパラメタについて，$max_{th} = 100$ (キュー長が max_{th} を超えたら破棄)，とする．

3.6 と同様，結果を図 4 に示す．平均値で見たとき，自律的 RWer 輻輳制御機構は，Tail + RG と比べ，約 1 秒早く，総追加実行数が約 120,000 回少なった．結果として，実行時間，総追加実行数の両方の観点でテールドロップに対する RQ の優位性が示された．RQ は テールドロップと違い，キューが満杯になる前から RWer を破棄し始めるため，輻輳を未然に検知しやすくなり，総追加実行数が少なくなる．そして，RWer 生成スピードの低下のデメリットよりも，総追加実行数減少のメリットの方が上回るため，実行時間が少なくなる．

4.6 RQ とテールドロップのキューの状態と Random Walker ロスの量

それぞれのサーバが同時に RW を開始し，全頂点から 100,000 回 RW を実行し，このときの RWer キューの状態と RWer ロスの量を RQ + RG，Tail + RG で比較した．図 5 が RQ + RG，図 6 が Tail + RG の結果を示す．今回は RWer が溜まりやすいサーバ 3 の結果のみ掲載する．

キューの平均長は RQ + RG，Tail + RG でそれぞれ 7.5，95.1 だった．RQ + RG では，Tail + RG に比べ，キューの平均長が小さく，RWer ロスも少ないことが示された．これにより，RQ を導入することによって輻輳を未然に検知し，防ぐことができることがわかった．また，図 6 に示すように，Tail + RG では，キュー長が常に上限付近で推移していることがわかる．このような状態だと，バースト的な RWer の到着が発生したとき，大量の RWer を破棄することになるので，不安定であると言え

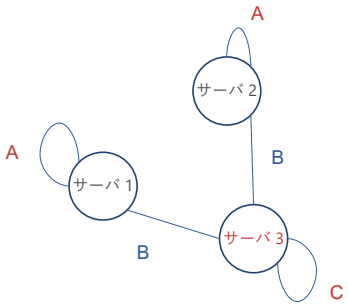

図 7: 実験 4.7 で使用するグラフの生成パラメタ

る．対して RQ + RG では， キュー長が上限よりも下の方で安定しているので，たとえ突然のバースト的な RWer の到着が発生したとしても， すぐに大量の RWer を破棄するという状況にはならない．

4.7 グラフトポロジを変化させたときの Random Walk の総追加実行数

図 7 に示すようなグラフトポロジ (それぞれのサーバが 1,000 頂点， 合計 3,000 頂点) を用いて， 実験を行った．グラフ生成には 4.3 で紹介した SBM [9] を使った．この実験では， 全頂点から 1,000 回分の RW が終了するまで実行を続け， そのときの総追加実行数を記録した．

実行に使用する手法は RQ + RG， Tail + RG であり， 図 7 の A : B， B : C をそれぞれ 1 : 2， 1 : 5， 1 : 10 と変化させ， その組み合わせ全て (9 通り) で実験を行った．A に対して B を大きくすることで， サーバ 1， 2 から サーバ 3 へ遷移する RWer を増やすことができ，B に対して C を大きくすることで， サーバ 3 から サーバ 3 へ遷移する RWer を増やすことができる．これを調整することによって， サーバ 3 の混雑具合を調整する．

表 1a が RQ + RG， 表 1b が Tail + RG での結果を示す．A に対し B が大きくなる， つまりサーバ 1， 2 から サーバ 3 へ遷移する RWer の量が増えると，総追加実行数は大きくなることがわかった．しかし， B に対し C が大きくなる， つまりサーバ 3 から サーバ 3 へ遷移する RWer の量が増えたときは，総追加実行数が大きくなる場合と小さくなる場合が混在していることがわかる．これは，サーバ 3 に滞在する， サーバ 3 で生成された RWer の量が増えるため，RQ からの直接通知による RWer ロス検知がしやすくなるからである．

また， B に対して C が大きくなったときの総追加実行数の変化について， Tail + RG ではあまり増減しないのに対し RQ + RG では顕著な減少傾向が見られることから， Tail + RG よりも RQ + RG に対しての方が， RQ からの直接通知による RWer ロス検知の効果が大きいと読み取れる．

表 1: Random Walk の総追加実行数 (×10000)

(a) RQ + RG

	B : C		
A : B	1 : 2	1 : 5	1 : 10
1 : 2	2.0	3.2	2.3
1 : 5	8.9	7.1	3.3
1 : 10	14.6	6.2	3.7

(b) Tail + RG

	B : C		
A : B	1 : 2	1 : 5	1 : 10
1 : 2	13.6	17.6	17.3
1 : 5	21.8	22.2	22.5
1 : 10	25.6	24.1	26.1

5. おわりに

本研究では， 不均等な分散グラフ上での， RWer 単位の非同期処理において，各サーバが RWer の流量を自律的に調整する機構として， 自律的 RWer 輻輳制御機構を提案した．

本機構は，RWer 単体の価値の低さ，冗長なサーバ間通信に注目し，1 RWer に対し 1 メッセージとして， UDP 通信を使用しメッセージの受け渡しを行なう．また， RED を適用し，RWer の破棄を許容しながらサーバの RQ を管理する．RED を用いたキュー管理と RWer ロス検知による輻輳の初期検知，そして sleep を用いた RWer 生成速度調整を行い，バースト的な RWer 生成を防ぐ．

不均等に分割されたグラフ上で，合計 3,000,000 回分の RW が終了するまでの実行時間は，キュー管理をテールドロップにした場合と比べ，約 1 秒早く， RWer ロスに起因する追加実行数が約 120,000 回少なくなった．実験結果から本機構の導入によって， RW の非同期処理において，高スループットかつ低 RWer 廃棄数を達成できることがわかった．

文　献

[1] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Trans. Netw.*, 1(4):397–413, Aug 1993.

[2] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, page 135–146, New York, NY, USA, 2010. Association for Computing Machinery.

[3] Ke Yang, MingXing Zhang, Kang Chen, Xiaosong Ma, Yang Bai, and Yong Jiang. Knightking: A fast distributed graph random walk engine. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 524–537, New York, NY, USA, 2019. Association for Computing Machinery.

[4] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Vishwesh Jatala, Keshav Pingali, V. Krishna Nandivada, Hoang-Vu Dang, and Marc Snir. Gluon-async: A bulk-asynchronous system for distributed and heterogeneous graph analytics. In *Proceedings of the 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 15–28, 2019.

[5] Minyang Han and Khuzaima Daudjee. Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems. *In Proceedings of the VLDB Endowment*, 8(9):950–961, May 2015.

[6] Wenfei Fan, Ping Lu, Xiaojian Luo, Jingbo Xu, Qiang Yin, Wenyuan Yu, and Ruiqi Xu. Adaptive asynchronous parallelization of graph algorithms. In *Proceedings of 2018 International Conference on Management of Data*, SIGMOD '18, page 1141–1156, New York, NY, USA, 2018. Association for Computing Machinery.

[7] K. K. Ramakrishnan and Raj Jain. A binary feedback scheme for congestion avoidance in computer networks. *ACM Trans. Comput. Syst.*, 8(2):158–181, May 1990.

[8] Andreas von Bechtolsheim, Lincoln Dale, and Hugh W. Holbrook. Why big data needs big buffer switches. 2016.

[9] Paul W Holland, Kathryn Blackmond Laskey, and Samuel Leinhardt. Stochastic blockmodels: First steps. *Social networks*, 5(2):109–137, 1983.