並列グラフ処理エンジンに対するグラフ圧縮と頂点順序最適化の適用

伊藤　竜一† 　新井　淳也†† 　佐々木勇和† 　鬼塚　　真†

† 大阪大学大学院情報科学研究科　〒 565–0871 大阪府吹田市山田丘　1–5

†† 日本電信電話株式会社　〒 180–8585 東京都武蔵野市緑町　3–9–11

E-mail: †{ito.ryuichi,sasaki,onizuka}@ist.osaka-u.ac.jp, ††arai.junya@lab.ntt.co.jp

あらまし　近年, 様々なデータを収集可能になったことやその蓄積先であるストレージの高容量化により大規模データが一般的になった. そのデータに対して PageRank に代表されるグラフ分析パターンを適用して推薦や検索に利用される機会が増えたことにより, 大規模グラフ構造データを高速に処理する手段が求められている. 特に, 汎用的にグラフ分析パターンを定義できるグラフ処理エンジンレベルでの高速化は重要である. しかしながら, マルチコア・高並列度環境でのグラフ処理では大量のグラフ構造データの読み込みによりメモリ帯域の飽和やキャッシュミスの増加といった問題が発生してしまい, 計算機の処理能力を活かせないことが知られている. そこで本稿では, グラフを圧縮することでこの問題を解決しグラフ処理エンジンの高速化に応用可能な 3 つの提案を行う. まず, 処理速度と圧縮率だけでなく, より多くのユースケースで利用可能な単一ソート済み整数列の圧縮手法を提案する. 次に, グラフ構造データが複数のソート済み整数列として表現できることに着目し, それを圧縮する手法を提案する. 最後に, 圧縮手法に依存しないキャッシュ効率を改善する伸長方法を提案する.

キーワード　整数列圧縮, グラフ圧縮, 頂点順序最適化

1. はじめに

インターネット上のサービスから得られるソーシャルグラフ, Web サイトのリンク関係を表す Web グラフや IoT を構成するセンサネットワークからなるネットワークグラフといったグラフ構造データの大規模化が進んでいる. 例えば, 2016 年 1 月時点での Twitter のアクティブユーザ数は 3 億 1000 万人にも及び [1], ユーザ同士の繋がりやポストとそれに対するコメントや共有といった要素が大規模なグラフ構造を成している. このような大規模グラフ構造データは検索や推薦に有用なデータを含んでいる一方, それは潜在的なものであり, 分析して取り出す必要がある. 分析パターンは目的によって様々であり個別に全ての処理を記述することは困難なため, 一般にグラフ構造データに対する任意の分析パターンを処理可能なグラフ処理エンジンが利用されている. 既に様々なグラフ処理エンジンが提案されているが [2–11], 本稿ではマルチコア環境で動作するもの [6–11] に焦点を当てる. マルチコア環境グラフ処理エンジンでは様々な高速化手法が提案されているが, グラフ構造データに対するアクセス局所性の低さやイテレーションを伴う処理がメインとなることから, メモリ帯域の飽和やキャッシュミス率の増加により高並列環境を活かしきれていないという問題がある [12].

本稿では, メモリ使用量を削減してメモリ帯域の飽和やキャッシュミスを防ぐことでグラフ処理を高速化するための圧縮手法を提案する.

既存のグラフ圧縮手法 [13–15] はディスク上の容量を削減することを目標としているため, 圧縮による高速化より伸長コストが大きくなってしまいメモリ帯域やキャッシュミスの改善には用いることができない. そこで, グラフ構造データは多数の整数列から成る形式で保持されることが多いことに着目し, メモリ帯域やキャッシュミスの改善を望むことができる整数列圧縮に基づいた高速なグラフ圧縮手法を目指す. 既存の整数列圧縮手法は主に単一の整数列を対象としており, また, 長大な整数列を高速に処理するために比較的大きなブロック単位で処理が行われるため, まず初めに速度や圧縮率だけでなく短い整数列も処理可能な柔軟性を持ち合わせた単一ソート済み整数列圧縮手法を提案する. 次にこの単一ソート済み整数列圧縮手法を内部的に利用した, グラフ構造データに直接適用可能な複数ソート済み整数列圧縮手法を提案する. この複数ソート済み整数列圧縮手法では頂点 ID の順序と整数列の関係を捉え, 頂点順序最適化の技術をプリプロセッシングとして利用することで相乗的な効率化を図る. また, 具体的な圧縮手法に依存せず, キャッシュを適切に利用しメモリ帯域を節約するための Map インターフェースを提案する.

評価実験を行い, 提案する圧縮手法と頂点順序最適化を組み合わせることでグラフ構造データの表現に使われる複数ソート済み整数列に対してほとんど伸長処理のオーバーヘッドなくグラフ構造データをメモリ上で扱うことに成功した.

本稿の構成は以下の通りである. 2 章で提案手法の前提となる知識を述べる. 3 章で提案手法を詳説し, 4 章で提案手法の実験とその評価・分析を行う. 5 章で関連する研究を紹介し, 6 章で本稿全体をまとめる.

2. 前提知識

この章では, 提案手法の前提知識を述べる. まず, 2.1 節でグラフ構造データのレイアウトについて, 2.2 節でグラフ圧縮の

ベースとなる整数列圧縮の基本的な考え方を, 2.3 節で計算処理の高速化に用いる SIMD 拡張命令を, 2.4 節でグラフの頂点順序最適化について述べる.

2.1 グラフ構造データのレイアウト

マルチコア環境向けのグラフ処理エンジンの多くがメモリ上でのグラフ構造データのレイアウトに注目することで高速化を図っている [6,10,11]. このことからも分かるように, ディスクと CPU の橋渡しとなるメモリ上にあるデータの扱いは処理性能に大きく関わることが知られている. 最も単純なレイアウトとしてエッジリスト形式 (図 2) が挙げられる. ロードや処理の際に変換処理が不要であるが, 多くのメモリを消費するという問題がある. より効率的なレイアウトとして隣接リスト形式 (図 3) が挙げられる. これは各頂点に隣接している (辺が存在する) 頂点の情報をリストで保持する形式である. 実際には全ての頂点の隣接頂点情報を一つの配列にパッキングし, 別の配列で各頂点に対応するオフセット配列を管理するという実装になる. 結果として |`Data Array`| は辺数 $|\mathbb{E}|$, |`Offset Array`| は頂点数 $|\mathbb{V}|$ となる. グラフ分析パターンの多くは注目頂点とその隣接頂点の情報を用いて処理が行うため, エッジリスト形式と比べて容量を削減しつつデータ利用時の変換コストがかからないことが特徴である.

また, 特殊なアーキテクチャではメモリアクセスを意識したデータの扱いがより重要となる. 一般に, NUMA 環境でリモートメモリへのアクセスは非常に遅延が大きく, 多発すると実行性能が低下することが知られている [11]. Polymer [11] では隣接リスト形式のデータを NUMA ノードごとに分割してから NUMA ローカルなグラフとして再構築することでレイテンシの大きいメモリアクセスを減らして NUMA 環境でも高速な処理を実現している.

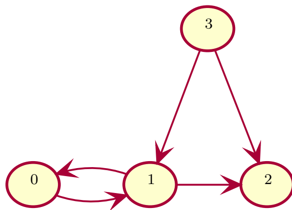

図 1 有向グラフの例

```
Edge Array = { {0, 1}, {1, 0}, {1, 2}, {3, 1}, {3, 2} }
```

図 2 有向グラフ (図 1) のエッジリスト形式

```
Data Array = { 1, 0, 2, 1, 2 }
Offset Array = { 0, 1, 3, 3 }
```

図 3 有向グラフ (図 1) の隣接リスト形式

2.2 整数列圧縮

整数列は汎用的なデータ構造であり様々な箇所で用いられているが, 例えば Web ページの転置インデックス ID でのユースケースあれば非常に大規模なものとなる. このような場合, ディスクや特にメモリ上の容量を節約し IO の削減やキャッシュの有効利用をするため, 整数列を効率よく保持することが重要となる. 整数列圧縮の手法は数多く提案されており [16–24], 一般に変換処理コストと圧縮率のトレードオフとなっている. そのうち, 提案手法のユースケースを前提とした軽量なソート済み整数列に対する圧縮手法を中心に述べる.

ソート済み整数列は D-Gaps [19] と呼ばれる差分リスト形式に変換してから具体的な圧縮が行われることが多い. 元のデータを利用する際には先頭から足し合わせていくことで伸長される. 非常にシンプルなテクニックであるが, 個々の要素を比較的小さい数値にできるため後続の圧縮効率を高めやすいという特徴がある. 一方, データを伸長する際に先頭から軽量でシーケンシャルな操作が必要となるため, 並列環境で扱うには適していないと言える.

具体的な軽量な整数列圧縮にはバイトレベルやビットレベルでのパッキングをメイン戦略とすることが多い. 一般に, プログラムで整数列を扱う際には 32 ビットや 64 ビットといった予め定められたビット長で全ての要素を管理する. しかしながら, 数値的に小さい値はより短いビット長で表現できることが多い. そこで, 要素の数値に合わせてサイズを可変としたコンテナにパッキングすることで圧縮を行う. このアプローチは複雑な符号化を行うことなくある程度の圧縮率を実現可能であり, コンテナやフラグのレイアウトや管理を工夫することで効率化された手法が提案されている [16,20].

応用的な整数列圧縮手法として StreamVByte [20,21]･SIMD-BP128 [18]･SIMD-FastPFor [17] が挙げられる. StreamVByte は可変長コンテナを複数まとめて利用することに加えて SIMD 拡張命令で高速化を行っている手法である. 4 個の要素を 1 ブロックとして処理を行う. SIMD-BP128 はチャンク化されたビットパッキングを主な戦略とし, SIMD 拡張命令で高速化を行っている手法である. 128 個の要素を 1 ブロックとして処理を行う. SIMD-FastPFor は PFor [22] と呼ばれる特殊なビットパッキング手法をベースとしている. 単純に最大値に合わせてパッキングするのではなく, 最大値より少し小さな値に必要なパックサイズを設定し, 一部の収まらない上位ビットを Exception として別領域に保存する. 伸長時には OR 演算で Exception のパッチ処理を行うことで値を復元する. 既存の PFor の実装は多くあるが [17,22–24], この手法は特に SIMD 拡張命令を効率的に利用できるように最適化されており, 既存の手法より優れていると報告されている [17]. いずれの手法も内部的に D-Gap のアプローチを組み込むことができる.

2.3 SIMD 拡張命令 (注1)

一般的なスカラー計算機では, 1 命令で 1 または 2 個のデータを対象に処理が行われる. 一方ベクトル計算機では 1 命令でベクトル化された複数の値を同時に処理することが可能である. そこで, スカラー計算機で部分的にベクトル計算を取り入れる

ために既存の命令セットを拡張したものがSIMD拡張命令である．一般にSIMD専用レジスタを設けることで実現され，レジスタ長に収まる複数の値をスカラー計算機でありながらベクトル計算機のように1命令で処理可能である．そのため，画像フィルタリング処理のような多数の値に同一の演算を適用する必要がある箇所で処理高速化のために利用されている．

2.4 頂点順序最適化

グラフ処理は頂点IDを基準に処理され，また，グラフ構造データは多くの場合ソートされた頂点IDで管理される．このことを踏まえると，隣接頂点が近いほど，言い換えれば隣接頂点同士のIDが近いほど局所性を活かせることが分かる．なお，一見個別に動作する並列環境においても，スレッド単位では同様に局所性を活かすことができる．そこで頂点順序最適化では，グラフ構造的に近い頂点同士が近い頂点IDを持つように頂点IDの振り直しを行う．多数のアルゴリズムが先行研究で提案されているが，理想的な頂点順序最適化はNP-hardであることが知られているため [25]，クラスタリングをベースとして用いるもの [26,27] や兄弟順序を考慮した貪欲な隣接頂点選択 [25] といったヒューリスティックな手法となっている．多くはグラフ処理のプリプロセッシングとして利用され，核となるグラフ処理エンジンや分析パターンの実装を変えることなく処理性能を向上できるという特徴がある．

3. 提案手法

本章では，メモリ帯域節約やキャッシュヒット率向上の面からグラフ処理を高速化するためのグラフ圧縮手法について詳説する．3.1.1節では，2.2節や2.3節で述べた技術を元に，処理速度や圧縮率と整数列長に依存しにくい柔軟な単一ソート済み整数列圧縮手法を提案する．3.1.2節では，3.1.1節で提案する手法をベースにしたグラフ構造データの表現方式の一つでもある複数ソート済み整数列を圧縮する手法を提案する．3.1.2節では，圧縮手法に依存せず，高いキャッシュ効率で伸長操作を可能にするMapインターフェースを提案する．3.2節では，これらの圧縮手法に加えて，グラフ構造の特徴に基づいて2.4節で述べた頂点順序最適化を利用することで相乗的な効率化を考える．

3.1 グラフ構造データに特化した整数列圧縮

提案手法では，現実世界のグラフ構造データとその隣接リスト形式に注目し圧縮を行う．圧縮処理を行わない場合と比べて変換処理のコストがかかるが，グラフ構造に基づいて容量を減らすことでメモリ帯域とキャッシュヒット率を改善し，全体性能の向上を狙う．近年扱われることの多い現実世界のグラフ構造データは頂点次数がPower-Lawに従うことが知られている [28]．つまり，大多数の頂点の次数は低いため，隣接リスト形式で見ると比較的短い整数列が多数で，極一部長大な整数列が含まれることになる．また，一般的なグラフ構造データはランダムに頂点IDが振られていることを考慮すると，隣接頂点のIDは疎であり，単純にD-Gapsを用いても得られる要素の値はあまり小さくならないことが分かる．そこで，本稿では上記のグラフ構造データを隣接リスト形式にした際の特徴を利用した圧縮手法を提案し，更に頂点順序最適化を組み合わせることで疎なグラフ構造データに対しても圧縮率の向上を図る．

3.1.1 単一ソート済み整数列圧縮

まず単一の整数列に対する圧縮について考える．先行研究では比較的大きなブロック単位で処理することを前提として高速化を行っているものが多い．しかしながらグラフ構造に現れる整数列一つ一つは非常に短いものが多いため適用できない．そこで，提案手法ではSIMDレジスタの幅に合わせ，8要素を1ブロックとして差分リスト生成とその圧縮を行う．SIMDを効率的に利用するため，D-Gapsでは，一般的な連続要素の差分リストではなく，前ブロックの最後の値，つまり前ブロックの最大値とブロック内の各要素とのブロック単位の差分リストとする．同様にパッキングもブロック単位の差分リスト内の最大値にフィットするサイズに領域を区切り，ビット長情報とともに各要素に対してビットレベルで行う．メモリアクセス回数を減らすため，パッキングが完了してからまとめてストアを行う (Algorithm. 1)．伸長時には先に取り出したビット長情報を元に，SIMDを用いてビットアンパックと差分リストの逆算をブロック単位で行う (Algorithm. 2)．一方整数列長が8未満の場合は，SIMDによるブロック単位の処理が行えないこと，また，一般的なキャッシュラインサイズに収まることから，エンコード/デコードの処理時間を優先し，D-Gapsを適用した後に予め定めた16ビットと32ビットのどちらかのサイズの整数としてそのフラグ情報とともにパックする (Algorithm. 3)．伸長時は先に取り出したフラグ情報を元に16または32ビット長の値として取り出し，差分リストの逆算を行う (Algorithm. 4)．

3.1.2 複数ソート済み整数列圧縮

次に，この単一整数列圧縮手法をベースとして，複数の整数列に対する圧縮を提案する．先も述べた通り，現実世界のグラフ構造に現れる整数列は短いものが大半を占める．そこで，短い整数列は複数連結することで効率的な圧縮を行う．連結することでソート済みという特徴が失われD-Gapsによる数値の縮小が行えないため，先行研究で提案されているSIMD-FastPFor [17] を用いて圧縮を行う．一方，長い整数列は前述の提案手法で単一整数列として圧縮を行う (Algorithm. 5)．メモリアクセスの効率化のため連結に関するメタ情報は付与していないため，伸長時はオフセット配列を読み直しを行う (Algorithm. 6)．予めヒューリスティックに決められた閾値に基いてアダプティブに圧縮手法を切り替えながら多数の整数列を圧縮・伸長することで，圧縮率と伸長の速度の両立を図る．

3.1.3 Mapインターフェース

これらの提案手法では単純な圧縮・伸長処理だけでなく，伸長操作の一つとして**Mapインターフェース** (注2) を提供する．従来の整数列圧縮手法はあくまで圧縮と伸長操作のみに注目し，その圧縮率と速度が追求されていた．しかしながら，長大な整数列を対象とする場合には本稿の最終目的のようなメモリ帯域削減が実現できない．圧縮済み整数列は確かに元の整数列より

(注1)：本稿では，特にIntelによるx86向け拡張であるAdvanced Vector Extensions (Intel®AVX) という文脈で用いることとする：
`https://software.intel.com/en-us/isa-extensions/intel-avx`

表 1 Algorithm で使用されている記号の定義

$ints$	対象となる単一ソート済み整数列
$ints_list$	対象となる複数ソート済み整数列
$packed$	圧縮された $ints$ または $ints_list$
$consumed$	$packed$ のサイズ
$calc_flag_size$	フラグ保存に必要なサイズ
$calc_pack_size$	数値を表現するために必要なビット数
$write_pack_size$	パックのビット長を書き込む
$read_pack_size$	パックのビット長を読み込む
$THRETHOLD$	整数列を連結して扱うかどうかの閾値
$encode_single$	Algorithm.1
$decode_single$	Algorithm.2
$encode_multiple$	非ソート済み整数列圧縮
$decode_multiple$	非ソート済み整数列伸長
$mm_$	SIMD 命令
$set_zero()$	レジスタを 0 にセットする
$load(p)$	メモリ p からレジスタに読み込む
$store(p, v)$	レジスタからメモリ p に書き出す
$broadcast(v)$	v でレジスタを fill する
$and(v0, v1)$	論理演算 (AND)
$or(v0, v1)$	論理演算 (OR)
$shiftr(v, c)$	右 c ビットシフト
$shiftl(v, c)$	左 c ビットシフト
$add(v0, v1)$	加算
$subtract(v0, v1)$	減算

コンパクトになり伸長操作終了時点までのメモリ帯域は節約できるが, 整数列が長大な場合は伸長が進むに連れて伸長後の整数列が前から順々にキャッシュから落ちてしまう. 全ての伸長が完了して整数列データを利用する計算が行われる時には再度メモリからキャッシュ, レジスタへのロードが必要となり, 結果として圧縮未使用時よりメモリ帯域を消費してしまうという問題がある. 提案する Map インターフェースによって圧縮済み整数列に対する先頭からのシーケンシャルな伸長操作と同時に伸長後の要素を利用した操作を可能にすることでキャッシュの利用効率向上を図る. なお, この手法は具体的な伸長操作ではなく伸長のためのインターフェースであるため, 他の整数列圧縮手法にも組み込むことができる.

3.2 頂点順序最適化の適用

2.4 節で述べた通り, 頂点順序最適化とはグラフ構造的に近い頂点に近い頂点 ID を振り直す手法である. グラフ処理では頂点 ID に基づいた順序で処理が進むため, 結果として局所性が高まり, キャッシュヒット率が向上して処理の高速化につながる. 提案手法では, プリプロセッシングとして入力データに頂点順序最適化を行うことでキャッシュヒット率を上げるだけでなく, 2.2 節で述べた D-Gaps との相乗効果で圧縮の効率化を図る.

4. 評価実験

提案手法を実装し, その実行速度と圧縮率についての評価を行った. また, 既存の頂点順序最適化手法である RabbitOrder [27] と Gorder [25] を用いることで, 頂点順序最適化

Algorithm 1 単一ソート済み整数列の圧縮 $(>= 8)$

Input: $*ints, *packed$
Output: $*packed, consumed$
1: $prev \leftarrow mm_setzero()$
2: $reg \leftarrow mm_setzero()$
3: $consumed \leftarrow calc_flag_size(ints)$
4: $n_used_bits \leftarrow 0$
5: $n_blocks \leftarrow \lceil |ints|/8 \rceil$
6: **for** $i = 0 \rightarrow n_blocks$ **do**
7: $curr \leftarrow mm_load(ints[i])$
8: $diff \leftarrow mm_subtract(curr, prev)$
9: $pack_size \leftarrow calc_pack_size(diff[-1])$
10: $write_pack_size(pack_size)$
11: **if** reg is full **then**
12: $mm_store(store.next(), reg)$
13: $consumed \leftarrow consumed + 32$
14: $reg \leftarrow diff$
15: $n_used_bits \leftarrow pack_size$
16: **else**
17: $reg \leftarrow mm_or(reg, mm_shiftl(diff, n_used_bits))$
18: $n_used_bits \leftarrow n_used_bits + pack_size$
19: **end if**
20: $prev \leftarrow mm_broadcast(curr[-1])$
21: **end for**
22: $consumed \leftarrow consumed + 32$
23: **return** $packed, consumed$

Algorithm 2 単一ソート済み整数列の伸長 $(>= 8)$

Input: $*packed, *ints$
Output: $*ints$
1: $prev \leftarrow mm_setzero()$
2: $reg \leftarrow mm_load(ints[0])$
3: $n_used_bits \leftarrow 0$
4: $n_blocks \leftarrow \lceil |ints|/8 \rceil$
5: **for** $i = 0 \rightarrow n_blocks$ **do**
6: $pack_size \leftarrow read_pack_size(i)$
7: **if** reg is full **then**
8: $reg \leftarrow mm_load(packed.next())$
9: $diff \leftarrow mm_shiftr(reg, n_used_bits)$
10: $n_used_bits \leftarrow pack_size$
11: **else**
12: $diff \leftarrow mm_and(mask[pack_size],$
13: $mm_shiftr(reg, n_used_bits))$
14: $n_used_bits \leftarrow n_used_bits + pack_size$
15: **end if**
16: $curr \leftarrow mm_add(diff, prev)$
17: $mm_store(ints[i], curr)$
18: $prev \leftarrow mm_broadcast(curr[-1])$
19: **end for**
20: **return** $ints$

とグラフ圧縮の相乗効果を評価した. グラフ構造データのサイズや傾向による差異を見るため, 表 2 のデータセットを利用した. なお, 頂点順序最適化の効果を明確にするため予め頂点順序のランダム化を行っている.

実験環境として, 物理 4 コア 4 スレッド・動作周波数 3.30GHz である `Intel Core i5-6600` が搭載され, メモリが計 16GB であるマシンを利用した. ソフトウェア環境として, OS に `Fedora 25 (kernel-4.9.6)`, コンパイラに `g++7.0.1` を最適化

(注2) : 関数型プログラミングにおける Map 操作に由来する

表 2　評価に用いたデータセット

グラフ名	頂点数	辺数	概要
WebGoogle [29–31]	875,713	5,105,039	Web グラフ
LiveJournal [29, 30, 32]	4,847,571	68,993,773	ソーシャルネットワークグラフ
WebBase [14]	118,142,155	1,019,903,190	Web グラフ

Algorithm 3 単一ソート済み整数列の圧縮 (< 8)

```
Input: *ints, *packed
Output: *packed, consumed
 1: prev ← 0
 2: consumed ← 0
 3: for i = 0 → |ints| do
 4:     packed| = (ints[i] − prev) << consumed
 5:     if ints[i] − prev <= 0xFFFF then
 6:         consumed ← consumed + 16
 7:         Write_pack_size(16)
 8:     else
 9:         consumed ← consumed + 32
10:         write_pack_size(32)
11:     end if
12:     prev ← ints[i]
13: end for
14: return packed, consumed
```

Algorithm 4 単一ソート済み整数列の伸長 (< 8)

```
Input: *packed, *ints
Output: *ints
 1: prev ← 0
 2: consumed ← 0
 3: for i = 0 → |ints| do
 4:     if read_pack_size(packed, i)means16 then
 5:         diff ← packed >> consumed & 0xFFFF
 6:         ints[i] ← prev + diff
 7:     else
 8:         diff ← packed >> consumed & 0xFFFFFFFF
 9:         ints[i] ← prev + diff
10:     end if
11:     prev ← ints[i]
12: end for
13: return ints
```

オプション-03 とともに利用した.

4.1　単一ソート済み整数列に対する評価

(Algorithm. 1〜 4) (Prop) からなる単一ソート済み整数列の圧縮・伸長性能を計測した.

比較手法として, 2.2 節で述べた StreamVByte (SVB) [20, 21](注3)・SIMD-BP128 (BP) [18](注4)・SIMD-FastPFor (FPF) [17](注5) を用いた. この実験ではソート済み整数列を扱うため, D-Gaps を有効にした. テストデータとして隣接リスト形式データセットのうち, ソート済みであることが保証されているオフセット配列を利用した. 加えて, 差分が正規分布となる長さ 5 億のソート済み人工データ (ints500m) を利用した.

単一ソート済み整数列圧縮の各手法の伸長速度の比較結果を図 4 に, 単一ソート済み整数列圧縮の各手法の圧縮率の比較結果を図 5 に示す (StreamVByte では内部エラーのため,

Algorithm 5 複数ソート済み整数列の圧縮

```
Input: *ints_list, *offsets, *packed
Output: *packed, consumed
 1: consumed ← 0
 2: consumed ← encode_single(offsets, packed)
 3: i ← 0
 4: while do
 5:     head ← ints_list + offsets[i]
 6:     length ← offsets[i + 1] − offsets[i]
 7:     if length > THRETHOLD then
 8:         consumed ← consumed + encode_single(head, packed)
 9:         i + +
10:     else
11:         acc_length ← 0
12:         while do
13:             if(length >= THRETHOLD
14:               && acc_length >= THRETHOLD)
15:               || i == |ints_list| then
16:               break
17:             end if
18:             length ← offsets[i + 1] − offsets[i]
19:             acc_length ← acc_length + length
20:             i + +
21:         end while
22:         consumed ← consumed
23:           +encode_multiple(head, acc_length, packed)
24:     end if
25:     if i == |ints_list| then
26:         break
27:     end if
28: end while
29: return packed, consumed
```

LiveJournal に対する評価が行えなかった). 図 4 から, いずれのデータセットにおいてもほぼ横並びであったが, 常に提案手法が最も早い結果であった. 一方図 5 から, 提案手法, SIMD-BP128 と SIMD-FastPFor が 5%程度の幅で拮抗している中, StreamVByte が 10%程度悪いという結果が確認された. 内部的な手法が類似している提案手法と SIMD-BP128 が伸長速度・圧縮率の両面で同等に優れている一方で, 提案手法は 8 要素ごと, SIMD-BP128 は 128 要素ごとに処理が可能な点を踏まえると, 提案手法の方が性能を保ちながらより柔軟な単一ソート済み整数列圧縮手法だと言える.

4.2　複数ソート済み整数列に対する評価

実際にグラフ処理エンジンで用いられる隣接リスト形式を想定した複数のソート済み整数列に対する評価実験を行った. 提案手法における整数列連結のための閾値はヒューリスティックに 64 と設定している. また, RabbitOrder と Gorder による頂点順序最適化を施したデータを用意し, 各圧縮手法の性能に与える影響を計測した. 比較手法には前節と同じく StreamVByte

Algorithm 6 複数ソート済み整数列の伸長

Input: $*packed, *ints_list, *offsets$

Output: $*ints_list, *offsets$

1: $decode_single(packed, |ints_list|, offsets)$
2: $i \leftarrow 0$
3: **while** **do**
4: $head \leftarrow ints_list + offsets[i]$
5: $length \leftarrow offsets[i+1] - offsets[i]$
6: **if** $length > THRETHOLD$ **then**
7: $decode_single(head, packed)$
8: $i{+}{+}$
9: **else**
10: $acc_length \leftarrow 0$
11: **while** **do**
12: **if**$(length >= THRETHOLD$
13: $\&\&\ acc_length >= THRETHOLD)$
14: $||\ i == |ints_list|$ **then**
15: **break**
16: **end if**
17: $length \leftarrow offsets[i+1] - offsets[i]$
18: $acc_length \leftarrow acc_length + length$
19: $i{+}{+}$
20: **end while**
21: $consumed \leftarrow consumed$
22: $+decode_multiple(head, acc_length, packed)$
23: **end if**
24: **if** $i == |ints_list|$ **then**
25: **break**
26: **end if**
27: **end while**
28: **return** $ints_list, offsets$

(SVB)・SIMD-BP128 (BP)・SIMD-FastPFor (FPF) を用いた．なお, 比較手法はオフセット配列とデータ配列を全て連結して単一未ソート整数列として扱うことになるため Map インターフェースが利用できず, 本研究で目標としているメモリ帯域の改善といったことは行えないためあくまで参考数値である．

複数ソート済み整数列圧縮の各手法の伸長速度の比較結果を図 6 に, 複数ソート済み整数列圧縮の各手法の圧縮率の比較結果を図 7 に示す (StreamVByte では内部エラーのため, WebBase のオリジナルデータと RabbitOrder を施したデータに対する評価を行えなかった)．図 6 から, 700〜2000(mis) で伸長を行えることが確認された．また図 7 から, 50〜80%程度の圧縮率が確認された．特に提案手法を Gorder と組み合わせることで頂点順序最適化を用いない提案手法と比較して最大 12%程度圧縮率を改善することができた．

4.3 Map インターフェースによる効率化の評価

Map インターフェースによる処理性能の評価実験を行った．対象となる単一ソート済み整数列内要素の合計値計算を目標タスクとし, Map インターフェースを利用して伸長しながら逐次与えられたタスクの計算を行う場合と全て伸長してから与えられたタスクの計算を行う場合を実行速度・メモリアクセス・キャッシュミス数の観点から比較した．テストデータには 4.1 節

(注3)：https://github.com/lemire/streamvbyte
(注4)：https://github.com/lemire/SIMDCompressionAndIntersection
(注5)：https://github.com/lemire/FastPFor

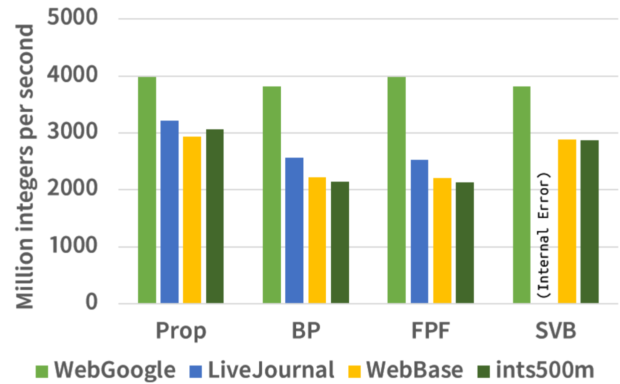

図 4　単一ソート済み整数列圧縮手法の伸長速度による比較

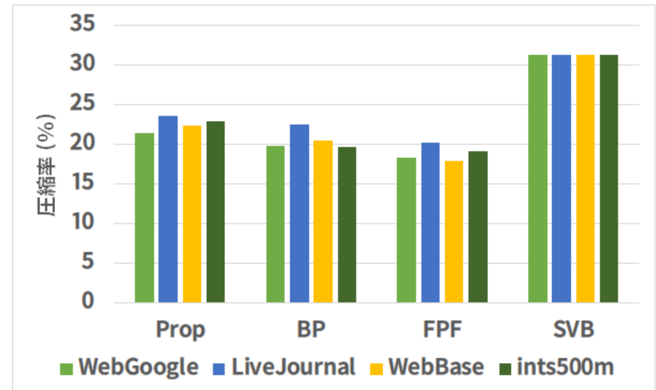

図 5　単一ソート済み整数列圧縮手法の圧縮率による比較

で利用したデータのうち, WebBase と ints500m を利用した．

目標タスクを実行する際に, ナイーブに実行する場合と比較して Map インターフェースを利用することによる各メトリックスの削減率を図 8 に示す．図 8 から, メモリリード・メモリライト・L2 キャッシュミス数・L3 キャッシュミス数の観点で 80〜90%程度改善されることが確認され, 実行時間でも 5〜15%程度の高速化に成功した．この実験では実行時間に与える影響は 10%前後であったが, メモリやキャッシュ効率の影響が出やすい並列環境での処理においてはより大きな高速化を望むことができる．

5. 関 連 研 究

5.1 グラフ処理エンジン

グラフ処理エンジンとは, ユーザ定義の分析パターンを任意のグラフ構造データに対して実行するためのミドルウェアである．言い換えると, ユーザは具体的な分析パターンのみを記述すればよく, グラフ構造データのロードや並列実行は全てグラフ処理エンジン内で自動的に行われる．

グラフ処理エンジンのアーキテクチャは分散環境向けとマルチコア環境向けの 2 つに大別される．前者は多数の計算機を並べ, 計算機間で通信しながら処理を行うアプローチを取る．代表的なエンジンとして GraphLab [2, 3], PowerGraph [4] や PowerLyra [5] が挙げられる．一方後者は単一計算機上でマルチコア・マルチプロセッサを共有メモリで利用して処理を行うアプローチを取る．前者に比べ計算機資源が限られること

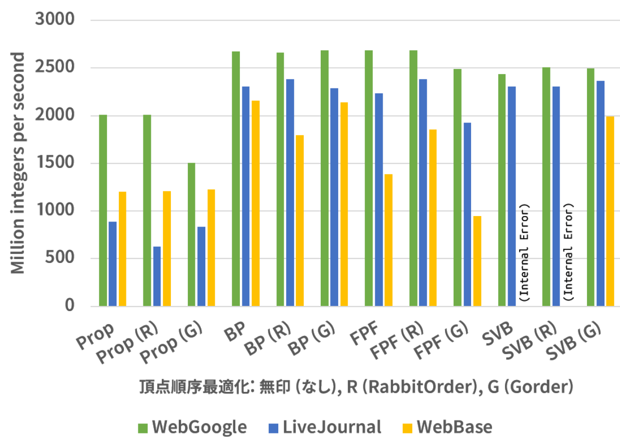

図 6 複数ソート済み整数列圧縮手法の伸長速度による比較と頂点順序最適化の影響

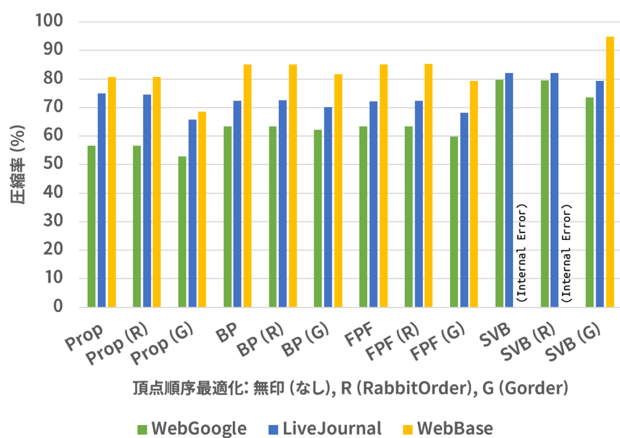

図 7 複数ソート済み整数列圧縮手法の圧縮率による比較と頂点順序最適化の影響

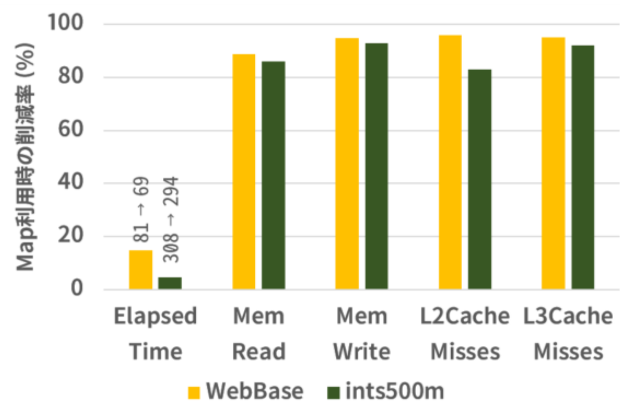

図 8 Map インターフェース利用による単一ソート済み整数列圧縮の効率化
(図中の数値は提案手法適用前後の実行時間 (msec) である)

に起因する処理性能の低さから注目されていなかったが，分散環境とは異なる低レイテンシな共有メモリ環境を活かし，分散環境に近い処理速度を GraphChi [6] が示したことを皮切りに，TruboGraph [7], VENUS [8] や NXGraph [9] といったエンジンが提案されている．いずれの研究も，データ構造や並列処理のアクセスパターンを工夫することでディスク IO 削減と CPU 利用率向上から処理の高速化を図っている．また，更に高並列環境や NUMA 環境に踏み込んだ研究として Ligra [10] や Polymer [11] が挙げられる．

6. ま と め

本稿では，グラフ処理のメモリ帯域節約やキャッシュヒット率向上の面からの高速化を行うための方法として，現実世界のグラフ構造データの性質に基づいた効率的なグラフ圧縮手法と圧縮手法によらないメモリやキャッシュの利用効率の良い伸長インターフェースを提案した．今後の課題として，提案手法の実装レベルでの更なる効率化を行い，最終的にマルチコア環境向けグラフ処理エンジンに組み込んでその性能を計測することが挙げられる．

文 献

[1] Twtiter. Twitter Company | About, 2016.

[2] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *Proceedings of VLDB*, Vol. 5, No. 8, pp. 716–727, aug 2012.

[3] Yucheng Low, Joseph E Gonzalez, Aapo Kyrola, Danny Bickson, Carlos E Guestrin, and Joseph Hellerstein. GraphLab: A new framework for parallel machine learning. *Proceedings of UAI*, jul 2014.

[4] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *Proceedings of OSDI*, pp. 17–30, Hollywood, CA, oct 2012. USENIX.

[5] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of EuroSys*, p. 1, 2015.

[6] Aapo Kyrola, Guy E Blelloch, and Carlos Guestrin. GraphChi: Large-Scale Graph Computation on Just a PC. In *Proceedings of OSDI*, Vol. 12, pp. 31–46, oct 2012.

[7] Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, Jeong-Hoon Lee, Min-Soo Kim, Jinha Kim, and Hwanjo Yu. TurboGraph: a fast parallel graph engine handling billion-scale graphs in a single PC. In *Proceedings of SIGKDD*, pp. 77–85, aug 2013.

[8] Jiefeng Cheng, Qin Liu, Zhenguo Li, Wei Fan, John C S Lui, and Cheng He. VENUS: Vertex-Centric Streamlined Graph Computation on a Single PC. In *Proceedings of ICDE*, apr 2015.

[9] Yuze Chi, Guohao Dai, Yu Wang, Guangyu Sun, Guoliang Li, and Huazhong Yang. NXgraph: An Efficient Graph Processing System on a Single Machine. *Proceedings of ICDE*, may 2016.

[10] Julian Shun and Guy E Blelloch. Ligra: a lightweight graph processing framework for shared memory. *ACM SIGPLAN Notices*, Vol. 48, No. 8, pp. 135–146, 2013.

[11] Kaiyuan Zhang, Rong Chen, and Haibo Chen. NUMA-aware graph-structured analytics. *ACM SIGPLAN Notices*, Vol. 50, No. 8, pp. 183–193, 2015.

[12] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. Challenges in parallel graph processing. *PPL*, Vol. 17, No. 01, pp. 5–20, 2007.

[13] Gregory Buehrer and Kumar Chellapilla. A scalable pattern mining approach to web graph compression with communities. *Proceedings of the international conference on Web search and web data mining - WSDM '08*, p. 95, 2008.

[14] Paolo Boldi and Sebastiano Vigna. The WebGraph Frame-

work I: Compression Techniques. In *Proceedings of WWW*, pp. 595–601, Manhattan, USA, 2004. ACM Press.

[15] Yongsub Lim, U. Kang, and Christos Faloutsos. SlashBurn: Graph compression and mining beyond caveman communities. *IEEE Transactions on Knowledge and Data Engineering*, Vol. 26, No. 12, pp. 3077–3089, 2014.

[16] Jimmy Lin and Chris Dyer. *Data-Intensive Text Processing with MapReduce*. Morgan & Claypool Publishers, 2010.

[17] Daniel Lemire and Leonid Boytsov. Decoding billions of integers per second through vectorization. *SPE*, Vol. 45, No. 1, pp. 1–29, 2015.

[18] Daniel Lemire, Leonid Boytsov, and Nathan Kurz. SIMD compression and the intersection of sorted integers. *SPE*, 2014.

[19] Ian H Witten, Alistair Moffat, and Timothy C Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, 1999.

[20] Jeffrey Dean. Challenges in building large-scale information retrieval systems. In *Proceedings of WSDM (invited talk)*, 2009.

[21] Jeff Plaisance, Nathan Kurz, and Daniel Lemire. Vectorized vbyte decoding. *Proceedings of iSWAG*, 2015.

[22] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. Super-scalar RAM-CPU cache compression. In *Proceedings of ICDE*, p. 59, 2006.

[23] Vo Ngoc Anh and Alistair Moffat. Inverted index compression using word-aligned binary codes. *IJIRR*, Vol. 8, No. 1, pp. 151–166, 2005.

[24] Jiangong Zhang, Xiaohui Long, and Torsten Suel. Performance of compressed inverted list caching in search engines. In *Proceedings of WWW*, pp. 387–396, 2008.

[25] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. Speedup Graph Processing by Graph Ordering. In *Proceedings of SIGMOD*, SIGMOD '16, pp. 1813–1828, New York, NY, USA, 2016. ACM.

[26] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of WWW*, pp. 587–596, 2011.

[27] Arai Junya, Shiokawa Hiroaki, Yamamuro Takeshi, Onizuka Makoto, and Iwamura Sotetsu. Rabbit Order: Just-in-time Parallel Reordering for Fast Graph Analysis. In *Proceedings of IPDPS*, 2016.

[28] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the internet topology. In *Proceedings of ACM SIGCOMM*, Vol. 29, pp. 251–262, 1999.

[29] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data, jun 2014.

[30] Jure Leskovec, Kevin J Lang, Anirban Dasgupta, and Michael W Mahoney. Statistical Properties of Community Structure in Large Social and Information Networks. In *Proceedings of WWW*, pp. 695–704, apr 2008.

[31] Google network dataset - KONECT, oct 2016.

[32] LiveJournal network dataset - KONECT, may 2015.