GPUを用いた大規模な文書に対する高精度検索のための高速な重み付け計算

柳本 晟熙† 欅 惇志†† 宮崎 純††

† 東京工業大学工学部情報工学科 〒 152–8550 東京都目黒区大岡山二丁目 1 2 - 1
†† 東京工業大学情報理工学院 〒 152–8550 東京都目黒区大岡山二丁目 1 2 - 1
E-mail: †yanagimoto@lsc.cs.titech, ††keyaki@lsc.cs.titech.ac.jp, †††miyazaki@cs.titech.ac.jp

あらまし 本研究では, ウェブ上に大量に存在する様々な文書から抽出された索引語に対し, 高精度な情報検索に必要となる重み付け計算を GPU を用いて行う. 並列処理のフレームワークである MapReduce と, MapReduce を GPU 上で実行するためのフレームワークである Mars を利用し, 重み付け計算を行う既存研究は存在するが, GPU のメモリ上に格納できない大規模なデータを想定していない. 本研究では, 大規模なデータに対してもデータを分割して効率的に計算を行うことできる手法を提案し, 評価を行う.

キーワード GPGPU, 情報検索, MapReduce, 索引語重み付け

1. はじめに

現在, 大量のウェブページが存在し, その中からユーザの要求に合致するページを正確に抽出する高精度な情報検索が必要とされている. 高精度な情報検索を実現するために, TF-IDF [1] や BM25 [2] など，さまざまな索引語重み付け手法が提案されている. これらの重み付け手法において用いられる統計量は, ある文書内で出現する索引語の統計量から算出される局所的統計量と, 文書集合全体の索引語の出現統計量から算出される大域的統計量の二種類に分類される.

上記の索引語重みを高速に計算するために, 文書集合を分割し, 並列に計算する研究が取り組まれている [3] [4] [5]. 並列処理を行うための代表的なフレームワークとして Google が開発した MapRedece [6] が存在する. MapRedece はデータセットをクラスタ内のノードに分散させ, 入力データから Key/Value ペアを生成する Map ステップと, それらを集約し, 結果を算出する Reduce ステップの二つのステップに分けて並列計算を行う. MapReduce の並列処理をクラスタ上ではなく, マルチコア CPU 上で実装した Phoenix [7] も存在する.

MapReduce で高速にデータを処理するためには, データを分散した各ノードで高速に処理することが求められる. その解決方法の一つが, GPU を汎用計算に利用する GPGPU (General-purpose Computing on Graphics Processing Units) である. GPU は本来, 画像処理を目的として開発されたが, その並列計算能力の高さから汎用計算に使用する研究がなされてきた. なお, 現在では一般用途のマシンにも GPU が搭載されていることが一般的であるため, 多数のマシンにおいて GPU を用いた並列処理を行うことが可能である. また, GPGPU の一環として, MapReduce を GPU 上で行うためのフレームワークである Mars [8] が存在する. GPU での並列コンピューティング開発環境として CUDA [9] が存在するが, 計算能力を十分に引き出すためには GPU のアーキテクチャに精通している必要がある. 一方, Mars は GPU プログラミングの複雑さをほとんど意識することなく使用することができるよう設計されているため, GPU プログラミングに精通していない人でも容易に扱うことができる.

森谷ら [10] は Mars を拡張し, 文書から抽出された索引語に対して上記の高精度検索用の重み付けを高速に行うことを可能にした. しかし, 一般にグラフィックカード上のメモリである VRAM は, コンピュータのメインメモリよりも容量が小さいため, 一度の MapReduce で扱える文書集合のサイズが限られており, 森谷らの手法では扱うことができる文書集合のサイズに限界がある.

この問題を解決するために, 本研究では大規模な文書集合をチャンクと呼ばれる小さな文書集合に分割して計算を行う. その際, チャンクごとの独立した計算結果では大域的統計量を算出することができない. そこで本研究では, 目的の異なる二種類の MapReduce を提案し, 1 回目の MapReduce でチャンクごとの大域的統計量を集約することで, 文書集合全体における大域的統計量を算出する. 続く 2 回目の MapReduce で索引語の重み付けを完了する. これにより大規模な文書集合に対しても重み付け計算を可能にする.

また, GPUTeraSort [11] は GPU を用いたソートアルゴリズムを提案し, 計算の各ステージをパイプライン処理で行っている. 本研究でも, 計算の各ステージに対してパイプライン処理を施すことによって, 提案手法の高速化を実現する.

本研究では, 大規模な文書集合における索引語の重み付けとして, 前述の BM25 の計算を行い, その評価実験を行う.

2. 関連研究

本節では, はじめに索引語の重み付け計算である BM25 [2] の計算式を述べる. また, 既存研究である GPU を用いた重み付け計算手法 [10] と, GPU を用いてデータベースの莫大なレコードをソートすることを実現した GPUTeraSort [11] について説

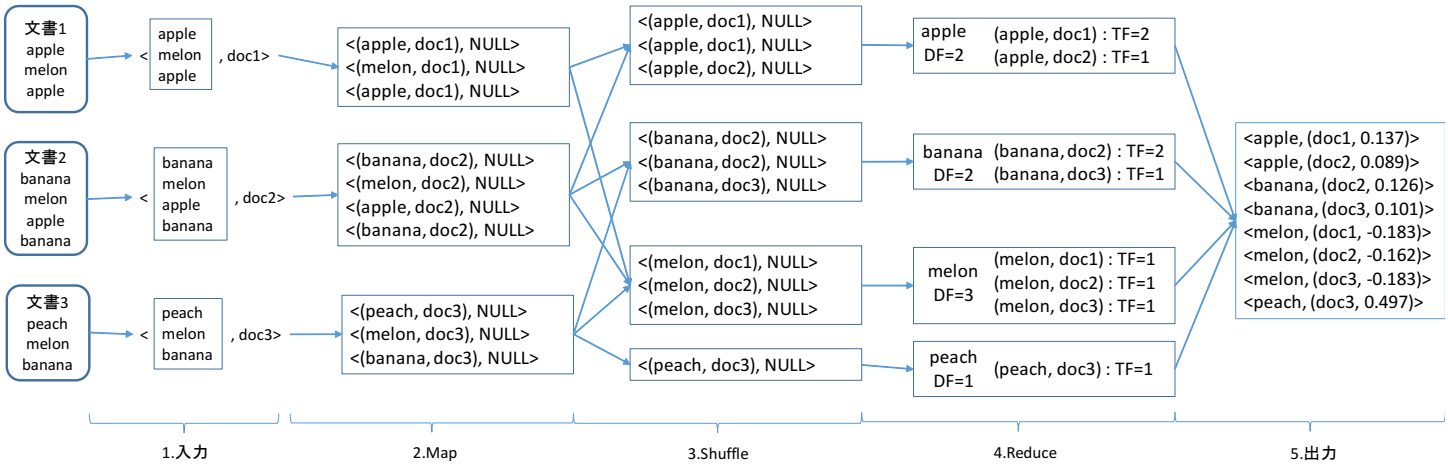

図 1　既存手法のワークフロー

明する．

2.1　BM25

まず, BM25 の計算式を示す．文書 d における索引語 t の重み $w_{d,t}$ は下記の式 (1) で算出できる．

$$w_{d,t} = \frac{(k_1+1)tf_{d,t}}{k_1((1-b)+b\frac{dl_d}{avdl})+tf_{d,t}} \cdot \log\frac{N-df_t+0.5}{df_t+0.5} \quad (1)$$

$$lw_{d,t} = \frac{(k_1+1)tf_{d,t}}{k_1((1-b)+b\frac{dl_d}{avdl})+tf_{d,t}} \quad (2)$$

$$gw_t = \log\frac{N-df_t+0.5}{df_t+0.5} \quad (3)$$

式 (1) 中の $tf_{d,t}$ を文書 d における索引語 t の出現頻度, df_t を索引語 t を含む文書数, N を文書集合全体の文書数, dl_d を文書 d に含まれる索引語の数, $avdl$ を文書集合全体の平均文書長である．また, k_1, b はパラメータであり, 以降それぞれ $k_1 = 1.2, b = 0.75$ と設定する．また, $avdl$ の値はウェブ文書においては頻繁に変化することはないと考えられるため, 既知とする．

ここで, 式 (1) の第一項目を以降局所的重みと呼び, 式 (2) に示し, 式 (1) の第二項目を以降大域的重みと呼び, 式 (3) に示す．

2.2　GPU を用いた重み付け計算手法

本節では森谷らによる, Mars を用いた BM25 を計算する手法 [10] について述べる．この手法の概要を図 1 に示す．

（1）　文書データの入力

BM25 の MapReduce を用いた計算手法を述べる．MapReduce は key/value ペアを入力とする．したがって, ここでは文書 (document) を key, 各文書に 1 対 1 で割り当てられる文書 ID(documentID) を value とする．すなわち, `<document, documentID>`の key/value ペアを入力データとする (図 1 中の 1. 入力)．この時, 文書数 N と文書長 dl_d を求める．入力されたデータは, 続く Map ステップでそれぞれ並列に処理される．なお, 入力作業は CPU 上で行われ, 続く Map ステップから Reduce ステップは GPU 上で行われる．

（2）　Map

Map ステップは, 中間データを key/value ペアで出力する (図 1 中の 2.Map)．key は, 文書から抽出された索引語 (term) と文書 ID の組から構成される．この時, 抽出した索引語を数えることで dl_d が求まる．value は続く Shuffle ステップでは必要としないため NULL とする．すなわち, Map ステップの出力は `<(term, documentID), NULL>`となる．

（3）　Shuffle

Shuffle ステップは Map ステップの出力をユーザーの定義に従って並べ替えるステップである (図 1 中の 3.Shuffle)．ここでは, key/value ペアを key の第一要素である索引語 (term) で辞書順に並べ替え, 索引語が同一のものについては key の第二要素である文書 ID(documentID) で昇順に並べ替える．そして, key に同一の索引語を持つ key/value ペアごとに次の Reduce ステップで並列に処理する．なお, 文書 ID が異なる key/value ペアであっても, 同一の索引語を持つ key の場合は, Reduce ステップでは同一のスレッドで処理することで効率的に処理を行う．

（4）　Reduce

Reduce ステップは Shuffle ステップの結果を集約し, 最終結果である BM25 を算出するステップである (図 1 中の 4.Reduce)．Shuffle ステップの出力データを 2 回走査することで文書 d における索引語 t の出現頻度 $tf_{d,t}$ と文書集合中における索引語 t の文書頻度 df_t を算出する．以上で BM25 の算出に必要な値を全て得ることができたため, Reduce ステップ内で索引語の重み $w_{d,t}$ を算出する．最終的な Reduce ステップの出力は`<term, documentID>, BM25>`となる．

（5）　結果の出力

Reduce ステップで得られた結果を．`<term, documentID, BM25>`の形でファイルに出力する (図 1 中の 5. 出力)．なお, この出力作業は CPU により行われる．

2.3　GPUTeraSort

本節では Govindaraju ら [11] による, GPU 上でデータベースの莫大なレコードをソートすることを実現した GPUTeraSort について述べる．GPUTeraSort はメモリ集約的, 計算集約的な処理を GPU 上で行い, ディスク I/O やリソース管理を CPU 上で行うことで高速なソートを実現している．また, GPUTeraSort は下記の五つのステージから構成される．入力データは一度に VRAM に格納できないため, 分割して処理を行う．分割された各データをチャンクと呼ぶ．各チャンクに対して, 下記 (1)Reader から (5)Writer 五つのステージを繰り返し行う．その際, 高速化のために各ステージはパイプライン処理で行われる．

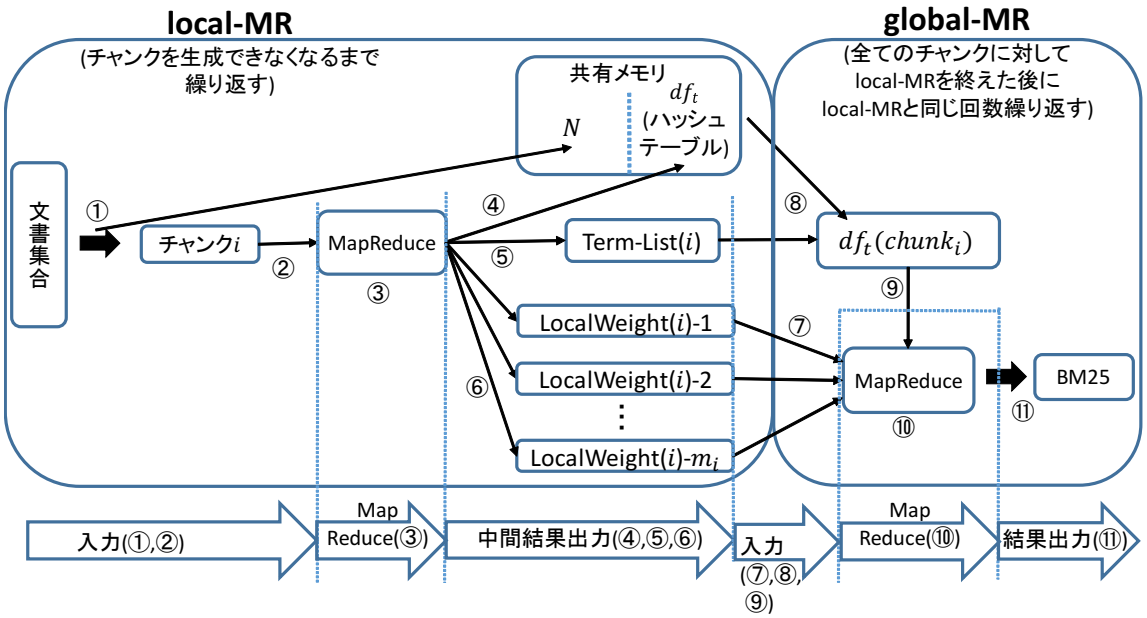

図 2　提案手法のワークフロー

（1） Reader

Reader ステージでは, ソートする対象のデータの各チャンクを順にメインメモリに読み込む. この時, 入力データをストライピングされたディスクに書き込んでおくことで, ディスク I/O のバンド幅が向上する.

（2） Key-Generator

Key-Generator ステージでは, ソートする各レコードのポインタに対応する Key を用意し, (Key, Record-pointer) ペアを生成する. このステージは計算集約的ではないが, メモリを多く使用するためメモリ集約的な処理である.

（3） Sorter

Sorter ステージでは, 前のステージで生成したペアを VRAM へ転送し, ソートを行う. このステージは計算集約的であり, メモリ集約的でもある. ソート結果は VRAM からメインメモリに転送される.

（4） Reorder

Reorder ステージでは, 前のステージでソート済みの (Key, Record-pointer) ペアにならい, 実際にレコードを並べ替える. この並べ替えられたデータを run と呼ぶ.

（5） Writer

Writer ステージでは, 前のステージで生成された run をディスクに書き込む.

（6） run の集約

以上の五つのステージを全てのデータに対して実行した結果, 複数の run がディスクに書き込まれる. これらの run をマージすることにより, 全データがソートされた最終結果が得られる.

なお, GPUTeraSort のソートアルゴリズムにはバイトニックソートを用いている. バイトニックソートは, 昇順のデータ列と降順のデータ列が交互に出現するバイトニック列を生成しながらソートを行うアルゴリズムである. データを分割してソートを行うことが可能であるため, GPU の得意とする並列計算と親和性が高い. また, GPU の苦手する条件分岐を行わない点においても相性が良い. したがって, Mars もソートアルゴリズムにバイトニックソートを採用している.

GPUTeraSort の性能は様々な要因に左右される. 使用する GPU の性能はもちろん, メインメモリのバンド幅やディスク I/O 性能にも影響を受ける. また, パイプラインの各ステージの負荷分散を適切に行うことでスループットが向上する. ソート対象のデータベースのサイズが極端に小さい場合を除き, データを VRAM へ転送する時間は計算時間と比較して相対的に小さくなる. また, チャンクサイズを変化させることにより, 全体の実行時間も変化する. 5 ステージから構成されるパイプライン処理を行っているため, チャンクサイズの理論上の最大値はメインメモリサイズの 1/5 となる.

3. 提 案 手 法

本研究は, 既存研究である Mars を用いて BM25 を算出する手法を拡張し, 既存研究では扱うことができない大規模な文書集合に対しても効率よく BM25 を計算する手法を提案する.

既存研究では, 文書集合全体を一度に VRAM へ転送して計算を行うが, 本研究では, 文書集合の全体を一度に VRAM に転送して計算することができないような, 大規模な文書集合を対象とする. そこで, 文書集合を複数のチャンクに分割して計算を行う. チャンク一つあたりのサイズを大きくすることで, 中間結果を出力するためのディスク I/O や, VRAM にデータを転送するためのオーバーヘッドは小さくなる. しかし, 一度に多くのデータをソートするため, 計算量は大きくなる. 一方, チャンクサイズを小さくすると, オーバーヘッドは大きくなるが, ソートの計算量は小さくなる. 特に VRAM とメインメモリ間でデータを転送するためのオーバーヘッドが相対的に大きくなる. したがって, 中間結果を出力するためのディスク I/O, VRAM へのデータ転送のオーバーヘッドとデータをソートする計算量にはトレードオフの関係がある. また, 提案手法では役割の異なる二つの MapReducce を提案し, それぞれ local-MR と global-MR とする. ここで, 提案手法の概要を図 2 に示す.

local-MR と global-MR はそれぞれ図 2 中青枠で囲まれている処理を指す.

3.1 local-MR

local-MR は, 全てのチャンクに対して逐次的に 1 回ずつ行われる. 図 2 の①から⑥を 1 回の local-MR の入力から出力とし, 全てのチャンクに対して繰り返す. 1 回の local-MR の目的は, チャンク i 内で索引語 t を含む文書数 $df_{t,i}$ とチャンク i に含まれる文書数 N_i, 更に BM25 を構成する式 (2) の局所的重み $lw_{d,t}$ を算出することである. 局所的重み $lw_{d,t}$ は局所的統計量で構成されるため, チャンクごとに独立に計算を行う.

それに対して, 大域的重み gw_t は大域的統計量であるため, 文書集合全体に対して計算しなければならない. そのため, 各チャンクの local-MR が完了する度に $df_{t,i}$ と N_i を集約する. その結果, 全てのチャンクに対する local-MR が完了した時点で, 文書集合全体における df_t と N の算出が完了する.

以降は, local-MR で上述の df_t と N, $lw_{d,t}$ を求める手法を示す.

（1） 文書集合の分割と入力

文書集合を予め設定したチャンクのサイズに分割して計算を行うため, 読み込む文書のサイズの合計が, 設定したチャンクサイズを上回らない最大の文書ファイル数をメインメモリに読み込む (図 2 中の①). この時, 読み込んだ文書数をメインメモリ内の共有領域に保持しておく. これを, チャンク i における文書数 N_i とする. 新たにチャンクを生成するたびに N_i を足し合わせていくことで, 全ての local-MR が完了した時点で, 文書集合全体の文書数 N が求まる (図 2 中の①).

続く MapReduce は key/value ペアを入力とする. 既存手法と同様, key を文書, value を文書に 1 対 1 で与えられる文書 ID とする. すなわち, `<document, documentID>`を MapReduce の入力とする (図 2 中の②).

（2） MapReduce

前のステップで生成したチャンクに対して, $df_{t,i}$ と N_i を算出する MapReduce を行う (図 2 中の③).

Map ステップは, 文書から索引語を抽出する. 抽出した索引語を数えることで dl_d が求まる. そして, 索引語 (term) と文書 ID(documentID) のペアを key とする. value は Shuffle ステップで必要ないため, NULL とする. すなわち, Map ステップの出力は`<(term, documentID), NULL>`となる.

Shuffle ステップは key/value ペアを, key(term, documentID) の第一要素である索引語 (term) で辞書順に並べ替え, 索引語が等しい key については文書 ID で並べ替える.

Reduce ステップは同じ索引語を key に含むペアごとに並列に行われる. key を走査し, dcumentID に重複がないようにカウントすることで, 現在扱っているチャンク i における索引語 t を含む文書数 $df_{t,i}$ が求まる. また, 同一の documentID を持つ key をカウントすることで $tf_{d,t}$ が求まる. 以上で, $lw_{d,t}$ の算出に必要な値が全て求まったため, Reduce ステップで算出する. こうして, Reduce ステップの出力は`<(term, documentID), (`$df_{t,i}$`, `$lw_{d,t}$`)>`の key/value ペアとなる.

（3） 中間結果の出力

本節では, local-MR の結果を保持する方法について述べる.

$df_{t,i}$ は索引語 t を key とし, $df_{t,i}$ を value とした key/value ペアと考えることができる. 本研究では, 計算対象をウェブ文書とするため, 文書集合に同一の索引語が複数回出現するものと考えられる. そこで, 索引語 t と $df_{t,i}$ のペアの集合をメインメモリにて保持するためには, 文書集合に出現する全ての索引語を 1 個ずつ格納できるサイズに加え, 索引語の種類と同じ数の整数値 df_t を格納できるサイズが必要となる. 一方, この条件を満たしておらず, メインメモリにて保持できない場合には, 中間ファイルとしてディスクに出力することで解決できる. 本研究では, 前述のペアの集合をメインメモリに格納できる状況を想定し, メインメモリ内に作成するハッシュテーブルにて管理する (図 2 中の④). ハッシュテーブルに登録されていない索引語の $df_{t,i}$ については新たに格納する. 既にハッシュテーブルに登録されている索引語については value である $df_{t,i}$ を加算する. これにより, 全てのチャンクに対して local-MR が完了した時点で, 文書集合全体における索引語 t を含む文書数 df_t が算出される. また, ハッシュテーブルに df_t を格納すると同時に, チャンク i に含まれる索引語集合を辞書順にテキストファイルに出力する. これは, 続く global-MR で用いられ, ファイルの名前を Term-List(i) とする (図 2 中の⑤).

$lw_{d,t}$ については, 索引語と文書 ID のペア (term, documentID) を key とし, $lw_{d,t}$ を value とした key/value ペアと考えることができる. この key/value ペアは文書集合全体における, 索引語とそれを含む文書の組み合わせの数だけ存在するため, 文書集合が大きくなるにつれ, メインメモリに保持するには大きすぎるサイズになることが想定される. そこで, $lw_{d,t}$ は中間ファイルとしてテキストファイルに出力する. 出力の形式は 1 行に `(term, documentID,` $lw_{d,t}$`)` の組を一つ出力する. ここで, global-MR の高速化のために, 中間ファイルは複数に分割して出力する. このチャンク i における j 番目の中間ファイルの名前を LocalWeight(i)-j とし, ファイル数を m_i とする (図 2 中の⑥). LocalWeight(i) は global-MR の入力データとなるが (図 2 中の⑦), Mars は入力データの数だけ並列に Map ステップを行う. そのため, LocalWeight(i) を分割することで並列処理数を増やすことができ, 効率的に MapReduce を行うことができる.

N については, 単一の整数であるため, サイズは極めて小さい. したがってメインメモリ内に保持する.

以上で, local-MR による N, df_t, $lw_{d,t}$ の算出と Term-List の作成が完了した.

3.2 global-MR

global-MR の目的は, local-MR で求めた中間結果から BM25 を算出することである. global-MR は, local-MR で生成されたチャンクの数だけ繰り返し行われる.

（1） 中間結果の入力

はじめに, local-MR が生成した中間ファイル LocalWeight(i)-j を読み込む. ここで, 一度の入力でチャンク i から生成された LocalWeight(i)-1 から LocalWeight(i)-m_i の m_i 個のファイルを全て読み込む (図 2 中の⑦). 続く索引語の重みを計算する MapReduce の入力は, key を LocalWeight(i)-j, value をファイルに 1 対 1 に割り当てられる ID とする. すなわち,

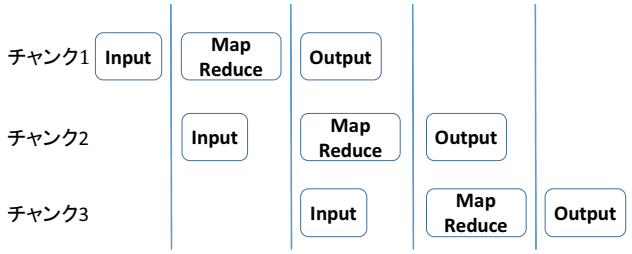

図 3　パイプライン処理の概要

<LocalWeight(i)-j, FileID>である.

続いて, Term-List(i) を参照し, チャンク i に出現する索引語の df_t をハッシュテーブルから取り出し<term, df_t>ペアの集合を生成する. この集合を $df_t(chunk_i)$ とする (図 2 中の⑧). $df_t(chunk_i)$ は Reduce ステップの引数として VRAM へ転送する (図 2 中の⑨). 同時に, N も VRAM へ転送する.

（2） MapReduce

MapReduce を行い, 最終的結果である BM25 を求める (図 2 中の⑩).

Map ステップは, 入力データの key である LocalWeight(i)-j ファイルから索引語, 文書 ID, $lw_{d,t}$ を抜き出し, 出力する. したがって, 出力 key/value ペアは, key を索引語, 文書 ID, value を $lw_{d,t}$ とする. すなわち, <(term, documentID), $lw_{d,t}$>である.

Shuffle ステップは key/value ペアを, key(term, documentID) の第一要素である索引語 (term) で辞書順に並べ替え, 索引語が等しい key については文書 ID で並べ替える.

Reduce ステップは同じ索引語を key に含むペアごとに並列に行われる. global-MR の入力時に, VRAM へ転送された<term, df_t>ペアのリストから, Reduce ステップの当該スレッドが担当する索引語の df_t 値を探索する. また, N も既に入力時に VRAM に転送されているので参照可能である. 以上で, gw_t の算出に必要な全ての値が揃ったため gw_t を算出し, $lw_{d,t}$ との積を求めることで BM25 による索引語重みとなる. こうして, Reduce ステップの出力は<(term, documentID), $w_{d,t}$>の key/value ペアとなる.

（3） 最終結果の出力

以上の global-MR を繰り返すことで, 全ての文書内の全ての索引語に対する BM25 の算出が完了する. この結果をテキストファイルに出力する (図 2 中の⑪). 出力の形式は (term, documentID, $w_{d,t}$) とする.

3.3　パイプライン処理

本研究では, 上述の local-MR と global-MR をそれぞれ三つのステージに分割し, パイプライン処理を行っている. local-MR は各チャンクごとに, global-MR は LocalWeight(i) ごとにパイプライン処理を行う. その概要を図 3 に示す. ただし, 全てのチャンクに対して local-MR の処理が完了するまで df_t と N の算出が完了しないため, global-MR を開始することができない. また, パイプラインの各ステージはそれぞれ独立したプロセスで行われる. ステージ間のデータの受け渡しは高速に行う必要があるため, メインメモリ内の共有領域にデータを置くこととする.

パイプライン処理の構成を述べる. はじめに, local-MR と global-MR において, パイプライン処理を行うステージをまとめる.

（1） Input

（2） MapReduce

（3） Output

Input ステージと Output ステージは local-MR, global-MR それぞれにおけるデータの入出力を行うステージである. MapReduce ステージは MapReduce 処理と, それに必要な入力や出力データを VRAM とメインメモリ間で転送するステージである. 本研究では, 上記の 3 ステージをパイプライン処理で行うことで高速化を行う.

3.4　コストモデル

以上の議論から, 提案手法の実行時間を見積もるためのコストモデルを立てる. 提案手法の実行時間はチャンクサイズ c を変数とした関数 $AllTime(c)$ とみなすことができる. また, local-MR, global-MR どちらにおいても, Input, Map, Reduce, Output ステップは計算量 $O(c)$ の処理であり, Mars では Shuffle ステップ内でバイトニックソートを使用しているため, Shuffle ステップは計算量 $O(c(\log c)^2)$ の処理であると考えられる. したがって, Input, Map, Reduce, Output ステップの各合計実行時間は下記式 (4), Shuffle ステップの合計実行時間は下記式 (5) で表すことができると仮定する.

- Input, Map, Reduce, Output ステップの各合計実行時間 $T_i(c), T_m(c), T_r(c), T_o(c)$

$$\begin{aligned} T_i(c) &= \frac{S}{c}(A'c + B') \\ &= SA' + \frac{SB'}{c} \\ &= A + \frac{B}{c} \end{aligned} \tag{4}$$

$T_m(c), T_r(c), T_o(c)$ はいずれも $T_i(c)$ と同様である.

- Shuffle ステップの実行時間 T_s

$$\begin{aligned} T_s(c) &= \frac{S}{c}(A'c(\log c)^2 + B'c) \\ &= SA'(\log c)^2 + SB' \\ &= A(\log c)^2 + B \end{aligned} \tag{5}$$

A', B' は式変形中にのみ出現する定数である. A, B は実験から求めることが可能な各ステップごとに異なる定数であり, S は文書集合のサイズである. したがって, $\frac{S}{c}$ は文書集合の分割数, すなわちチャンク数である. ここで, パイプライン処理における MapReduce ステージの合計実行時間 $T_{mr}(c)$ は下記式 (6) となる.

$$\begin{aligned} T_{mr}(c) &= T_m(c) + T_s(c) + T_r(c) \\ &= \left(A_m + \frac{B_m}{c}\right) + \left(A_s(\log c)^2 + B_s\right) + \left(A_r + \frac{B_r}{c}\right) \\ &= A_s(\log c)^2 + \frac{B_m + B_r}{c} + A_m + A_r \\ &= A(\log c)^2 + \frac{B}{c} + D \end{aligned} \tag{6}$$

$A_m, B_m, A_s, B_s, A_r, B_r$ はいずれも式変形中にのみ出現する定

数である．また，A, B, D は実験から求めることができる定数である．以上より，$AllTime(c)$ は下記式 (7) となる．

$$AllTime(c) = max\langle T_{i:l}(c), T_{mr:l}, T_{o:l}(c)\rangle + max\langle T_{i:g}(c), T_{mr:g}, T_{o:g}(c)\rangle \quad (7)$$

ここで，実行時間を表す $T(c)$ の添字に含まれる l, g はそれぞれ local-MR と global-MR を意味する．また，$max\langle a, b, c\rangle$ は a, b, c の内，最大のものを表す．

4. 評価実験

本節では，提案手法の有効性の検証のための評価実験について述べる．本研究の提案手法による索引語の重み付け計算の実行時間を計測した．また，森谷らの提案した手法 [10] との比較も行った．

なお，評価実験には下記の PC を用いた．

- CPU: Intel Core i7-4790 (3. 6GHz, 4 コア)
- メインメモリ: 16GB
- HDD: TOSHIBADT01ACA200(2TB)
- OS: CentOS 7
- GPU: NVIDIA GeForce GTX TITAN Z

なお，GPU の VRAM のサイズは 6GB である．

4.1 実験準備

本実験にはウェブからクローリングした英文文書から，索引語のみを抽出したテキストファイルの集合を文書集合とした．文書集合のサイズは，50MB, 250MB, 500MB, 1GB, 2GB の 5 種類である．

4.2 提案手法の実行時間

まず，提案手法においてチャンクサイズを変更させた時の実行時間の変化を図 4 に示す．チャンクサイズは，5MB, 10MB, 50MB の 3 種類で実験を行った．いずれの文書集合においても，チャンクサイズが 10MB の場合に最も良い結果が得られた．

図 5, 図 6 に各チャンクサイズにおける，パイプラインのステージごとの合計実行時間を示す．いずれも 500MB の文書集合を使用した．図 5 が local-MR, 図 6 が global-MR における結果である．つまり．最も実行時間の長いステージが local-MR, global-MR それぞれの実行時間となり，それらを足した値が全体の実行時間となる．例として，図 3 において，チャンク 1 の Output ステージ，チャンク 2 の MapReduce ステージ，チャンク 3 の Input ステージは同時に実行が開始されため，その中で最も実行時間が長いステージが全体の実行時間となる．local-MR ではチャンクサイズが小さいほどディスク I/O のオーバーヘッドが大きくなり，Input ステージと Output ステージの実行時間は長くなる．一方，MapReduce ステージではチャンクサイズが小さいほど Shuffle ステップで行うソートのオーバーヘッドは小さくなるが，VRAM とメインメモリ間でのデータ転送のオーバーヘッドは大きくなる．つまり，ソートにかかるコストとデータ転送にかかるコストはトレードオフの関係にあり，チャンクサイズが 10MB の場合が最良の結果となった．global-MR においても local-MR と同様の理由により，チャンクサイズが小さいほど Input ステージと Output ステージの実行時間は長

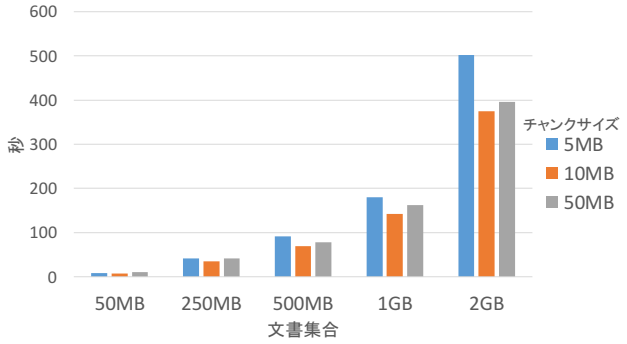

図 4　提案手法の実行時間

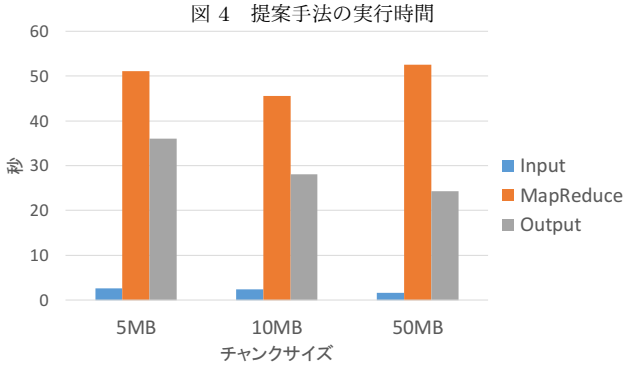

図 5　各チャンクサイズにおける local-MR のパイプラインのステージ合計実行時間 (文書集合サイズ 500MB)

くなる．一方，global-MR の MapReduce ステージの実行時間はチャンクサイズが大きいほど短くなる．これは，global-MR の Shuffle ステップでソートするデータ数が local-MR よりも少なく，チャンクサイズを大きくしても Shuffle ステップの実行時間の変化が小さいためである．理由を詳述すると，local-MR でソートするデータ数は，チャンクに含まれる文書の文書長 (= 文書に含まれる索引語数) の合計に等しいのに対し，global-MR でソートするデータ数は，索引語とそれを含む文書の重複のない組み合わせの数に等しいため，このような結果が得られる．

図 7, 図 8 に各チャンクサイズにおける，ステップごとの合計実行時間を示す．いずれも 500MB の文書集合を使用した．図 7 が local-MR, 図 8 が global-MR における結果である．local-MR においては，チャンクサイズを大きくするにつれて Shuffle ステップの実行時間が増えている．これは，Mars が Shuffle ステップで行うソートアルゴリズムに，計算量 $O(N(\log N)^2)$ のバイトニックソートを使用しているためである．また，Map ステップではチャンクサイズが大きい場合に実行時間が短くなっている．Map ステップでは複数の文書が GPU のコアに分散されて並列に処理されるため，チャンクサイズが小さい場合は並列度が低くなるためである．global-MR においても，Map ステップは local-MR と同様の理由により，チャンクサイズが大きい場合に実行時間は短くなる．global-MR における Shuffle ステップの処理時間は local-MR に比べて短くなっている．これは，前述の通り global-MR でソートするデータ数が local-MR よりも少ないためである．

図 9 に，提案手法においてパイプライン処理を行わず逐次処理で行った場合の実行時間を示す．各文書集合において最も処理時間が短いチャンクサイズを太字で示す．逐次処理で行う

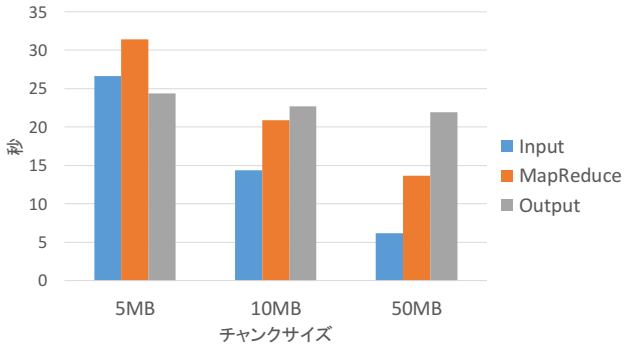

図 6　各チャンクサイズにおける global-MR のパイプラインのステージ実行時間
(文書集合サイズ 500MB)

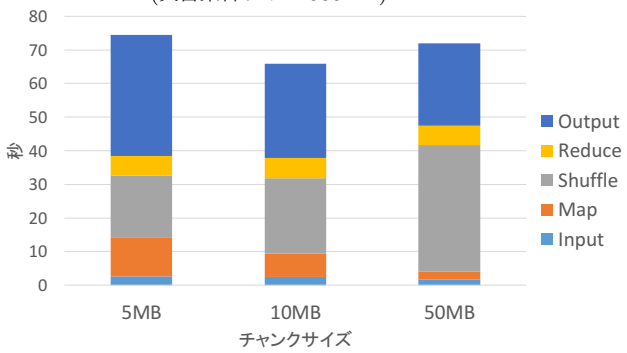

図 7　各チャンクサイズにおける local-MR の各ステップの合計実行時間
(文書集合 500MB)

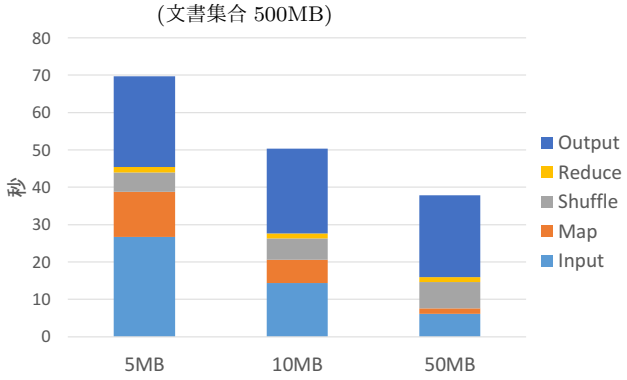

図 8　各チャンクサイズにおける global-MR の各ステップの合計実行時間
(文書集合 500MB)

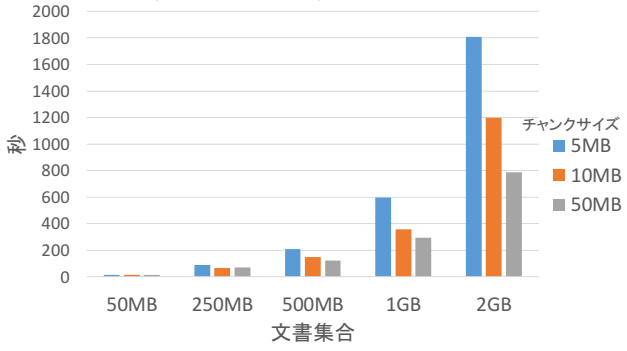

図 9　パイプライン処理を行わない場合の実行時間

場合, パイプライン処理で行う場合と異なり文書集合 50MB, 250MB ではチャンクサイズが 10MB, 文書集合 500MB 以上ではチャンクサイズが 50MB の場合に最も良い結果が得られた. 逐次処理を行う場合とパイプライン処理を行う場合の最も高速な結果を比較すると, 文書集合 1GB, 2GB ではパイプライン処理を行うことで約 2.1 倍の高速化を実現できた.

4.3　コストモデルの評価

本節では, 4 章で設計したコストモデルの評価を行う. 本節で扱う文書集合のサイズは全て 500MB とする.

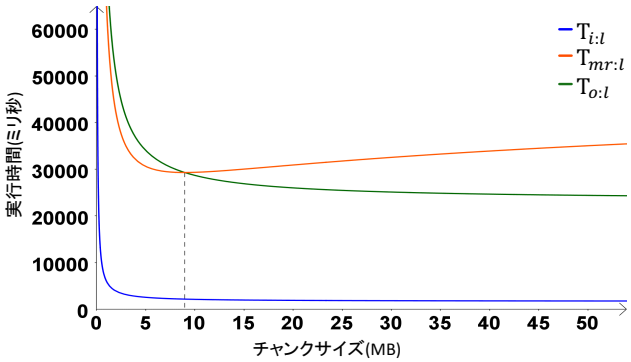

図 10　local-MR におけるコストモデルグラフ

チャンクサイズを 2MB から 50MB の間で 2MB ずつ変化させ, ステップごとの合計実行時間を計測する. その結果から, コストモデルの各式中の定数 A, B, D を求めることができる. その際, Shuffle ステップの合計実行時間 T_s において, チャンクサイズ c が十分小さい値で変化するため, $(\log c)^2 \fallingdotseq \log c$ とした. 図 10 にコストモデルによる各ステージの合計実行時間 $T_{i:l}, T_{mr:l}, T_{o:l}$ を示す. チャンクサイズが約 9MB の場合を境に $T_{mr:l}$ が $T_{i:l}$ を上回っていることがわかる.

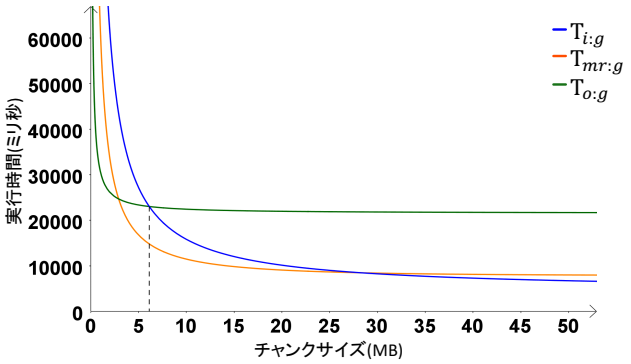

図 11　global-MR におけるコストモデルグラフ

同様に global-MR について, 図 11 にコストモデルによる各ステージの合計実行時間 $T_{i:g}, T_{mr:g}, T_{o:g}$ を示す. チャンクサイズが約 6MB の場合を境に $T_{o:g}$ が $T_{i:g}$ を上回っていることがわかる.

以上より, 全体の実行時間 $AllTime(c)$ を求めることができる.

$$AllTime(c) = \begin{cases} T_{o:l} + T_{i:g} & (2 \leqq c \leqq \text{約 } 6) \\ T_{o:l} + T_{o:g} & (\text{約 } 6 < c < \text{約 } 9) \\ T_{mr:l} + T_{o:g} & (\text{約 } 9 \leqq c) \end{cases} \quad (8)$$

図 12 は, $AllTime(c)$ をグラフで表した図である. グラフから, $AllTime(c)$ はチャンクサイズが約 9MB の場合に最小値を取ることがわかる. 評価実験ではチャンクサイズが 10MB の場合が最良であったので, コストモデルの結果と概ね一致する.

以上により, 提案したコストモデルの妥当性を示した.

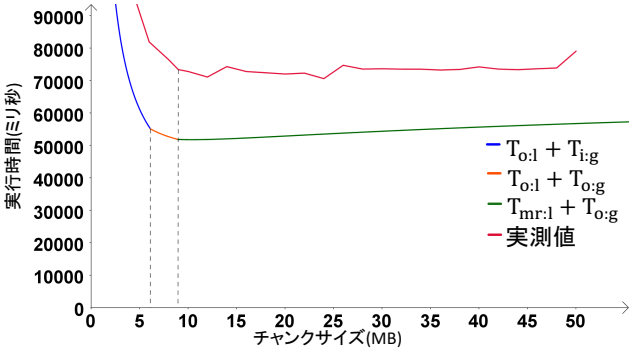

図 12　$AllTime(c)$ のグラフ

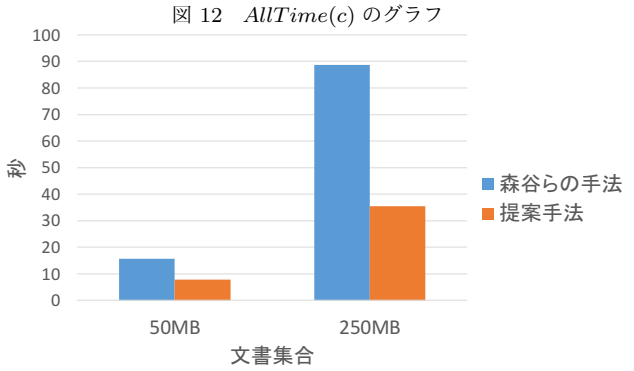

図 13　森谷らの手法と提案手法の実行時間

4.4　既存手法との比較

続いて, 森谷ら [10] の提案した, チャンクに分割せず一度に全データに対して処理を行う手法の実行時間と, 提案手法の実行時間を図 13 に示す. 提案手法の実行時間は, チャンクサイズを 10MB とした時のものである. なお, 森谷らの手法では 500MB の文書集合ではメモリ不足であったため, 50MB と 250MB における比較結果のみを掲載する. 50MB の文書集合に対しては 1.9 倍, 250MB の文書集合に対しては 2.5 倍の高速化を実現した.

5.　お わ り に

本研究は, GPU を用いて正確な情報検索のための索引語の重み付けを行う際に, データの分割を行うことで, 大規模データに対しても処理を行うことを可能とした. また, コストモデルを立て、検証を行うことで提案手法の実行時間の挙動を把握すると共に, 最適なチャンクサイズを計算により推定することが可能となった. その結果, 最適なチャンクサイズは, 一度に VRAM に転送して計算を行うことができる最大のサイズではなく, VRAM へデータを転送するコストやソートに要する時間の観点から, VRAM のサイズに対して小さな 10MB 程度をチャンクサイズとすることが良いということが判明した. 更に, 計算の各ステージに対してパイプライン処理を施すことで, 効率よく計算を行うことを実現した.

今後の課題として, 計算を行うマシンを増やすことでスケールアウトを行い, より大規模な文書集合を高速に処理することが考えられる. しかし, 文書集合を複数のマシンに分散することで, BM25 による重み算出に必要な文書集合全体における df_t と N を単純には算出することができない. したがって, マシン間で df_t と N を効率的に集約・共有する手法を考案する必要がある.

更に, 高速化のために文書中の文字列の扱い方を変えることが考えられる. 本研究の提案手法では, 文字列に対するソート処理や,　文字列をハッシュテーブルのキーとしている. GPU 上で索引語を文字列のまま扱うことはコストが大きくかかるため, 若月ら [12] は文字列である索引語を整数値と紐付ける辞書をメインメモリ上に構築し, GPU で処理を行う前に文字列を整数値に変換することで高速化を実現した. この手法を本研究にも適用することで, 更に効率的に大規模な文書の処理を行うことができると考える.

謝　　辞

本研究の一部は, JSPS 科研費 JP26280115, JP15H02701, JP16H02908, JP15K20990 の助成を受けたものである. ここに記して謝意を表す.

文　　献

[1] Christopher D. Manning, Prabhakar Raghavan, Hinrich Schütze, “Introduction to Information Retrieval”, Cambridge University Press. 2008.
[2] K. S. Jones, S. Walker, S.e. Robertson, “A probabilistic model of information retrieval: Development and comparative experiments”, Information Processing and Management, 36 (6): 779-808, 809-840, 2000
[3] J. Lin, C. Dyer, “Data-Intensive Text Processing with MapReduce”, Morgan & Claypool Publisher, 2010
[4] S. Tatikonda, F. Junqueira, B.B. Cambazoglu, V. Plachouras, “On efficient posting list intersection with multicore processors”, In Proc. of the 32nd ACM SIGIR, pp.738–739, 2009
[5] S. Tatikonda, B.B. Cambazoglu, F. Junqueira, “Posting list intersection on multicore architectures”, In Proc. of the 34th ACM SIGIR, pp.963–972, 2011
[6] J. Dean and S. Ghemawat, “Mapreduce: simplified data processing on large clusters”, Commun. ACM, 51(1):107-113, 2008
[7] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, C. Kozyrakis, “Evaluating mapreduce for multi-core and multiprocessor systems”, In Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture (HPCA ’07), pp. 13–24, Washington, DC, USA, 2007
[8] B. He, W. Fang, Q. Luo, N. K. Govindaraju, T. Wang, “Mars: A MapReduce Framework on Graphics Processors”, In Proc. of PACT 2008, pp. 260–269, 2008
[9] M. Garland et al. , “ Parallel Computing Experiences with CUDA”, IEEE Micro, Vol. 28, Iss. 4, pp. 13–27 , 2008
[10] 森谷 祐介, 欅 惇志, 宮崎 純「GPU を用いた MapReduce による高精度検索のための高速な重み計算」第 7 回データ工学と情報マネジメントに関するフォーラム, 2015
[11] Naga K. Govindaraju, Jim Gray, Ritesh Kumar, Dinesh Manocha “GPUTeraSort: High Performance Graphics Coprocessor Sorting for Large Database Management” ACM SIGMOD 2006, 325-336
[12] 若月駿尭, 欅 惇志, 宮崎 純「効率的なテキスト処理を目指した簡潔データ構造を用いるトライ木の GPU 上での実装」第 8 回データ工学と情報マネジメントに関するフォーラム, 2016