GPU 上の MapReduce による大規模データ処理の最適な分割粒度の動的推定

柳本 晟熙† 欅 惇志†,†† 宮崎 純†

† 東京工業大学情報理工学院情報工学系 〒 152–8550 東京都目黒区大岡山二丁目 12–1
†† 国立研究開発法人科学技術振興機構 〒 332–0012 埼玉県川口市本町 4–1–8
E-mail: †{yanagimoto,keyaki}@lsc.cs.titech.ac.jp, ††miyazaki@cs.titech.ac.jp

あらまし 本研究では, GPU 上で実装された並列分散処理フレームワーク MapReduce による大規模データの処理を行う際のデータ分割粒度について, コストモデルを用いた動的な最適化とその評価を行う. 我々の過去の研究から, 大規模データの処理を行う際の分割粒度には最適値が存在することが判明した. 本研究では, 最適な分割粒度を処理中に動的に推定する手法を提案し, 真の最適値でデータを分割した時との計算時間の比較を行う.

キーワード GPGPU,MapReduce, 最適化, 索引語重み付け

1. はじめに

情報社会の発展がめまぐるしい昨今, 大量のデータを高速に処理することが必要とされている. その方法の一つとして, 並列分散処理が挙げられる. 並列分散処理は複数のコンピュータ, プロセッサが互いに通信を行い並列に動作することで, 大規模なデータに対して高速に処理を行う手法である. 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 が提案した MapReduce [3] が存在する. MapReduce 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, それらのソート, グループ化を行う Shuffle ステップ, グループごとに Key/Value ペアを集約し, 結果を算出する Reduce ステップの三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 [4] も存在する.

MapReduce で高速に処理を行うためには, 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. 現在では一般的なマシンにも GPU が搭載されていることが珍しくないため, 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 を一台の GPU 上で行うためのフレームワークである Mars [5] 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 [6] 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扱えるように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ユーザでも容易に扱うことができる.

一般に, GPU を用いて大規模なデータに対してある処理を行う場合, 可能な限り大きなデータに分割して GPU のメモリに転送し, 処理を行うことが計算時間の観点から望ましいとされている [7]. しかしながら, 我々の過去の研究 [12] では, 分割粒度には計算時間を短くするある最適値が存在し, 実行するタスクや扱うデータの特性, 更にはマシンの性能によってその最適値が異なることが判明した. 分割粒度の最適値を知るためには計算時間を見積もるコストモデルを用いるが [12], そのためには予めタスクを数回実行しておく必要があった. そこで本研究では, 予めタスクを実行することなく動的に最適値を推定し, 最適値が未知のタスクについても効率的に計算を行う手法を提案する. 手法の評価には語の重み付け計算タスクと整数値ソートタスクの二種類のタスクを用いる.

2. 関連研究

2.1 MapReduce/Mars

MapReduce [3] は Google によって提案された並列分散処理のためのフレームワークである. 大きなデータセットをクラスタ内のノードに分散させ, 並列処理を行うことで高速に計算を行うことができる.

MapReduce は Map, Shuffle, Reduce ステップの三つのステップで構成される. 始めに入力データを分割し, 各ノードに分散する. Map ステップでは, 各ノードが受け取ったデータに対して Key/Value ペアを生成する. 続く Shuffle ステップは, ペアの Key を元にデータをソートし, 同じ Key を持つペアごとにノードに割り当てる. Reduce ステップでは, 各ノードで割り当てられたペアを集約することで最終結果を求める.

MapReduce を GPU 上で実装するためのフレームワークとして, Mars [5] が存在する. Mars を用いることで, GPU プログラミングの複雑さをユーザーがほとんど意識することなく, MapReduce を GPU 上で実装することが可能である. Mars において, MapReduce の各ノードは GPU のコアに相当する. データを各コアに割り当てる処理は CPU が行い, 実際の Map, Shuffle, Reduce ステップは GPU が行う. また, GPU で処理するデータは VRAM へ格納されるが, GPU は VRAM の動的

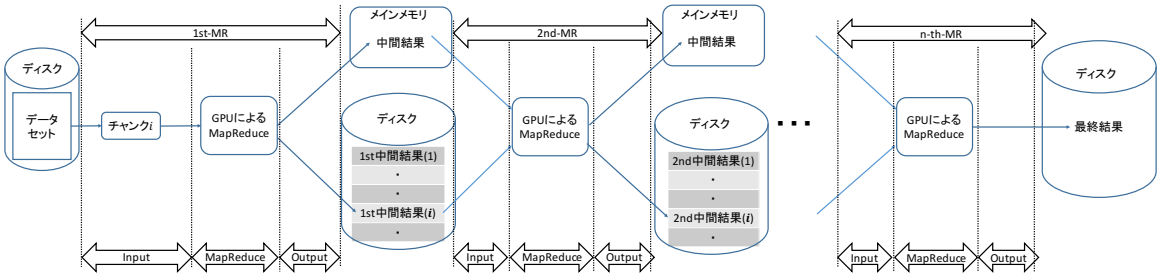

図 1　提案手法のワークフロー

確保を得意としていない．そこで, 各ステップを実行する前に擬似的計算を行い, 出力データのサイズを予め算出する工夫を施している．Mars は GPU における MapReduce の実装を容易にするだけでなく, 計算集約的演算において, マルチコア CPU 上で MapReduce を実装した Phoenix [4] よりも高速である場合が多い [5].

2.2　GPUTeraSort

Govindaraju ら [10] 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 CPU と GPU を協調させることで高速なソートを実現している．入力データは一度に VRAM に格納できないため, 複数のチャンクに分割して処理を行う．各チャンクに対して, 下記 (1)Reader から (5)Writer の五つのステージを繰り返し行う．各ステージは高速化のためにパイプライン処理で行われる．

（1） Reader
チャンクを一つメインメモリに読み込む．入力データはストライピングで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
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．

（3） Sorter
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．ソート結果は VRAM からメインメモリに転送される．

（4） Reorder
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
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チャンク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 をマージすることで最終結果とする．

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．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, 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, 全体の計算時間も変化する．

2.3　moderngpu

moderngpu [8] は Baxter により開発された CUDA 用ライブラリである．GPUTeraSort や Mars が採用しているバイトニックソートの計算量が $O(n(\log n)^2)$ であるのに対し, Baxter のマージソートは $O(n \log n)$ である．計算量ではバイトニックソートと比較してマージソートが高速であるが, マージソートは GPU が不得意とする条件分岐のコストがかかる．本研究では, Mars 標準のバイトニックソートと, Baxter のマージソートの二つのソートアルゴリズムの計算時間に対する影響について比較する．

3.　提 案 手 法

始めに, 予め指定した分割粒度でデータを分割し, タスクを実行する静的分割手法について述べる．続いて, 高速化のためのパイプライン処理 [10] とディスク I/O 分散, 更に分割粒度による計算時間への影響を把握するためのコストモデルについて述べる．最後に, コストモデルを用いて分割粒度をタスク実行中に動的に推定する動的推定方法について述べる．

3.1　静的分割手法

静的分割手法は, 入力データを予め指定した分割粒度で複数のチャンクと呼ばれる小さな集合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, ソートの計算時間は長くなる．一方, チャンクサイズを小さくすると, 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大きくなるが, ソートの計算時間は短くなる．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時間にはトレードオフの関係がある．

静的分割手法では, 複数の MapReduce タスクを繋げることで一連の処理を行う．この概要を図 1 に示す．図 1 中の 1st-MR, 2nd-MR, n-th-MR と記載された範囲それぞれが MapReduce タ

スクである．各 MapReduce タスクは, Input ステージ, MapReduce ステージ, Output ステージの三つのステージで構成される．MapReduce タスクの出力データはその大きさや特性に応じてディスクまたはメインメモリへ格納され, 次の MapReduce タスクへ渡される．各 MapReduce タスクは, 全てのチャンクに対して逐次的に 1 回ずつ行われ, 全てのチャンクで処理が完了した時点, すなわち中間結果が揃った時点で次の MapReduce タスクが開始される．最後の MapReduce タスクの出力が最終結果である．なお, 2nd-MR 以降の各 Input ステージで読み込むデータサイズは, 一つ前の MapReduce タスクの各チャンクの出力結果に依存する．すなわち, 指定したチャンクサイズでデータを分割するのは 1st-MR のみである．

3.2 パイプライン処理

本研究では高速化のために, MapReduce タスクを構成する 3 ステージに対し, パイプライン処理 [10] を適用する (図 2)．ただし, 3.1 節で述べた通り, 全てのチャンクに対してある MapReduce タスクが完了するまで中間結果が揃わないため, 次の MapReduce タスクを開始することはできない．また, Input ステージから MapReduce ステージへ渡すチャンクと MapReduce ステージから Output ステージへ渡す MapReduce の結果はメインメモリ内の共有領域にデータを置くこととする．

3.3 ディスク I/O の分散

本研究では, 前述の通りパイプライン処理を行っているため, Input ステージと Output ステージは同時に実行される．この時, 1 台のディスクにファイル入出力を集中的に行うことはディスク I/O オーバーヘッドの観点から好ましくない．一般に, 図 1 の構成はディスク 1 台で構成可能だが, 今回の実験では高速化の工夫として 2 台のディスクを用いてファイルの入出力を行う．ある MapReduce タスクで, Input ステージがあるディスクからデータの入力を行う時には Output ステージの出力は他方のディスクへ書き込む．これにより, 1 台のディスクに対しては常に読み込み/書き込みのどちらか一方のみが行われることになり, ディスク I/O オーバーヘッドを小さくすることができる．

3.4 コストモデル

タスクの計算時間を見積もるためのコストモデルを立てる．タスクの計算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, n-th-MR までのいずれの MapReduce タスクにおいても, Input, Output ステージ, Map, Reduce ステップ, メインメモリ・VRAM 間のデータ転送は計算量 $O(c)$ の処理である．Shuffle ステップは, マージソートを使用する場合は計算量 $O(c\log c)$, バイトニックソートを使用する場合は $O(c(\log c)^2)$ の処理である．したがって, 各ステージ・ステップの計算時間は下記の通り表すことができると仮定する．

Input, Output ステージの各合計計算時間 $T_{in}(c), T_{out}(c)$

$$\begin{aligned} T_{in}(c) &= \frac{S}{c}\cdot\frac{1}{T_h}(I_1 c + I_2) \\ &= I_1' + \frac{I_2'}{c} \end{aligned} \tag{1}$$

(S は文書集合のサイズ, T_h はディスクのスループット, I_1, I_2, I_1', I_2' は実験から推定可能な定数)

$T_{out}(c)$ は I_1, I_2 の値が異なるだけで同様の式となる．

Map, Reduce ステップの各合計計算時間 $T_{map}(c), T_{rdc}(c)$

$$\begin{aligned} T_{map}(c) &= \frac{S}{c}\cdot\frac{1}{C_u C_l}(M_1 c + M_2) \\ &= M_1' + \frac{M_2'}{c} \end{aligned} \tag{2}$$

(C_u は CUDA コア数, C_l は GPU のベースクロック数, M_1, M_2, M_1', M_2' はタスクごとに値が異なる実験から推定可能な定数)

$T_{rdc}(c)$ は M_1, M_2 の値が異なるだけで同様の式となる．

Shuffle ステップの合計計算時間 T_{sfl}(マージソートの場合)

$$\begin{aligned} T_{sfl}(c) &= \frac{S}{c}\cdot\frac{1}{C_u C_l}(S_{h1} c\log c + S_{h2}) \\ &= S_{h1}'\log c + \frac{S_{h2}'}{c} \end{aligned} \tag{3}$$

($S_{h1}, S_{h2}, S_{h1}', S_{h2}'$ は実験から推定可能な定数)

メインメモリ・VRAM 間のデータ転送時間 T_{trs}

$$\begin{aligned} T_{trs}(c) &= \frac{S}{c}\cdot\frac{1}{B_w}(T_{r1} c + T_{r2}) \\ &= T_{r1}' + \frac{T_{r2}'}{c} \end{aligned} \tag{4}$$

(B_w は PCI Express のバンド幅, $T_{r1}, T_{r2}, T_{r1}', T_{r2}'$ は実験から推定可能な定数)

MapReduce ステージの合計計算時間 $T_{mr}(c)$(マージソートの場合)

$$\begin{aligned} T_{mr}(c) &= T_{map}(c) + T_{sfl}(c) + T_{rdc}(c) + T_{trs} \\ &= \alpha\log c + \frac{\beta}{c} + \gamma \end{aligned} \tag{5}$$

(α, β, γ は実験から推定可能な定数)

全体の計算時間 $AllTime(c)$

$$AllTime(c) = \sum_{i=1}^{n} max\langle T_{in:i}(c), T_{mr:i}, T_{out:i}(c)\rangle \tag{6}$$

ここで, 計算時間を表す $T(c)$ の添字に含まれる : i は i 回目の MapReduce タスクを意味する．また, $max\langle a, b, c\rangle$ は a, b, c の内, 最大のものを表す．

3.5 動的推定手法

3.1 節では, チャンクサイズをタスクの実行前に指定する静的分割手法について述べた．本節では, 静的分割手法を改良し, 最適なチャンクサイズをタスク実行中に動的に推定する動的推定手法について述べる．

3.4 節で, タスクの計算時間を見積もるためのコストモデルについて述べた．計算時間を短くする最適なチャンクサイズの推定には, 各ステージのコストモデル (式 (1) や (5)) に含まれる定数 ($I_1', I_2', \alpha, \beta, \gamma$) を算出する必要がある．一番多く定数を含んでいるコストモデルは MapReduce ステージのものであり, その数は三つである．そこで, データの分布に偏りがないという条件のもとで, 最低三点 (三つの異なるチャンクサイズ) での計算時間を計測することで, 全てのコストモデルに含まれる定数を算出することができる．すなわち, 式 (7) をベクトル

(α, β, γ) について解くことで, (α, β, γ) を算出することができる. ここで, c_1, c_2, c_3 はそれぞれ異なるチャンクサイズであり, $T_{mr\_msrd}(c)$ は MapReduce ステージの合計計算時間の実測値である. 同様に, Input ステージのコストモデルに含まれる定数 I'_1, I'_2 は式 (8) で算出することができる. $T_{in\_msrd}(c)$ は Input ステージの合計計算時間の実測値である. (Output ステージのコストモデルに含まれる定数も Input ステージと同様に算出可)

$$\begin{pmatrix} \log c_1 & 1/c_1 & 1 \\ \log c_2 & 1/c_2 & 1 \\ \log c_3 & 1/c_3 & 1 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \\ \gamma \end{pmatrix} = \begin{pmatrix} T_{mr\_msrd}(c_1) \\ T_{mr\_msrd}(c_2) \\ T_{mr\_msrd}(c_3) \end{pmatrix} \tag{7}$$

$$\begin{pmatrix} 1 & 1/c_1 \\ 1 & 1/c_2 \end{pmatrix} \begin{pmatrix} I'_1 \\ I'_2 \end{pmatrix} = \begin{pmatrix} T_{in\_msrd}(c_1) \\ T_{in\_msrd}(c_2) \end{pmatrix} \tag{8}$$

以上の手法で定数を算出するためには計算時間の実測値 $T_{mr\_msrd}(c)$ や $T_{in\_msrd}(c)$ が必要であるが, これは全てのチャンクの合計計算時間であるため, 当然ながら全てのチャンクについて計算が完了するまで定数を算出することができない. ここで, 式 (7) を式 (9) のように変形する. $t_{mr\_msrd}(c)$ は MapReduce ステージのチャンク一つあたりの計算時間の実測値である. sf

$$\begin{aligned} & A \begin{pmatrix} c_1 \log c_1 & 1 & c_1 \\ c_2 \log c_2 & 1 & c_2 \\ c_3 \log c_3 & 1 & c_3 \end{pmatrix} \begin{pmatrix} \alpha' \\ \beta' \\ \gamma' \end{pmatrix} = A \begin{pmatrix} t_{mr\_msrd}(c_1) \\ t_{mr\_msrd}(c_2) \\ t_{mr\_msrd}(c_3) \end{pmatrix} \\ \Leftrightarrow & \begin{pmatrix} \log c_1 & 1/c_1 & 1 \\ \log c_2 & 1/c_2 & 1 \\ \log c_3 & 1/c_3 & 1 \end{pmatrix} \begin{pmatrix} \alpha' \\ \beta' \\ \gamma' \end{pmatrix} = \begin{pmatrix} t_{mr\_msrd}(c_1)/c_1 \\ t_{mr\_msrd}(c_2)/c_2 \\ t_{mr\_msrd}(c_3)/c_3 \end{pmatrix} \\ & A = \begin{pmatrix} S/c_1 & 0 & 0 \\ 0 & S/c_2 & 0 \\ 0 & 0 & S/c_3 \end{pmatrix} \end{aligned} \tag{9}$$

式 (9) を用いることで, 全てのチャンクの合計計算時間 $T_{mr\_msrd}(c)$ ではなく, チャンク一つあたりの計算時間 $t_{mr\_msrd}(c)$ を用いて定数 $(\alpha', \beta', \gamma')$ を算出することが可能である. 同様に, 式 (8) を式 (10) のように変形し定数 I''_1, I''_2 を算出する. t_{in_msrd} は Input ステージのチャンク一つあたりの計算時間の実測値である. (Output ステージの定数も Input ステージと同様に算出可)

$$\begin{aligned} & B \begin{pmatrix} c_1 & 1 \\ c_2 & 1 \end{pmatrix} \begin{pmatrix} I''_1 \\ I''_2 \end{pmatrix} = B \begin{pmatrix} t_{in\_msrd}(c_1) \\ t_{in\_msrd}(c_2) \end{pmatrix} \\ \Leftrightarrow & \begin{pmatrix} 1 & 1/c_1 \\ 1 & 1/c_2 \end{pmatrix} \begin{pmatrix} I''_1 \\ I''_2 \end{pmatrix} = \begin{pmatrix} t_{mr\_msrd}(c_1)/c_1 \\ t_{mr\_msrd}(c_2)/c_2 \end{pmatrix} \\ & B = \begin{pmatrix} S/c_1 & 0 \\ 0 & S/c_2 \end{pmatrix} \end{aligned} \tag{10}$$

図 2 に動的推定手法の概要を示す. 一つの MapReduce タスク内の最初の三つのチャンクサイズをそれぞれ c_1, c_2, c_3 として

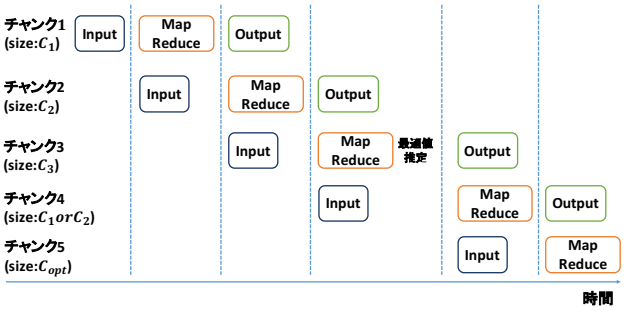

図 2　最適なチャンクサイズの推定

実行し (図 2 中チャンク 1, 2, 3), 得られた計算時間から式 (9) や (10) を用いて定数を算出する. 定数の算出によって定まったコストモデルから, 最適なチャンクサイズの推定値 c_{opt} を算出する (式 (11)).

$$\begin{aligned} c_{opt} &= \arg\min_c max\langle f_{in}(c), f_{mr}(c), f_{out}(c)\rangle \\ f_{in}(c) &= I''_1 + \frac{I''_2}{c} (f_{out}(c) \text{ は定数 } I''_1 \text{ と } I''_2 \text{ の値が異なる}) \\ f_{mr}(c) &= \alpha' \log c + \frac{\beta'}{c} + \gamma' \end{aligned} \tag{11}$$

図 2 のチャンク 4 については, Input ステージ開始時点で c_{opt} が算出されていない. そこで, チャンク 1 とチャンク 2 を比較し, Input ステージと MapReduce ステージのスループットが高いチャンクサイズをチャンク 4 のサイズとする. チャンク 5 以降は c_{opt} で実行する.

4. 評価を行うタスク

本研究では, 提案手法の評価を行うためのタスクとして BM25 による語の重み付け計算タスクと整数値ソートタスクを用いる. 本節では始めに重み付け手法の一つである BM25 について述べ, その後に重み付け計算タスクと整数値ソートタスクについて述べる.

4.1 BM25 による語の重み付け計算

4.1.1 BM25

BM25 [2] は語の重み付け手法の一つであり, 別の重み付け手法である TF-IDF [1] と比較して精度が高いことが知られている [2]. BM25 による重みは式 (12) で算出される. $w_{d,t}$ は文書 d における索引語 t の重みである. 式 (12) 中の $tf_{d,t}$ は文書 d における索引語 t の出現頻度, df_t は索引語 t を含む文書数, N は文書集合全体の文書数, dl_d は文書 d に含まれる索引語の数, $avdl$ は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, それぞれ $k_1 = 1.2, b = 0.75$ と設定する. $avdl$ 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 (12) の第一項目を以降局所的重みと呼び, 式 (13) に示し, 式 (12) の第二項目を以降大域的重みと呼び, 式 (14) に示す.

$$w_{d,t} = \frac{(k_1+1)tf_{d,t}}{k_1((1-b) + b\frac{dl_d}{avdl}) + tf_{d,t}} \cdot \log \frac{N - df_t + 0.5}{df_t + 0.5} \tag{12}$$

$$lw_{d,t} = \frac{(k_1+1)tf_{d,t}}{k_1((1-b)+b\frac{dl_d}{avdl})+tf_{d,t}} \quad (13)$$

$$gw_t = \log\frac{N-df_t+0.5}{df_t+0.5} \quad (14)$$

4.1.2 語の重み付け計算

森谷らは, GPU 上で実装された MapReduce を用いることで, 効率的に語の重み付け計算を行う手法を提案した [9]. しかし, GPU のメモリである VRAM のサイズに限りがあるため, 森谷らの手法 [9] では扱うことができる文書集合のサイズが限られている. そこで, 我々の過去の研究 [11] では, 静的分割手法を語の重み付け計算に適用することで, 森谷らの手法では扱うことができないサイズの文書集合を扱うことを可能とした.

静的分割手法で語の重み付け計算を行う手法 [11] を述べる. 重み付け計算に必要な MapReduce タスクの数は二つである. 1st-MR の目的は, BM25 による重み付け計算に必要な統計量を算出することである. BM25 の重み算出式 (式 (12)) に含まれる大域的重み (式 (14)) を算出するためには df_t が必要となる. df_t の算出には文書集合に含まれる全ての文書を参照しなければならないが, 1st-MR はチャンクごとに行われるためチャンクごとの df_t しか算出することができない. そこで, 1st-MR ではチャンクごとの df_t を算出し, それらを統合することで, 全てのチャンクに対して 1st-MR が完了した時に文書集合全体における df_t が算出される. また, 1st-MR では局所的重み (式 (13)) の算出も行う. 局所的重みは, 文書集合全体を参照する必要がなく, ある文書を参照するだけで算出可能である.

2nd-MR は, 1st-MR で算出した局所的重み, 文書集合全体における df_t を用いて最終結果である BM25 による重み付け計算を行う. なお, 2nd-MR は 1st-MR と同じ回数繰り返し行われる.

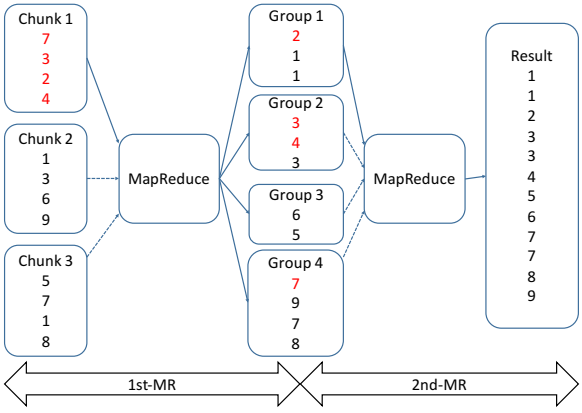

図 3　整数値ソートタスクの簡略図

4.2 整数値ソート

整数値ソートタスクも重み付け計算タスクと同様の構成 (図 1) で行う. ソートアルゴリズムはバケットソートを基に構成する. このアルゴリズムの簡略図を (図 3) に示す. 1st-MR では, チャンク内の整数値をいくつかのグループへ振り分けを行う. グループ数は後述する. 各グループには決められた範囲の整数値が含まれる. また, グループには順序があり (図 3 の Group1 から 4), グループ間で含まれる整数値には大小関係がある. 例として図 3 では, "Group1 に含まれる任意の整数値 <Group2 に含まれる任意の整数値 <Group3..." が成り立つ. 連続する複数のグループ, または単一のグループでバケットを構成する. バケットにはグループと同じ順序関係があり, 2nd-MR のチャンク一つはバケット一つに一致する. 全てのチャンクに対して 1st-MR が完了した (全ての整数値の振り分けが完了した) 後, 2nd-MR へ移行する. 2nd-MR では, 1st-MR で作成したバケットを一つ読み込み, バケット内の整数値のソートを行う. この時のソートは, MapReduce の Shuffle ステップに用いるソートアルゴリズムで行う. 全てのバケットに対してソートを行い, バケットを順序通り並べることで全整数値のソートが完了する.

グループとバケットの関係について述べる. 静的分割手法においては, データの分割数 (チャンク数) と同じ数のグループへ振り分け, 一つのグループで一つのバケットを構成する. しかし, この方法では 2nd-MR のチャンクサイズ (バケットサイズ) が固定されるため, 動的推定手法で動的にチャンクサイズを決定することができない. そこで, 動的推定手法ではグループ一つ当たりのサイズをチャンクサイズに対して十分小さくし, 2nd-MR では, チャンクサイズを超えない範囲で含めることができる最大数のグループで一つのバケットを構成する. こうすることで, グループ間の順序関係を保ったまま 2nd-MR で動的にチャンクサイズを決定することができる.

5. 評価実験

本節では, BM25 による重み付け計算タスクと整数値ソートタスクについて, 静的分割手法と動的推定手法のそれぞれの評価実験について述べる. また, 各タスクでバイトニックソートとマージソートの両方で実験を行った. 以降では BM25_Bitonic は重み付け計算タスクでバイトニックソートを用いた場合, BM25_Merge は重み付け計算タスクでマージソートを用いた場合, Sort_Bitonic は整数値ソートタスクでバイトニックソートを用いた場合, Sort_Merge は整数値ソートタスクでマージソートを用いた場合を表す. なお, 評価実験に用いたマシンの構成を表 1 に示す.

表 1　実験に使用したマシンの構成

CPU		Intel Core i7-4790 (3.6GHz, 4 コア)
RAM		16GB
HDD		TOSHIBA DT01ACA200
	容量	2TB
	最大データ転送速度	1815 Mbit/s
GPU		NVIDIA GeForce GTX TITAN Z (2 基搭載のうち 1 基のみ使用)
	CUDA コア数	2880
	ベースクロック	705MHz
	メモリ量	6GB GDDR5X
OS		CentOS7

5.1 データセット

BM25 による重み付け計算タスクに使用した文書集合は, 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とした. 文書集合のサイズは 4GB である.

整数値ソートタスクには 32bit 符号なし整数値を約 10 億個含んだファイルを使用した．すなわち，ファイルサイズは 4GB である．含まれる整数値は一様分布に従う乱数とした．

5.2 静的分割手法の実験結果

本節では重み付け計算タスクと整数値ソートタスクに静的分割手法を適用した結果について述べる．

重み付け計算タスクに静的分割手法を適用し，チャンクサイズ (横軸) を 2MB から 150MB まで変化させた時の計算時間 (縦軸) のグラフを図 4 の BM25_Bitonic_Measured と BM25_Merge_Measured に示す．実験結果から作成したコストモデルによる実行時間のグラフを図 4 の BM25_Bitonic_Model と BM25_Merge_Model に示す．BM25_Bitonic_Measured から，BM25_Bitonic はチャンクサイズ 2MB で計算時間は最大値を取り，22MB で最小値 216.9sec を取ることがわかる．その後は上昇に転じている．

一方，BM25_Merge は BM25_Merge_Measured から，チャンクサイズ 2MB で最大値を取ることは BM25_Bitonic と同様だが，チャンクサイズを大きくしても目立った上昇には転じず，概ね横ばいとなっていることがわかる．

整数値ソートタスクに静的分割手法を適用し，チャンクサイズ (横軸) を 2MB から 300MB まで変化させた時の計算時間 (縦軸) のグラフを図 5 の Sort_Bitonic_Measured と Sort_Merge_Measured に示す．実験結果から作成したコストモデルによる実行時間のグラフを図 5 の Sort_Bitonic_Model と Sort_Merge_Model に示す．Sort_Bitonic_Measured から，Sort_Bitonic はチャンクサイズ 2MB で計算時間は最大値を取り，254MB で最小値 164.0sec を取ることがわかる．その後は上昇に転じている．

一方，Sort_Merge は BM25_Merge と同様に，チャンクサイズ 2MB で最大値を取り，チャンクサイズを大きくしても目立った上昇には転じず，概ね横ばいとなっていることがわかる．

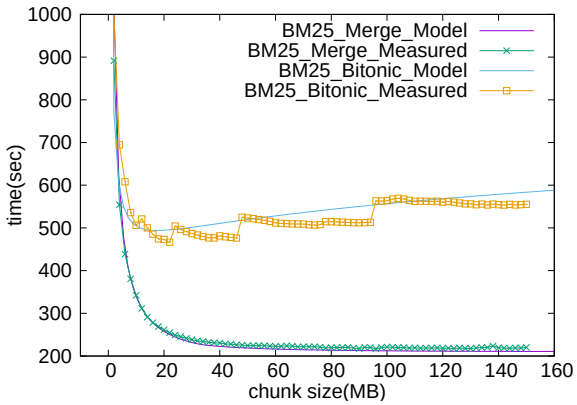

図 4 重み付け計算タスクにおける静的分割手法のコストモデルと実測値

5.3 動的推定手法の実験結果

本節では重み付け計算タスクと整数値ソートタスクに動的推定手法を適用した結果について述べる．始めに，最適値推定に必要な三つのチャンクサイズ c_1, c_2, c_3 の組み合わせを変えた時の各タスクの計算時間を表 2 に示す．

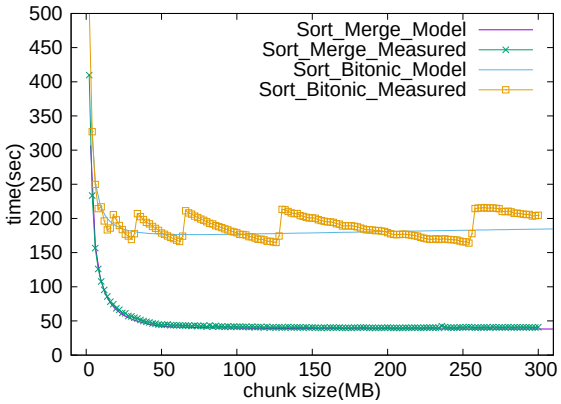

図 5 整数値ソートタスクにおける静的分割手法のコストモデルと実測値

$(c_1, c_2, c_3) = (30, 75, 120\text{MB})$ の Sort_Bitonic は推定失敗となった．これは，図 5 の Sort_Bitonic_Measured からわかるように計算時間が鋸歯状になっており，三点の組み合わせが悪くコストモデルの定数が妥当な値とならなかったからだと考える．全タスクの合計計算時間 (図 5 の Sum) を見ると，$(c_1, c_2, c_3) = (10, 50, 100\text{MB})$ が最良の結果となった．したがって，以降の結果は全て $(c_1, c_2, c_3) = (10, 50, 100\text{MB})$ のものとする．

(MB)	5,80,150	10,50,100	10,75,140	30,75,120
BM25_Bitonic	488.1	476.1	474.8	506.7
BM25_Merge	243.4	244.5	246.4	243.2
Sort_Bitonic	187.3	185.6	188.6	—
Sort_Merge	57.1	58.2	58.0	48.7
Sum	975.9	964.4	967.8	—

表 2 動的推定手法のタスクごとの計算時間と全タスク合計時間 (sec)

続いて，重み付け計算タスクに動的推定手法を適用した時の実験結果を表 3 と表 4 に示す．表 3 はバイトニックソートを用いた時，表 4 はマージソートを用いた時の結果である．各表の t_{de} は動的推定手法の計算時間 (sec)，c_{de_opt} は推定した最適なチャンクサイズ (MB)，t_{ss_min} は静的分割手法で最も短い計算時間 (sec)，c_{ss_min} はその時のチャンクサイズ (MB) である．ratio は t_{de} と t_{ss_min} の比 t_{de}/t_{ss_min} を表す．それぞれ 1st-MR, 2nd-MR, そしてタスク全体の結果である Total について記載する．なお，3.1 節で述べた通り，静的分割手法におけるチャンクサイズは 1st-MR に与えるものであり，2nd-MR の入力サイズとチャンクサイズは必ずしも一致しない．

BM25_Bitonic に動的推定手法を適用した時の全体の計算時間 t_{de} は，静的分割手法の最良の計算時間 t_{ss_min} の 1.02 倍であった．その時のチャンクサイズである c_{de_opt} と c_{ss_min} は大きく異なるものの，計算時間については概ね等しい結果が得られたと言える．BM25_Merge に動的推定手法を適用した時の全体の計算時間 t_{de} は，静的分割手法の最良の計算時間 t_{ss_min} の 1.13 倍であった．1st-MR においては，動的推定手法の計算時間は静的分割手法の最良計算時間の 1.03 倍であったが，2nd-MR においては 1.38 倍と動的推定手法が大きく劣る結果となった．

	$t_{de}(c_{de\_opt})$	$t_{ss\_min}(c_{ss\_min})$	ratio
1st-MR	386.4(33.8)	360.7(22)	1.07
2nd-MR	89.5(114.8)	88.7(74)	1.01
Total	476.1(—)	466.5(22)	1.02

表 3 重み付け計算タスクにおける動的推定手法と静的分割手法の計算時間の比較 (バイトニックソート)
計算時間: (sec), チャンクサイズ: (MB)

	$t_{de}(c_{de\_opt})$	$t_{ss\_min}(c_{ss\_min})$	ratio
1st-MR	160.3(125.2)	155.3(120)	1.03
2nd-MR	84.0(150)	60.8(96)	1.38
Total	244.5(—)	216.9(96)	1.13

表 4 重み付け計算タスクにおける動的推定手法と静的分割手法の計算時間の比較 (マージソート)
計算時間: (sec), チャンクサイズ: (MB)

	$t_{de}(c_{de\_opt})$	$t_{ss\_min}(c_{ss\_min})$	ratio
1st-MR	118.8(62.6)	72.6(256)	1.64
2nd-MR	66.8(63.4)	73.2(30)	0.91
Total	185.6(—)	164.0(254)	1.13

表 5 整数値ソートタスクにおける動的推定手法と静的分割手法の計算時間の比較 (バイトニックソート)
計算時間: (sec), チャンクサイズ: (MB)

	$t_{de}(c_{de\_opt})$	$t_{ss\_min}(c_{ss\_min})$	ratio
1st-MR	28.5(122.2)	22.4(70)	1.27
2nd-MR	29.6(197.2)	16.4(220)	1.80
Total	58.2(—)	39.8(170)	1.46

表 6 整数値ソートタスクにおける動的推定手法と静的分割手法の計算時間の比較 (マージソート)
計算時間: (sec), チャンクサイズ: (MB)

整数値ソートタスクに動的推定手法を適用した時の実験結果を表 5 と表 6 に示す. 結果の単位は表 3 と表 4 同様に, 計算時間は (sec), チャンクサイズは (MB) である. Sort_Bitonic に動的推定手法を適用した時の計算時間 t_{de} は, 静的分割手法の最良の計算時間 t_{ss_min} の 1.13 倍であった. 1st-MR においては, 動的推定手法の計算時間は静的分割手法の最良計算時間の 1.64 倍と動的推定手法が大きく劣る結果となったが, 2nd-MR では 0.91 倍と動的推定手法が静的分割手法を上回る結果となった. Sort_Merge に動的推定手法を適用した時の計算時間 t_{de} は, 静的分割手法の最良の計算時間 t_{ss_min} の 1.46 倍であった. これは全ての結果の中で動的推定手法の計算時間と静的分割手法の最良の計算時間の差が最も大きい結果である.

6. おわりに

本研究は, GPU 上で実装された MapReduce を用いて大規模なデータの処理を行う際に, 最適な分割粒度を動的に推定する手法を提案し, その評価を行った. 我々の過去の研究では, 計算時間に与えるソートアルゴリズムの影響と分割粒度の影響をコストモデルを用いて明らかにした. その際, 計算時間を短くする最適な分割粒度が存在することが判明したが, タスクの実行前に最適値を算出し, その最適なチャンクサイズでタスクを実行することは不可能であった. 一方, 本研究の提案手法は, タスクの実行中に最適値を動的に推定することを可能とした. 評価実験の結果, 動的推定手法による計算時間は静的分割手法の最良の計算時間と比較して, 重み付け計算タスクでは 1.02-1.13 倍, 整数値ソートタスクでは 1.13-1.46 倍の計算時間で実行することができると判明した.

今後の課題として, 重み付け計算や整数値ソート以外のタスクで動的推定手法の検証を行いたい. また, 今回のケースでは MapReduce フレームワークを使用したが, それ以外の並列分散処理モデルにおいても同様の検証を行いたいと考えている.

7. 謝　辞

本研究の一部は，JSPS 科研費 (JP15H02701，JP16H02908，JP15K20990，JP17K12684)，JST ACT-I 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. Dean and S. Ghemawat, “Mapreduce: simplified data processing on large clusters”, Commun. ACM, 51(1):107-113, 2008
[4] 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
[5] 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
[6] M. Garland et al. , “ Parallel Computing Experiences with CUDA”, IEEE Micro, Vol. 28, Iss. 4, pp. 13–27 , 2008
[7] Michael Boyer et al. , “Improving GPU Performance Prediction with Data Transfer Modeling”, IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum, pp. 1097–1106, 2013
[8] Sean Baxter: moderngpu 2.0, https://github.com/moderngpu/moderngpu/ (2017 年 6 月アクセス)
[9] 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
[10] Naga K. Govindaraju, Jim Gray, Ritesh Kumar, Dinesh Manocha “GPUTeraSort: High Performance Graphics Coprocessor Sorting for Large Database Management” ACM SIGMOD 2006, 325-336
[11] 柳本 晟熙, 欅 惇志, 宮崎 純「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」第 9 回データ工学と情報マネジメントに関するフォーラム, 2017
[12] 柳本晟熙, 欅惇志, 宮崎純「GPU 上の MapReduce による大規模データの処理におけるソートアルゴリズムの影響と評価」情報処理学会 第 165 回 データベースシステム研究会, 情報処理学会研究報告, 2017