情報処理学会論文誌　データベース　Vol.11　No.3　23–34 (Oct. 2018)

GPU上のMapReduceを利用した大規模データ処理の最適化

柳本 晟熙[1,a)]　欅 惇志[1,b)]　宮崎 純[1,c)]

受付日 2018年3月10日, 採録日 2018年5月9日

概要：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処理の最適化手法を提案する．一般に GPU のメモリサイズはメインメモリよりも小さく，大規模なデータを一度にすべて GPU のメモリに転送して処理を行うことは困難である．そこで本研究では，データを複数のチャンクに分割して GPU 上で繰り返し MapReduce 処理を行う際の最適な分割粒度をコストモデルを用いて明らかする．評価実験の結果，GPU のメモリに格納して計算ができ得る最大のサイズで分割を行うよりも，計算時間の観点からより小さなサイズで分割を行うことがよいと判明した．さらに，処理中に最適な分割粒度を動的に推定する動的推定手法を提案する．本研究で評価を行った BM25 による語の重み付け計算タスクでは動的推定手法の計算時間を，真の最適値でデータを分割したときの計算時間の最大 1.13 倍，整数値ソートタスクでは 1.46 倍に抑えることができた．

キーワード：GPGPU，MapReduce，最適化

Optimization of Large Data Processing Using MapReduce on a GPU

Masaki Yanagimoto[1,a)]　Atsushi Keyaki[1,b)]　Jun Miyazaki[1,c)]

Received: March 10, 2018, Accepted: May 9, 2018

Abstract: We present two optimization methods for processing a large amount of data with MapReduce using a graphics processing unit (GPU). It is difficult to transfer a large amount of data to the GPU memory (VRAM) and process them at a time, because the VRAM size is smaller than that of main memory in most cases. Thus, we investigate how to divide the data optimally with a cost model into multiple chunks so that each chunk fits into the VRAM of the GPU and can be processed efficiently with MapReduce on it. The experimental result showed that it exists an optimal chunk size which is smaller than the maximum one that GPU can store, and we can optimize it with our static optimization method. Moreover, we present a dynamic chunk size estimation method which finds an optimal chunk size online, and evaluated it. As a result, the dynamic chunk size estimation method could execute in 1.13x longer time for a term weighting task and in 1.46x for a sorting task compared to their ideal cases.

Keywords: GPGPU, MapReduce, optimization

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 1 つとして，並列分散処理があげられる．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技術として MPI [3] や OpenMP [4]，Google が提案した MapReduce [6] などが存在する．OpenMP は共有メモリ環境で利用できる API で，ディレクティブを記述することで並列処理を行うことができる．MPI は複数のプロセスが互いにメッ

[1] 東京工業大学情報理工学院情報工学系
Department of Computer Science, School of Computing, Tokyo Institute of Technology, Tokyo, Meguro 152–8550, Japan
[a)] yanagimoto@lsc.cs.titech.ac.jp
[b)] keyaki@lsc.cs.titech.ac.jp
[c)] miyazaki@cs.titech.ac.jp

セージをやりとりすることで協調動作するためのライブラリである．MPI は処理を詳細に記述できる反面，プログラミングが複雑となる．一方 MapReduce は処理を Map と Reduce の 2 つのメソッドのみで記述するため実装が容易であるが，自由に処理を記述できないという点で柔軟性には欠ける．

並列分散処理により大規模データ処理を高速に行うことができるが，さらに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．その解決方法として，MapReduce をマルチコア CPU 上で実装した Phoenix [9] が存在する．また，GPU を汎用計算に利用する GPGPU（General-purpose Computing on Graphics Processing Units）が存在する．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．現在では汎用コンピュータにも GPU が搭載されていることが一般的になりつつあるため，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と期待できる．また，GPGPU の一環として，1 台の GPU 上での MapReduce 実装である Mars [10] 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 [11] 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．

Mars などの研究により GPU プログラミングを行うことは容易になりつつあるが，大規模なデータを処理する際には課題が残る．GPU のメモリサイズはメインメモリよりも小さい場合が多く，大規模なデータを一度にすべて GPU のメモリに転送して処理を行うことは困難である．したがってデータを複数の小さなかたまり（チャンク）に分割して繰り返し処理を行う必要がある．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，VRAM に格納して計算を行える最大のサイズでデータを分割して GPU 上で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 [12]．しかし，我々の過去の研究では，データを複数のチャンクに分割して GPU 上で繰り返し MapReduce タスクを行うケースにおいては計算時間を短くするある最適な分割粒度が存在することが判明した [18]．チャンク 1 つあたりのサイズを大きくすることで，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．そこで，データの分割粒度であるチャンクサイズと計算時間の関係を見積もるコストモデルを立てる．このコストモデルにより計算時間を短くする最適な分割粒度でデータを分割して処理を行うことが可能となる．

本研究では，コストモデルを利用して最適な分割粒度を算出する静的分割手法 [19] と動的推定手法を提案する．静的分割手法はコストモデルによるキャリブレーションを行うことで判明する最適な分割粒度で入力データを分割する．そのため，同じ性質を持つデータに対して同じ処理を繰り返し行うような定型処理にはその最適な分割粒度で分割することが有効である．それに対して動的推定手法は，最適な分割粒度が未知のアドホックな処理に対して処理中に最適な分割粒度を動的に推定する．すなわち，静的分割手法では最適値を算出できないアドホックな処理に対しても最適値を推定して実行が可能である．本研究で評価を行った BM25 による語の重み付け計算タスクでは動的推定手法の計算時間を，真の最適値でデータを分割したときの計算時間の最大 1.13 倍，整数値ソートタスクでは 1.46 倍に抑えることができた．

2. 関連研究

並列分散処理を実現する技術として MPI [3] や OpenMP [4] などが存在する．しかし，MPI はプログラミングの複雑さ，OpenMP は適切な箇所にディレクティブを挿入する必要があるなどの課題がある．また，CUDA [11] による GPU プログラミングや OpenCL [5] による GPU を用いた並列分散処理も存在するが，いずれも同様の課題がある．これらに対して，MapReduce [6] は処理を Map と Reduce の 2 つのメソッドのみで実装ができ，さらに Mars [10] は CUDA による GPU プログラミングの複雑さを意識することなく GPU 上で MapReduce を実装することができる．

2.1 MapReduce/Mars

MapReduce [6] 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 3 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，各ノードが受けと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，同じ Key を持つペアをグルーピングし，グループごとに Reduce ステップのノードに割り当てる．グルーピングはハッシュやソートによって行う．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．Map，Shuffle，Reduce ステップの内，ソートを行う Shuffle ステップが計算時間で支配的となる．

GPU 上で MapReduce タスクを行う研究はいくつか存在する [7], [8]．また，MapReduce を GPU 上で実装するためのフレームワークとして，Mars [10] 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，VRAM の動的確保は高コストである．そこで，各ステップを実行する前に出力データのサイズをあらかじめ算出して VRAM の確保を行い，その後に実際の処理を実行する．さらに，同様の理由で Shuffle ステップのグルーピングはハッシュではなくソートにより行う．Mars が標準で Shuffle ステップに用いるソートアルゴリズムはバイトニックソートである．MapReduce は任意のキーでソートを行う必要があるため，ソートを行えるキーの種類に制限がある基数ソート [14] などは不向きである．

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 [9] よりも高速である場合が多い [10].

2.2 GPUTeraSort

Govindaraju ら [17] 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，(1) チャンクの読み込み，(2) ソート対象のレコードへのポインタに対応するキーの生成，(3) キーを GPU 上でソート，(4) ソート済みのキーを用いてレコードの並べ替え (5) ソート済みのデータの書き込み，以上 5 つのステージをパイプライン処理で並列に実行した後，各結果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 では高コストな条件分岐を行わない点においても相性がよい．

GPUTeraSort の性能は GPU の性能のみでなく，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 [13] 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．近年の GPU 上では，条件分岐をともなうマージソートをベースとしたソートアルゴリズムであってもバイトニックソートより高速である．また，$O(nk)$ の基数ソートは扱えるキーの種類が限られており汎用性に欠けるため，小澤ら [14] もマージソートベースのソートアルゴリズムを提案している．

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 2 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

本章ではまず，本研究の提案手法である静的分割手法と動的推定手法を構成する MapReduce タスクと，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．続いて本研究の提案手法である静的分割手法と動的推定手法について述べる．静的分割手法はコストモデルによるキャリブレーションを行うことで判明する最適な分割粒度で入力データを分割する．そのため，同じ性質を持つデータに対して同じ処理を繰り返し行うような定型処理にはその最適な分割粒度で分割することが有効であるが，一度しか実行しないアドホックな処理には静的分割手法を用いて最適な分割粒度を算出することはできない．それに対して動的推定手法は，最適な分割粒度が未知のアドホックな処理に対して処理中に最適な分割粒度を動的に推定する．すなわち，静的分割手法では最適値を算出できないアドホックな処理に対しても最適値を推定して実行が可能である．

3.1 MapReduce タスク

静的分割手法と動的推定手法ではデータを複数の小さなかたまり（チャンク）に分割し，GPU 上で繰り返し MapReduce 処理を行う．そこで大きな枠組みとして MapReduce タスクを定義し，これを**図 1** に示す．図 1 中の 1st-MR，2nd-MR，n-th-MR と記載された区間それぞれが MapReduce タスクである．このように MapReduce タスクを複数つなげることで様々な処理を実現することができる．各 MapReduce タスクは，Input ステージ，MapReduce ステージ，Output ステージの 3 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．MapReduce タスクを構成

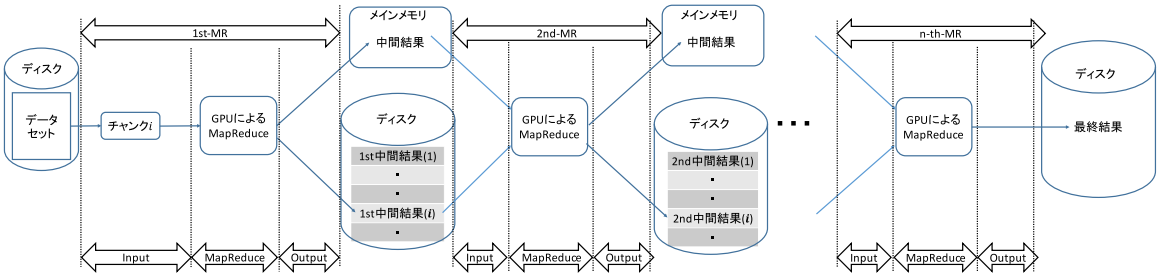

図 1　複数 MapReduce のワークフロー
Fig. 1　Workflow of multiple MapReduce jobs.

する 3 ステージはパイプライン処理 [17] で並列に行われる．ただし，すべ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，メインメモリ内に設けるパイプライン用のバッファを介して受け渡される．

ここで，一般的な MapReduce と本研究の提案手法の違いを述べる．一般的な MapReduce では，一度にすべてのデータに対して Shuffle ステップが行われ，続く Reduce ステップでは同一のキーを持つペアすべてが集約されて最終結果となる．一方，本研究の提案手法においては，各 MapReduce タスク内の Shuffle ステップはチャンクごとに行われるため，MapReduce タスクが完了した時点では同一のキーを持つペアすべてが集約されているとは限らない．

3.2　コストモデル

データの分割粒度であるチャンクサイズとタスクの計算時間の関係を見積もる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 $AllTime(c)$ で表す．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，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．Map，Reduce ステップで各インスタンスが処理するデータサイズは全体のデータサイズに対してきわめて小さく，また，典型的なテキスト処理やグラフ処理の Map，Reduce ステップは $O(c)$ となる [15]．そこで，Map と Reduce ステップは $O(c)$ と近似できると仮定する．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とおり表すことができる．

Input ステージの各合計計算時間 $T_{in}(c)$

$$\begin{aligned} T_{in}(c) &= \frac{S}{c}\cdot\frac{1}{T_h}(I_1 c + I_2) \\ &= I'_1 + \frac{I'_2}{c} \end{aligned} \tag{1}$$

（S は MapReduce タスクへの入力データサイズ，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）

Output ステージの各合計計算時間 $T_{out}(c)$

$$\begin{aligned} T_{out}(c) &= \frac{S}{c}\cdot\frac{1}{T_h}(O_1 c + O_2) \\ &= O'_1 + \frac{O'_2}{c} \end{aligned} \tag{2}$$

（O_1，O_2，O'_1，O'_2 は実験から推定可能な定数）

Map ステップの各合計計算時間 $T_{map}(c)$

$$\begin{aligned} T_{map}(c) &= \frac{S}{c}\cdot G_s(M_1 c + M_2) \\ &= M'_1 + \frac{M'_2}{c} \end{aligned} \tag{3}$$

（G_s は GPU の性能を表す定数，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）

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

$$\begin{aligned} T_{sfl}(c) &= \frac{S}{c}\cdot G_s(S_{h1} c\log c + S_{h2}) \\ &= S'_{h1}\log c + \frac{S'_{h2}}{c} \end{aligned} \tag{4}$$

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

Shuffle ステップの合計計算時間 T_{sfl}（バイトニックソートの場合）

$$T_{sfl}(c) = S'_{h1}(\log c)^2 + \frac{S'_{h2}}{c} \tag{5}$$

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

Reduce ステップの各合計計算時間 $T_{rdc}(c)$

$$T_{rdc}(c) = R'_1 + \frac{R'_2}{c} \tag{6}$$

（R_1，R_2，R'_1，R'_2 はタスクごとに値が異なる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}

$$\begin{aligned} T_{trs}(c) &= \frac{S}{c}\cdot\frac{1}{B_w}(T_{r1} c + T_{r2}) \\ &= T'_{r1} + \frac{T'_{r2}}{c} \end{aligned} \tag{7}$$

（B_w はメインメモリと VRAM 間のバンド幅，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）

$$T_{mr}(c) = T_{map}(c) + T_{sfl}(c) + T_{rdc}(c) + T_{trs} = \alpha \log c + \frac{\beta}{c} + \gamma \quad (8)$$

（α，β，γ は実験から推定可能な定数）

全体の計算時間 $AllTime(c)$

$$AllTime(c) = \sum_{i=1}^{n} max\big(T_{in:i}(c), T_{mr:i}, T_{out:i}(c)\big) \quad (9)$$

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，$max\,(a, b, c)$ は a，b，c の内，最大のものを表す．

3.3　静的分割手法

静的分割手法は，データに極端な偏りがないという前提でコストモデルを利用したキャリブレーションを行うことで判明する最適なチャンクサイズで入力データを分割する．そのため，同じ性質を持つデータに対して同じ処理を繰り返し行うような定型処理にはその最適なチャンクサイズで分割することが有効である．2nd-MR 以降のチャンクサイズは，1 つ前の MapReduce タスクの各チャンクの出力結果に依存する．

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 (8)）に含まれる定数（I'_1，I'_2，α，β，γ）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 3 つである．そこで，最低 3 点（3 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，すべ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 (10) 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 (11) 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同様に式 (12) で算出可することができる．$T_{out\_msrd}(c)$ は Output ステージの合計計算時間の実測値である．

$$\begin{pmatrix} \log c_1 & 1/c_1 & 1 \\ \log c_2 & 1/c_2 & 1 \\ \log c_3 & 1/c_3 & 1 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \\ \gamma \end{pmatrix} = \begin{pmatrix} T_{mr\_msrd}(c_1) \\ T_{mr\_msrd}(c_2) \\ T_{mr\_msrd}(c_3) \end{pmatrix} \quad (10)$$

$$\begin{pmatrix} 1 & 1/c_1 \\ 1 & 1/c_2 \end{pmatrix} \begin{pmatrix} I'_1 \\ I'_2 \end{pmatrix} = \begin{pmatrix} T_{in\_msrd}(c_1) \\ T_{in\_msrd}(c_2) \end{pmatrix} \quad (11)$$

$$\begin{pmatrix} 1 & 1/c_1 \\ 1 & 1/c_2 \end{pmatrix} \begin{pmatrix} O'_1 \\ O'_2 \end{pmatrix} = \begin{pmatrix} T_{out\_msrd}(c_1) \\ T_{out\_msrd}(c_2) \end{pmatrix} \quad (12)$$

各ステージのコストモデルに含まれる定数を算出することでコストモデルが定まり，計算時間を短くする最適なチャンクサイズを求めることができる．ここで，コストモデルに含まれる定数を算出するためには最低 3 点のチャンクサイズが必要であるが，4 点以上のチャンクサイズでの計算時間を計測することでより精度の高い推定が可能となる．一方で，キャリブレーションにかかる時間が増大するという欠点がある．

3.4　動的推定手法

3.3 節では，キャリブレーションにより最適なチャンクサイズを求めることができる定型処理に有効な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．動的推定手法は，データに極端な偏りがないという前提で静的分割手法では最適値を算出できないアドホックな処理に対しても最適値を推定して実行が可能である．

3.3 節で述べた静的分割手法のキャリブレーション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，これはすべてのチャンクの合計計算時間であるため，当然ながらすべ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．そこで，式 (10) を下記式 (13) のように変形する．$t_{mr\_msrd}(c)$ は MapReduce ステージのチャンク 1 つあたりの計算時間の実測値である．

繰り返し回数（チャンク数）を表す行列 A を

$$A = \begin{pmatrix} S/c_1 & 0 & 0 \\ 0 & S/c_2 & 0 \\ 0 & 0 & S/c_3 \end{pmatrix}$$

と定義する．A を用いて式 (10) から以下の式が導出される．

$$A \begin{pmatrix} c_1 \log c_1 & 1 & c_1 \\ c_2 \log c_2 & 1 & c_2 \\ c_3 \log c_3 & 1 & c_3 \end{pmatrix} \begin{pmatrix} \alpha' \\ \beta' \\ \gamma' \end{pmatrix} = A \begin{pmatrix} t_{mr\_msrd}(c_1) \\ t_{mr\_msrd}(c_2) \\ t_{mr\_msrd}(c_3) \end{pmatrix}$$

$$\Leftrightarrow \begin{pmatrix} \log c_1 & 1/c_1 & 1 \\ \log c_2 & 1/c_2 & 1 \\ \log c_3 & 1/c_3 & 1 \end{pmatrix} \begin{pmatrix} \alpha' \\ \beta' \\ \gamma' \end{pmatrix} = \begin{pmatrix} t_{mr\_msrd}(c_1)/c_1 \\ t_{mr\_msrd}(c_2)/c_2 \\ t_{mr\_msrd}(c_3)/c_3 \end{pmatrix} \quad (13)$$

式 (13) を用いることで，すべ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 1 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 (11) を式 (14) 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 1 つあたりの計算時間の実測値である．Output ステージも同様に式 (12) を式 (15) に変形し，定数 O''_1，O''_2 を算出する．t_{out_msrd} は Output ステージのチャンク 1 つあたり

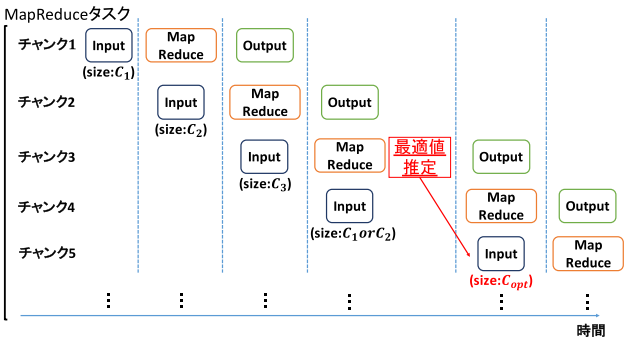

図 2　最適なチャンクサイズの推定
Fig. 2　Estimation of the optimal chunk size.

の計算時間の実測値である．

$$\begin{pmatrix} 1 & 1/c_1 \\ 1 & 1/c_2 \end{pmatrix} \begin{pmatrix} I_1'' \\ I_2'' \end{pmatrix} = \begin{pmatrix} t_{in\_msrd}(c_1)/c_1 \\ t_{in\_msrd}(c_2)/c_2 \end{pmatrix} \tag{14}$$

$$\begin{pmatrix} 1 & 1/c_1 \\ 1 & 1/c_2 \end{pmatrix} \begin{pmatrix} O_1'' \\ O_2'' \end{pmatrix} = \begin{pmatrix} t_{out\_msrd}(c_1)/c_1 \\ t_{out\_msrd}(c_2)/c_2 \end{pmatrix} \tag{15}$$

図 2 に動的推定手法の概要を示す．1 つの MapReduce タスク内の最初の 3 つのチャンクサイズをそれぞれ c_1，c_2，c_3 としてデータのサンプリングを行う．この 3 つのチャンクサイズで実行（図 2 中チャンク 1，2，3）したときの計算時間からコストモデルの定数を式 (13) や (14) を用いて算出し，最適なチャンクサイズの推定値 c_{opt} を算出する（式 (16)）．

$$\begin{aligned} c_{opt} &= \arg\min_c max\big(f_{in}(c), f_{mr}(c), f_{out}(c)\big) \\ f_{in}(c) &= I_1'' + \frac{I_2''}{c} \\ f_{mr}(c) &= \alpha' \log c + \frac{\beta'}{c} + \gamma' \\ f_{out}(c) &= O_1'' + \frac{O_2''}{c} \end{aligned} \tag{16}$$

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

提案手法は，大規模データに対する GPGPU において静的および動的に分割粒度を決定して効率的にパイプライン処理を行うフレームワークの提案であるため，その適用範囲は広いが，本論文では，その中でも主要かつ汎用的なタスクである文書集合に対する 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に対して評価を行う．

BM25 は確率モデルに基づく高精度な情報検索のための語の重み付けであり，大規模データを対象とした情報検索システムにおいて高速な重み計算は必要不可欠である．また，整数値のソートはきわめて多様なシステムにおいて汎用的に行われる処理である．

本章では始めに 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] は確率モデルに基づく語の重み付け手法であり，古典的な語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 (17) 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 (17) 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．

$$w_{d,t} = \frac{(k_1+1)tf_{d,t}}{k_1((1-b)+b\frac{dl_d}{avdl})+tf_{d,t}} \cdot \log\frac{N-df_t+0.5}{df_t+0.5} \tag{17}$$

$$lw_{d,t} = \frac{(k_1+1)tf_{d,t}}{k_1((1-b)+b\frac{dl_d}{avdl})+tf_{d,t}} \tag{18}$$

$$gw_t = \log\frac{N-df_t+0.5}{df_t+0.5} \tag{19}$$

ここで，式 (17) の第 1 項目を以降局所的重みと呼び，式 (18) に示し，式 (17) の第 2 項目を以降大域的重みと呼び，式 (19) に示す．

4.1.2　語の重み付け計算

森谷らは，GPU 上の MapReduce で可変長データを扱う語の重み付け計算が効率的に行えることを示した [16]．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 [16] 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 [18] 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．BM25 による語の重み付け計算を図 1 の構成で実現するためには，2 回の MapReduce タスクが必要である．以降では 2 回の MapReduce タスクである 1st-MR と 2nd-MR それぞれの役割について述べる．また，BM25 以外の重み付け計算も図 1 の構成で行うことができる．

4.1.3　1st-MR

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 による重みは，局所的重み（式 (18)）と大域的重み（式 (19)）の積により算出される．局所的重みはチャンクに含まれる各文書ごとに独立して計算を行うことができるため，1st-MR の各チャン

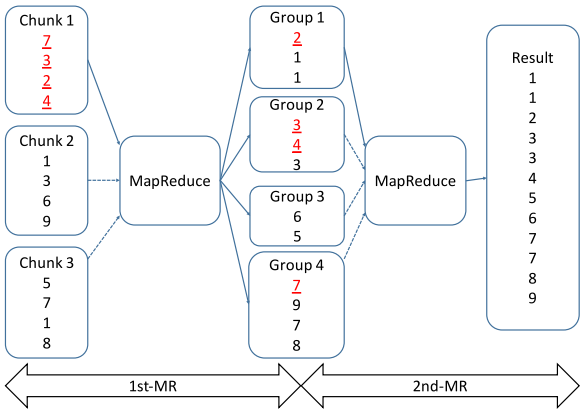

図 3　整数値ソートタスクの簡略図
Fig. 3　Overview of sorting task.

クの計算で算出する．一方大域的重みは，索引語 t を含む文書数 df_t，文書集合全体の文書数 N が必要となるため，文書やチャンクごとに独立して計算を行うことができない．そこで，チャンク i における索引語 t を含む文書数である $df_{t,i}$ とチャンク i に含まれる文書数 N_i をチャンクごとに求め，チャンクの計算が完了するたびにこれらを足し合わせていくことですべてのチャンクの計算が完了したとき（1st-MR の完了時）に df_t，N を算出することができる．

df_t，N，$lw_{d,t}$ を 1st-MR の結果として出力する．N については，単一の整数でありサイズはきわめて小さいため，メインメモリ内に保持する．また，df_t は索引語 t をキーとする key/value ペア$<t, df_t>$として保持する．このペアは索引語の種類の数だけ存在するが，本研究ではメインメモリに格納できる状況を想定する．メインメモリで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．$lw_{d,t}$ は索引語 t と文書 d を表す識別子 dID の組をキーとした key/value ペア$<(t, dID), lw_{d,t}>$として保持する．この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合せの数だけ存在するため，$<(t, dID), lw_{d,t}>$ペアはファイルとしてディスクに出力する．

4.1.4　2nd-MR

2nd-MR の目的は，1st-MR で求めた df_t，N，$lw_{d,t}$ から BM25 を算出することである．

Input ステージでは，$<(t, dID), lw_{d,t}>$ペアをファイルからメインメモリへ読み込む．MapReduce ステージでは，Input ステージで読み込んだ$<(t, dID), lw_{d,t}>$と$<t, df_{t,i}>$，N を VRAM へ転送する．これらの値から GPU 上で BM25 による重み $w_{d,t}$ を計算する．Output ステージでは MapReduce ステージの結果$<(t, dID), w_{d,t}>$をファイルへ出力する．

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．ここで，整数値の集合をグループ g，グループの集合を $G = \{g_1, g_2, \cdots, g_n\}$ とし，G は任意の 2 要素で順序関係が成り立つ全順序集合とする．すなわち，二項関係 $\ll$ を $A \ll B \Leftrightarrow a < b$（$\forall a \in A$，$\forall b \in B$）と定義したとき，$\forall g_x, g_y \in G$（$g_x \neq g_y$）に対して $g_x \ll g_y$ または $g_y \ll g_x$ が成り立つ．整数値ソートタスクの 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．このソートの手順を**図 3** に簡単に示す．

単一もしくは連続する複数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 1 つはバケット 1 つに一致する．すべてのチャンクに対して 1st-MR が完了した（すべ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 1 つ読み込み，バケット内の整数値のソートを行う．このときのソートは，MapReduce の Shuffle ステップに用いるソートアルゴリズムで行う．すべてのバケットに対してソートを行い，バケットを順序どお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，1 つのグループで 1 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 1 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 1 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

本章では，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 MapReduce の Shuffle ステップに用いるソートアルゴリズムをバイトニックソートとマージソートとして実験を行った．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，評価実験に用いた計算機の構成を**表 1** に示す．本実験では HDD を 2 台用いて MapReduce タスクの入出力を分散させた．

表 1　実験に使用したマシンの構成
Table 1　Machine specifications.

CPU		Intel Core i7-4790 （3.6 GHz，4 コア）
RAM		16 GB
HDD		TOSHIBA DT01ACA200（2 台）
	容量	2 TB
	最大データ転送速度	1,815 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	2,880
	ベースクロック	705 MHz
	メモリ量	6 GB GDDR5X
OS		CentOS7

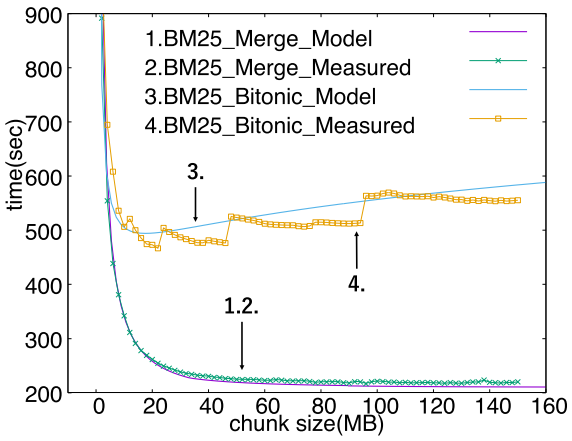

図 4　重み付け計算タスクにおける静的分割手法のコストモデルと実測値
Fig. 4　Cost model and measured time of Static Partitioning method for a term weighting task.

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 4 GB である．

整数値ソートタスクには 32 bit 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 4 GB 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．

5.2.1　BM25 による重み付け計算タスク

重み付け計算タスクに静的分割手法を適用し，キャリブレーションを行わずにチャンクサイズ（横軸）を 2 MB から 150 MB まで変化させたときの計算時間（縦軸）のグラフを**図 4** の BM25_Bitonic_Measured と BM25_Merge_Measured に示す．BM25_Bitonic は，チャンクサイズ 2 MB で計算時間は最大値をとり，22 MB で最小値 216.9 秒をとる．その後は上昇に転じている．一方，BM25_Merge は，チャンクサイズ 2 MB で最大値をとることは BM25_Bitonic と同様だが，チャンクサイズを大きくしても上昇には転じず，おおむね横ばいとなっている．

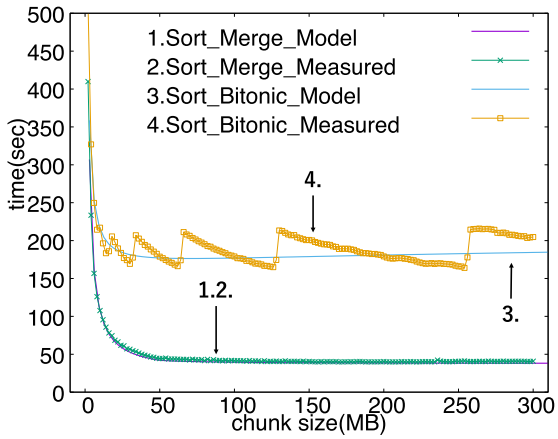

図 5　整数値ソートタスクにおける静的分割手法のコストモデルと実測値
Fig. 5　Cost model and measured time of Static Partitioning method for a sorting task.

5.2.2　整数値ソートタスク

整数値ソートタスクに静的分割手法を適用し，キャリブレーションを行わずにチャンクサイズ（横軸）を 2 MB から 300 MB まで変化させたときの計算時間（縦軸）のグラフを**図 5** の Sort_Bitonic_Measured と Sort_Merge_Measured に示す．Sort_Bitonic_Measured から，Sort_Bitonic はチャンクサイズ 2 MB で計算時間は最大値をとり，254 MB で最小値 164.0 秒をとることが分かる．その後は上昇に転じている．Sort_Merge は BM25_Merge と同様，チャンクサイズ 2 MB で最大値をとり，チャンクサイズを大きくしても上昇には転じず，おおむね横ばいとなっている．

5.2.3　コストモデルの評価

3.3 節でコストモデルを用いたキャリブレーションについて述べた．キャリブレーションに用いるチャンクサイズ 3 点は $(c_1, c_2, c_3) = (10, 50, 100)$ MB とした．3.3 節で述べたとおり 4 点以上のチャンクサイズを用いることで精度の高い推定が可能となるが，本実験では推定に必要な最小数のチャンクサイズで実験を行う．

キャリブレーションにより求めたコストモデルのグラフを図 4 の BM25_Bitonic_Model と BM25_Merge_Model，図 5 の Sort_Bitonic_Model と Sort_Merge_Model に示す．BM25_Bitonic においては，計算時間が最小となるチャンクサイズが実測値は 22 MB，キャリブレーションで求めた最適なチャンクサイズは約 17.5 MB となりおおむね一致している．キャリブレーションにより求めた最適なチャンクサイズでの計算時間を**表 3** の（B）に示す．計算時間の実測値が最小となった理想的なチャンクサイズ時の計算時間

表 3　理想的なチャンクサイズで分割した時と動的推定手法，静的分割手法の計算時間比較
Table 3　Comparison of the computation times of ideal chunk size cases, Dynamic Estimation method, and Static Partitioning method.

タスク	ソート（Shuffle）		動的推定手法（A）	静的分割手法（B）	理想チャンクサイズ時（C）	計算時間比（A/C）	計算時間比（B/C）
BM25	Bitonic	1st-MR	386.4（33.8）	371.6（17.5）	360.7（22）	1.07	1.03
		2nd-MR	89.5（114.8）	102.5（—）	88.7（74）	1.01	1.16
		Total	476.1（—）	474.2（—）	466.5（22）	1.02	1.02
	Merge	1st-MR	160.3（125.2）	158.5（203.9）	155.3（120）	1.03	1.02
		2nd-MR	84.0（150）	66.9（—）	60.8（96）	1.38	1.10
		Total	244.5（—）	225.7（—）	216.9（96）	1.13	1.04
Sort	Bitonic	1st-MR	118.8（62.6）	112.1（70）	72.6（256）	1.64	1.54
		2nd-MR	66.8（63.4）	94.2（—）	73.2（30）	0.91	1.29
		Total	185.6（—）	206.3（—）	164.0（254）	1.13	1.26
	Merge	1st-MR	28.5（122.2）	28.6（500）*	22.4（70）	1.27	1.28
		2nd-MR	29.6（197.2）	26.8（—）*	16.4（220）	1.80	1.63
		Total	58.2（—）	55.5（—）*	39.8（170）	1.46	1.39
			単位：sec（MB）	sec（MB）	sec（MB）		

* Sort_Merge において静的分割手法で求めた最適値 524.2 MB は VRAM のサイズが足りずに実行不可．したがって実行可能な範囲で 524.2 MB に最も近い 500 MB での結果を記載．

（表 3 の（C））と比較するとタスク全体で 1.02 倍に抑制した．

BM25_Merge においては，計算時間が最小となるチャンクサイズが実測値は 96 MB，キャリブレーションで求めた最適なチャンクサイズは 203.9 MB であった．キャリブレーションにより求めた最適なチャンクサイズでの計算時間（表 3 の（B））と計算時間の実測値が最小となった理想的なチャンクサイズ時の計算時間（表 3 の（C））と比較するとタスク全体で 1.04 倍に抑制した．

Sort_Bitonic においては，計算時間が最小となるチャンクサイズが実測値は 254 MB，キャリブレーションにより求めた最適なチャンクサイズは約 70.1 MB となり大きく差が開く結果となった．この結果は，バイトニックソートの仕組みに起因する．バイトニックソートはソートの対象となるデータの個数が 2^n でない場合，ダミーのデータを追加する必要がある．つまり，データの個数が 2^n の場合は最も効率が良くソートが行え，$2^n + 1$ で大きく効率が悪くなる．そして，$2^n + 1$ から 2^{n+1} までの間に次第に効率が良くなる．これは，図 4 の BM25_Bitonic_Measured や図 5 の Sort_Bitonic_Measured が倍々の間隔で鋸歯状になっていることからも分かる．コストモデルのグラフは鋸歯状の実測値を滑らかに近似するため，計算時間を最小にするチャンクサイズが実測値とコストモデルで異なる結果となった．キャリブレーションにより求めたチャンクサイズでの計算時間（表 3 の（B））と計算時間の実測値が最小となった理想的なチャンクサイズ時の計算時間（表 3 の（C））を比較するとタスク全体で 1.26 倍に抑制した．

Sort_Merge においては，計算時間が最小となるチャンクサイズが実測値は 170 MB，キャリブレーションで求めた最適なチャンクサイズは 524.2 MB であった．Sort_Merge では VRAM のサイズ不足によりチャンクサイズ 524.2 MB での実行が不可であった．そこで，実行可能な最大のチャンクサイズ 500 MB を表 3 の（B）に記載した．これと計算時間の実測値が最小となったときの理想的なチャンクサイズ時の計算時間（表 3 の（C））を比較するとタスク全体で 1.39 倍に抑制した．

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
Table 2　Computation time of each task by Dynamic Estimation and sum of all tasks.

タスク	ソート（Shuffle）	5,80,150	10,50,100	10,75,140	30,75,120
					（MB）
BM25	Bitonic	488.1	476.1	474.8	506.7
	Merge	243.4	244.5	246.4	243.2
Sort	Bitonic	187.3	185.6	188.6	—
	Merge	57.1	58.2	58.0	48.7
sum		975.9	964.4	967.8	—
					単位：sec

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 3 つのチャンクサイズ c_1，c_2，c_3 の組合せを変えたときの各タスクの計算時間を**表 2** に示す．$(c_1, c_2, c_3) = (30, 75, 120)$ 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，3 点の組合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である．全タスクの合計計算時間（表 2 の Sum）

表 4　最大チャンクサイズで実行したときの計算時間
Table 4　Computation time of partitioning with the maximum chunk size.

タスク	ソート（Shuffle）		静的分割手法 最大チャンク時（D）	計算時間比（B/D）**	各 MR タスク 最大チャンク時（E）	計算時間比（A/E）**
BM25	Bitonic	1st-MR	520.3（300）	0.71	517.7（300）	0.75
		2nd-MR	94.7（—）	1.08	108.7（800）	0.82
		Total	615.3（—）	0.77	626.6（—）	0.76
	Merge	1st-MR	178.6（450）	0.89	178.5（550）	0.90
		2nd-MR	65.9（—）	1.02	82.3（1,200）	1.02
		Total	244.9（—）	0.92	261.3（—）	0.94
Sort	Bitonic	1st-MR	90.6（300）	1.24	158.4（500）	0.75
		2nd-MR	115.6（—）	0.81	81.4（500）	0.82
		Total	206.3（—）	1.00	239.8（—）	0.77
	Merge	1st-MR	28.6（500）	1.00	31.0（300）	0.92
		2nd-MR	26.8（—）	1.00	31.3（300）	0.94
		Total	55.5（—）	1.00	62.5（—）	0.93
			単位：sec（MB）		sec（MB）	

** A と B の値は表 3 参照

を見ると，$(c_1, c_2, c_3) = (10, 50, 100)$ MB が最良の結果となった．したがって，以降の動的推定手法の評価はすべて $(c_1, c_2, c_3) = (10, 50, 100)$ MB とした．

続いて，動的推定手法の計算時間と計算時間の実測値が最小となった理想的なチャンクサイズ時の計算時間をそれぞれ表 3 の（A），（C）に示す．また，表 3 に（A）と（C）との比（A/C）を示す．なお，3.3 節で述べたとおり，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，（C）における 2nd-MR と Total の括弧内のチャンクサイズは 1st-MR に与えたチャンクサイズである．

BM25_Bitonic に動的推定手法を適用したときの全体の計算時間（A）は，理想的なチャンクサイズ時の計算時間（C）の 1.02 倍であった．そのときのチャンクサイズは動的推定手法と静的分割手法で大きく異なるものの，計算時間についてはおおむね等しい結果が得られたといえる．BM25_Merge に動的推定手法を適用したときの全体の計算時間（A）は，理想的なチャンクサイズ時の計算時間（C）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．

Sort_Bitonic に動的推定手法を適用したときの計算時間（A）は，理想的なチャンクサイズ時の計算時間（C）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ときの計算時間（A）は，理想的なチャンクサイズ時の計算時間（C）の 1.46 倍であった．これはすべ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計算時間の差が最も大きい結果である．

5.4　最大チャンクサイズでの実行

静的分割手法において，キャリブレーションを行わずにチャンクサイズを実行可能な限り大きく設定したときの計算時間を**表 4** の（D）に示す．また，キャリブレーションを行ったときの計算時間（表 3 の（B））と（D）との比（B/D）を表 4 に示す．重み付け計算タスクでは，キャリブレーションを行ったときの計算時間は最大チャンクサイズ時の計算時間よりも BM25_Bitonic では 0.77 倍，BM25_Merge では 0.92 倍高速であった．Sort_Bitonic では，キャリブレーションによって求めた最適値 70 MB での計算時間と最大チャンクサイズ 300 MB での計算時間では差が見られなかった．これは，図 5 の BM25_Bitonic_Measured から分かるようにチャンクサイズ 70 MB 付近でバイトニックソートの効率が悪くなっていることに起因する．Sort_Merge については表 3 の脚注に記載したように，（B）と（D）どちらもチャンクサイズ 500 MB での計算時間である．

さらに，1st-MR と 2nd-MR のそれぞれのチャンクサイズを計算可能な限り大きく設定したときの計算時間を表 4 の（E）に示す．動的推定手法の計算時間（A）との比（A/E）も同様に表 4 に示す．BM25_Merge の 2nd-MR 以外は動的推定手法の計算時間（A）が各 MapReduce タスクにおいて最大チャンクを与えたときの計算時間（E）よりも短いことが分かる．BM25_Merge の 2nd-MR では（A）が（E）を上回っているものの，タスク全体の計算時間では（A）の方が短い．これらの結果から，本研究のように GPU 上で繰り返し MapReduce タスクを行うようなケースでは，可能な限り粗い粒度でデータを分割するよりも，それよりも細かいある最適な粒度で分割を行うことが計算時間の観点

から望ましいといえる．

6. おわりに

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処理の最適化手法を提案し，評価を行った．静的分割手法はコストモデルによるキャリブレーションを行うことで判明する最適な分割粒度で入力データを分割する．したがって，同じ性質を持つデータに対して同じ処理を繰り返し行う定型処理に有効な手法である．一方，動的推定手法はタスクの実行中に動的に最適な分割粒度を推定し，データの分割を行う．こちらはあるデータに対してアドホックな処理でも最適値を推定して実行可能な手法である．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と比較し，重み付け計算タスクで 1.02–1.13 倍，整数値ソートタスクで 1.13–1.46 倍に抑制することができた．

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が，今回のように GPU 上で繰り返し MapReduce タスクを行う場合にはある最適な粒度で分割を行う方がより短い時間で計算を行うことができた．

一方，意図的に 1st-MR の出力サイズが入力チャンクごとに大きく偏るようにしたデータセットを用いた実験も行っており，極端に出力サイズが異なる場合には，動的推定手法の計算時間が静的分割手法よりも短くなるケースがあることを確認している．

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 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．さらに，複数のマシン，GPU を用いたスケールアウトを行いたい．この場合，中間データの保持方法やマシン間のデータの通信コストを新たに考慮する必要がある．

謝辞　本研究の一部は，JSPS 科研費（18H03242, 18H03342, 16H02908, 17K12684, 15H02701），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．

参考文献

[1] Manning, C.D., Raghavan, P. and Schütze, H.: Introduction to Information Retrieval, Cambridge University Press (2008).
[2] Jones, K.S., Walker, S. and Robertson, S.E.: A probabilistic model of information retrieval: Development and comparative experiments, *Information Processing and Management*, Vol.36, No.6, pp.779–808, 809–840 (2000).
[3] MPI, available from ⟨http://mpi-forum.org/⟩ (accessed 2018-03-14).
[4] OpenMP, available from ⟨http://www.openmp.org/⟩ (accessed 2018-03-14).
[5] OpenCL, available from ⟨https://www.khronos.org/opencl/⟩ (accessed 2017-03-14).
[6] Dean, J. and Ghemawat, S.: Mapreduce: simplified data processing on large clusters, *Commun. ACM*, Vol.51, No.1, pp.107–113 (2008).
[7] Stuart, J.A. and Owens, J.D.: 'Multi-GPU MapReduce on GPU Cluster, *Proc. 25th IEEE International Parallel and Distributed Processing Symposium, Ser. IPDPS '11* (May 2011).
[8] Shirahata, K. Sato, H. and Matsuoka, S.: Out-of-core GPU memory management for MapReduce-based large-scale graph processing, *2014 IEEE International Conference on Cluster Computing* (*CLUSTER*), pp.221–229 (2014).
[9] Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G. and Kozyrakis, C.: Evaluating mapreduce for multi-core and multiprocessor systems, *Proc. 2007 IEEE 13th International Symposium on High Performance Computer Architecture* (*HPCA '07*), pp.13–24 (2007).
[10] He, B., Fang, W., Luo, Q., Govindaraju, N.K. and Wang, T.: Mars: A MapReduce Framework on Graphics Processors, *Proc. PACT 2008*, pp.260–269 (2008).
[11] Garland, M. et al.: Parallel Computing Experiences with CUDA, *IEEE Micro*, Vol.28, No.4, pp.13–27 (2008).
[12] Boyer, M. 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).
[13] Sean Baxter: moderngpu 2.0, available from ⟨https://github.com/moderngpu/moderngpu/⟩ (accessed 2017-06).
[14] 小澤佑介，天笠俊之，北川博之：データ分割と協調的マージに基づく GPU 上の効率的ソートアルゴリズム，第 6 回データ工学と情報マネジメントに関するフォーラム，A2-1 (2014).
[15] Lin, J. and Dyer, C.: *Data-Intensive Text Processing with MapReduce*, Morgan and Claypool Publishers (2010).
[16] 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，G3-6 (2015).
[17] Govindaraju, N.K., Gray, J., Kumar, R. and Manocha, D.: GPUTeraSort: High Performance Graphics Coprocessor Sorting for Large Database Management, *ACM SIGMOD*, pp.325–336 (2006).
[18] 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，C8-3 (2017).
[19] 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).

柳本 晟熙　（学生会員）

2017 年東京工業大学工学部情報工学科卒業．東京工業大学情報理工学院情報工学系修士課程在学中．並列分散処理，GPU コンピューティングの研究に従事．

欅 惇志　（正会員）

東京工業大学情報理工学院助教．博士（工学）．2014 年奈良先端科学技術大学院大学情報科学研究科博士後期課程修了．2012～2014 年日本学術振興会特別研究員（DC2）．2013 年マイクロソフト・リサーチアジアリサーチインターン．2016～2017 年シンガポール国立大学客員研究員．ACM，電子情報通信学会，日本データベース学会，言語処理学会，人工知能学会各会員．

宮崎 純　（正会員）

東京工業大学情報理工学院教授．博士（情報科学）．1992 年東京工業大学工学部情報工学科卒業．1997 年北陸先端科学技術大学院大学情報科学研究科博士後期課程修了．同大助手，奈良先端科学技術大学院大学情報科学研究科准教授を経て現職．2000～2001 年テキサス大学アーリントン校客員研究員．2003～2007 年科学技術振興機構さきがけ研究員．データベース，情報検索ならびに高性能計算の研究に従事．電子情報通信学会，日本データベース学会，ヒューマンインタフェース学会，ACM，IEEE 各員．本会シニア会員．

（担当編集委員　井上 潮）