Vol.2017-DBS-165 No.28
Vol.2017-IFAT-128 No.28
2017/9/20

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

柳本 晟熙[1,a)]　欅 惇志[1,b)]　宮崎 純[1,c)]

概要：本研究では, GPU 上で実装された並列分散処理のフレームワークである MapReduce を用いて, 大規模なデータに対して処理を行う場合について, 使用するソートアルゴリズムの違いによる計算時間のコストモデルとその評価を行う. 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. 本研究では, バイトニックソートとマージソートの違いにより, データの分割粒度が処理時間にどのような影響を与えるのか, 整数値のソートとウェブ上の文書から抽出された索引語に対する重み付け計算の二種類のタスクを利用して明らかにする.

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

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 が開発した MapRedece[3] 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]. 本研究では, Mars を用いて大規模なデータを分割しながらある処理を行う場合, 使用するソートアルゴリズムの違いによる計算時間への影響をコストモデルを立て, 評価を行う. その際, データの最適な分割粒度を明らかにする. 本研究で評価を行うタスクは, ウェブ上の文書から抽出された索引語に対する重み付け計算と整数値ソートの二種類とする. 使用するソートアルゴリズムはバイトニックソートとマージソートの二種類とする.

2. 関連研究

本節では始めに, 本研究で用いる並列分散処理フレーム

[1] 東京工業大学
Meguro, Tokyo 152–8550, Japan
[a)] yanagimoto@lsc.cs.titech.ac.jp
[b)] keyaki@lsc.cs.titech.ac.jp
[c)] miyazaki@cs.titech.ac.jp

ワークである MapReduce と, それを GPU 上で実装するためのフレームワークである Mars について述べる. 続いて, 語に対する重み付け手法の一つである BM25 と, Mars を用いて語に重み付けを行う手法について述べる. また, GPU を用いたソート手法である GPUTeraSort と, CUDA のライブラリである moderngpu について述べる.

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 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 (1) 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 (1) 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 (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.

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

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

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

2.3 Mars を用いた語の重み付け手法

森谷ら [9] は, Mars を用いて効率的に BM25 による重み付け計算を行う手法を提案した. 文書データを入力とし, <索引語, 文書 ID, BM25 による重み> を出力とする. Map ステップでは, 文書から索引語を抽出し, 文書 ID を付与する. Shuffle ステップでは, Map ステップで作成した Key/Value ペアを索引語, 文書 ID についてソート・グループ化を行う. Reduce ステップではグループごとに BM25 による重みの計算を行う.

森谷らによる Mars を用いた重み付け手法は, マルチコア CPU 上で重み付け計算を行った場合と比較し, 約 4-11 倍の高速化を実現した [9].

2.4 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.

2.5 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 $O(n \log n)$ であり, バイトニックソートと比較して高速である. 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. 評価を行うタスク

本研究で評価を行う二種類のタスクについて述べる. 一つ目は森谷らの手法を改良した BM25 による重み付け計算である. 二つ目は整数値ソートである. 始めにこれらのタスクについて述べ, 続いて高速化のためのパイプライン処理とディスク 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.

3.1 BM25 による重み付け計算

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, 森谷らの手法を改良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手法を提案した. これを提案手法と呼ぶ. 更に, 提案手法は後述のパイプライン処理などを施すことで, 森谷らの手法の高速化も実現した [11].

提案手法では, 文書集合を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, 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 タスクを提案し, それぞれ 1st-MR と 2nd-MR とする. ここで, 提案手法の概要を図 1 に示す. 1st-MR と 2nd-MR はそれぞれ図 1 内で示されている範囲を指し, いずれも Input, MapReduce, Output の三つのステージから構成される. また, 図 1 に示されるような 1st-MR で中間結果を求め, 2nd-MR で最終結果を求める構成は, 重み付け計算だけではなく後述の整数値ソートにも適用可能である.

図 1 の構成で語の重み付け計算を行う手法 [11] を述べる. 1st-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. 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 1) に含まれる大域的重み (式 3) 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 2) 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.

3.2 整数値ソート

重み付け計算タスクと同様の構成 (図 1) で整数値ソートを行う. 整数値ソートの簡略図を (図 5) に示す. 1st-MR ではチャンク内の整数値をチャンクの数と同じ数のファイルへ振り分ける. 全てのチャンクに対して 1st-MR が完了した (振り分けが完了した) 後, 2nd-MR へ移行する. 2nd-MR では, 1st-MR で作成したファイルを順番に読み込み, 含まれる整数値のソート行う. 各ファイルの 2nd-MR の結果をマージすることで, ソートが完了する.

3.3 パイプライン処理

本研究では高速化のために, 前述の 1st-MR と 2nd-MR を構成する 3 ステージに対し, GPUTeraSort[10] 同様パイプライン処理を適用する. ただし, 全てのチャンクに対して 1st-MR の処理が完了するまで中間結果が揃わないため, 2nd-MR を開始することはできない. また, 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.

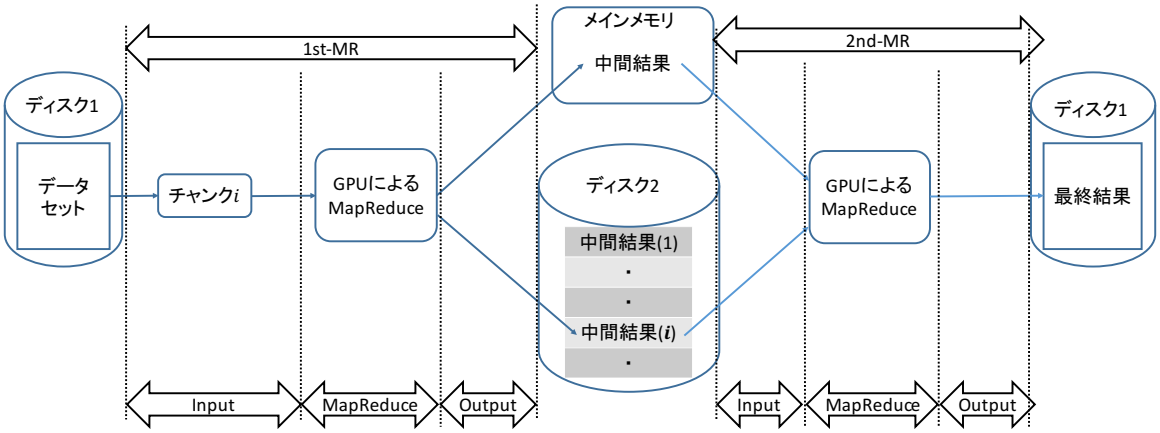

図 1　提案手法のワークフロー

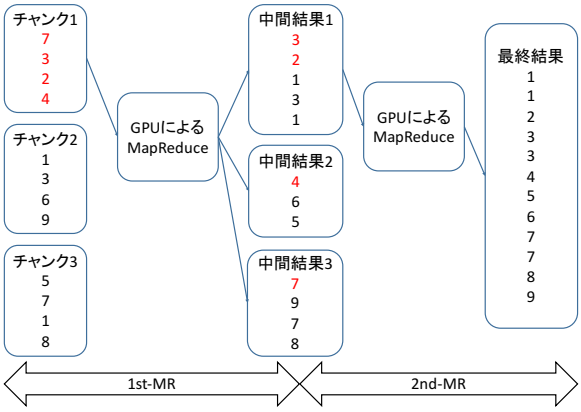

図 2　整数値ソートタスクの概要

3.4　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. そこで, 本研究では 2 台のディスクを用いてファイルの入出力を行う (図 1 のディスク 1, 2). 1st-MR の Input ステージ, すなわち文書の入力はディスク 1 から行い, 同時に行われる Output ステージ, すなわち中間ファイルの出力はディスク 2 に対して行う. 同様に 2nd-MR の Input ステージはディスク 2 から, Output ステージはディスク 1 に対して行う. こうすることにより, 1st-MR, 2nd-MR 共に 1 台のディスクに対しては読み込み/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.

3.5　コストモデル

以上の議論から, 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, 1st-MR, 2nd-MR どちらにおいても, 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 $T_{in}(c), T_{out}(c)$

$$\begin{aligned} T_{in}(c) &= \frac{S}{c} \cdot \frac{1}{T_h}(I_1 c + I_2) \\ &= I'_1 + \frac{I'_2}{c} \end{aligned} \tag{4}$$

(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)

ここで, I_2 は I_1 と比較して十分小さい値であるとみなせるため, $I_2 \fallingdotseq 0$ と近似する. したがって,

$$T_{in}(c) = I'_1 \tag{5}$$

$T_{out}(c)$ は I_1, I_2 の値が異なる.

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

$$\begin{aligned} T_{map}(c) &= \frac{S}{c} \cdot \frac{1}{C_u C_l}(M_1 c + M_2) \\ &= M'_1 + \frac{M'_2}{c} \end{aligned} \tag{6}$$

(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.

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

$$\begin{aligned} T_{sfl}(c) &= \frac{S}{c} \cdot \frac{1}{C_u C_l}(S_{h1} c \log c + S_{h2}) \\ &= S'_{h1} \log c + \frac{S'_{h2}}{c} \end{aligned} \tag{7}$$

($S_{h1}, S_{h2}, S'_{h1}, S'_{h2}$ 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}

$$\begin{aligned} T_{trs}(c) &= \frac{S}{c} \cdot \frac{1}{B_w}(T_{r1} c + T_{r2}) \\ &= T'_{r1} + \frac{T'_{r2}}{c} \end{aligned} \tag{8}$$

(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)

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

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

$$T_{mr}(c) = T_{map}(c) + T_{sfl}(c) + T_{rdc}(c) + T_{trs} = \alpha \log c + \frac{\beta}{c} + \gamma \quad (9)$$

(α, β, γ は実験から推定可能な定数)

全体の計算時間 $AllTime(c)$

$$AllTime(c) = max\langle T_{in:1}(c), T_{mr:1}, T_{out:1}(c)\rangle + max\langle T_{in:2}(c), T_{mr:2}, T_{out:2}(c)\rangle \quad (10)$$

ここで, 計算時間を表す $T(c)$ の添字に含まれる $1, 2$ はそれぞれ 1st-MR と 2nd-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

本説では, BM25 による重み付け計算と整数値ソートの二種類のタスクについて, それぞれバイトニックソートとマージソートを用いた評価実験について述べる. 以降は重み付け計算タスクにおいてバイトニックソートとマージソートを用いた実験をそれぞれ BM25_Bitonic, BM25_Merge とし, 整数値ソートタスクを Sort_Bitonic, Sort_Merge とする. なお, 評価実験に用いたマシンの構成を表 (1) に示す.

4.1 データセット

BM25 による重み付け計算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.

4.2 BM25 による重み付け計算タスク

重み付け計算タスクの計算時間のグラフを図 4 の BM25_Bitonic_Measured と BM25_Merge_Measured に示す. 横軸は文書集合の分割粒度であるチャンクサイズを示し, 2MB から 150MB まで変化させた. BM25_Bitonic は, チャンクサイズ 2MB で計算時間は最大値を取り, 22MB で最小値を取る. その後は上昇に転じている. 一方, BM25_Merge は, チャンクサイズ 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上昇には転じず, 概ね横ばいとなっている.

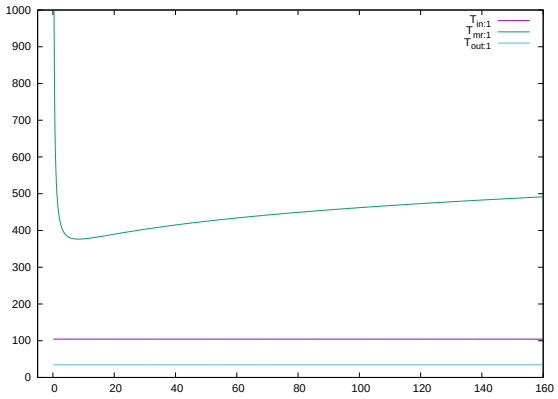

図 3　BM25_Bitonic における 1st-MR のコストモデル

4.3 整数値ソートタスク

整数値ソートタスクの計算時間のグラフを図 5 の Sort_Bitonic_Measured と Sort_Merge_Measured に示す. チャンクサイズは 2MB から 300MB まで変化させた. Sort_Bitonic は, チャンクサイズ 2MB で計算時間は最大値を取り, 62MB で最小値を取る. Sort_Merge は BM25_Merge 同様にチャンクサイズを大きくしても上昇には転じず, 概ね横ばいとなる.

4.4 コストモデルの評価

3.5 節で設計したコストモデルの評価を行う.

ここでは例として, BM25_Bitonic タスクの 1st-MR について評価を行う. 式 (9) において, (α, β, γ) の 3 値を定めれば, MapReduce ステージのコストモデルを決定することができる. そこで, 任意のチャンクサイズを三つ選び, その計算時間の実測値から解析的に (α, β, γ) を決定することができる. ここでは, チャンクサイズ 10MB, 50MB, 100MB を用いる. その結果, コストモデル $T_{mr:l}$ では, $(\alpha = 6.305 \times 10, \beta = 6.691 \times 10^2, \gamma = 1.651 \times 10^2)$ と算出できる. Input ステージのコストモデルである式 (5) についてはチャンクサイズ 100MB の実測値を I'_1 の値とし, $T_{in:l} = 1.043 \times 10^2$ となる. Output ステージのコストモデルも同様に決定でき, $T_{out:l} = 3.432 \times 10$ となる. 以上で求めたコストモデルのグラフを図 3 に示す. いずれのチャンクサイズにおいても MapReduce ステージの計算時間が三つのステージの中で最大である.

同様に, 2nd-MR のコストモデルにおける定数を求め, 1st-MR のコストモデルと組み合わせることで全体の計算時間のコストモデルを求めることができる. そのグラフを図 4 の BM25_Bitonic_Model に示す. 計算時間が最小となるチャンクサイズは, 実測値では 22MB, コストモデルでは約 17.5MB となり概ね一致している.

BM25_Merge, Sort_Bitonic, Sort_Merge における実測値

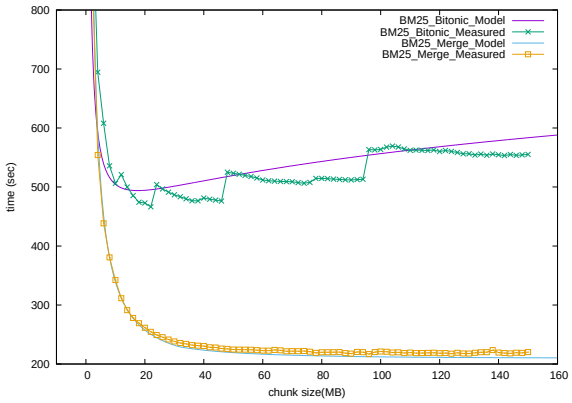

図 4　重み付け計算のコストモデルと実測値

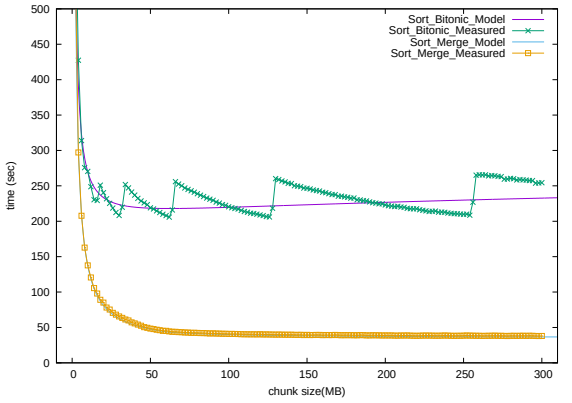

図 5　整数値ソートのコストモデルと実測値

とコストモデルのグラフをそれぞれ図 4, 5 に示す. いずれのコストモデルも実測値と概ね一致していることがわかる.

Sort_Bitonic について, 計算時間が最小となるチャンクサイズは, 実測値では 62MB, コストモデルでは約 62.3MB となり概ね一致している. BM25_Merge と Sort_Merge のコストモデルは, どちらも実験を行った範囲には計算時間を最小にするチャンクサイズが存在しないが, 計算時間の減少量が小さくなる時のチャンクサイズをコストモデルから算出することが可能である. 計算時間が大きく減少しないにもかかわらず, チャンクサイズを大きくすることはメモリ消費の観点から望ましくない. そこで, 計算時間が短くなる最適なチャンクサイズを算出することが可能である.

以上より, いずれのタスク, ソートアルゴリズムにおいてもコストモデルと実測値の挙動が一致することを確認し, 最適なチャンクサイズを計算で算出することが可能であることを明らかにした.

5. おわりに

本研究は, GPU 上で実装された MapReduce を用いて, 大規模なデータに対する処理の評価を行った. その際, 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. 今回評価を行った重み付け計算タスクと整数値ソートタスクは, バイトニックソートを使用する場合, 計算時間を最小にするチャンクサイズをコストモデルから算出することが可能であると判明した. 一方, マージソートを使用する場合, 計算時間を最小にするチャンクサイズは今回実験を行った範囲には存在しなかったが, メモリ消費と計算時間の観点から最適なチャンクサイズを算出することができると判明した.

今回評価を行った二種類のタスクと二種類のソートアルゴリズム以外の場合にも, 同様にコストモデルを立てることで最適なチャンクサイズを求めることが可能である.

今後の課題として, GPU を搭載したマシンを複数用いてスケールアウトを行うことが考えられる. スケールアウトを行うことで更に大きなデータセットに対しても高速に処理を行うことが可能である. その際, 同様にコストモデルを立ててチャンクサイズの評価を行いたい.

6. 謝辞

本研究の一部は, JSPS 科研費 15K20990, 17K12684, 15H02701, 16H02908 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] 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 Co-processor 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