Vol.2016-HPC-154 No.11
2016/4/25

GPUアクセラレータと不揮発性メモリを考慮した大規模分散ソート (Unrefereed Workshop Manuscript)

社本 秀之[1,2]　佐藤 仁[1,2]　松岡 聡[1,2]

概要：我々は，GPU のデバイスメモリ，ホストメモリ，SSD で構成されたバーストバッファ，並列ファイルシステムの多階層のメモリ・ストレージを考慮した Disk-to-Disk の大規模分散ソートを提案する．提案手法は，一般的なスプリッタを用いる分散ソートに基づくがファイル I/O の性能低下を抑えるために，データ圧縮を積極的に取り入れる．提案手法の有効性について TSUBAME2.5 上で検証し，128 台の計算ノード上の 128GPU を使用して 128GB の 32bit 整数のレコードのソートでデータ非圧縮の手法と比較して 1.56 倍の性能向上を確認した．さらに，性能モデルを用いてバーストバッファの効果を調査し，バーストバッファが並列ファイルシステムへの I/O を仲介することで，I/O スループットが 4 倍になれば，128 台の計算ノードでのソート処理を 1/4 の計算ノード数 (32 台) で実行したとしても 1.06 倍の性能で処理することが可能であることを確認した．

1. はじめに

ソートは，データベース管理システム，ビッグデータミドルウェアフレームワーク，大規模データ処理を伴う科学技術計算アプリケーションなど様々なソフトウェアで基本的な処理であり，その高速化は常に重要な課題である．特に，ヘルスケア，システム生物学，ソーシャル・ネットワーク，ビジネスインテリジェンス，スマートグリッドなどのビッグデータアプリケーションでは大規模なデータセットの解析の高速化のために大容量の DRAM を必要とする．一般的には，大容量の DRAM を必要とする場合は，複数の計算ノードを用いることで利用可能なホストメモリの容量を増やし，大規模なデータセットを高速にソートするアプローチが採られる．とりわけ，GPU アクセラレータはソートの高速化に有効な技術であり，我々はこれまでに 1000 台を超える規模で複数の計算ノード上に搭載された GPU デバイスを利用して大規模分散ソートを行う試みなどを行っている [16].

DRAM の問題点として，容量あたりの導入コストの高さや消費電力の高さ，また，将来のスパコンのアーキテクチャに向けてはプロセッサのコア数の増大や DRAM を構成する半導体の集積度の限界などにより利用可能なコアあたりの DRAM のバンド幅・容量が少なくなることが挙げられる一方で，昨今，DRAM と比較して低バンド幅と高レイテンシだが大容量で低コストという特性を持ったフラッシュなどに代表される不揮発性メモリデバイスが登場し，バンド幅を必要としない処理に伴うデータセットを積極的に不揮発性メモリへオフロードすることで，アプリケーションが必要とするバンド幅と容量を稼ぐなどの活用が期待されている．とりわけ，スパコンのストレージにおいては，不揮発性メモリデバイスの一種である SSD を用いて構成されたバーストバッファと呼ばれるキャッシュ層を導入することで並列ファイルシステムの I/O 性能をアクセラレーションさせる試みが進められており，今後，実環境に普及していくことが期待されている．このため，将来のスパコンのアーキテクチャ向けて，容量や帯域にトレードオフを持つメモリ階層を活用することによってアプリケーションを高速化することが求められるが，外部記憶を伴う Disk-to-Disk の大規模分散ソートに関して多階層のメモリ・ストレージを効率的に活用する手法や効果は明らかでない．

そこで，我々は，GPU のデバイスメモリ，ホストメモリ，SSD で構成されたバーストバッファ，並列ファイルシステムの多階層のメモリ・ストレージを考慮した Disk-to-Disk の大規模分散ソートを提案し，その実装を既存のスパコン上で実行した結果をもとに，将来のスパコンアーキテクチャを想定した性能モデルを構築して性能予測を行う．提案手法

1 東京工業大学
2 科学技術振興機構, CREST

は，一般的なスプリッタを用いる分散ソートに基づくがファイル I/O の性能低下を抑えるために，データ圧縮を積極的に取り入れる．提案手法の有効性について TSUBAME2.5 上で検証し，128 台の計算ノード上の 128GPU を使用して 128GB の 32bit 整数のレコードのソートでデータ非圧縮の手法と比較した結果，1.56 倍の性能向上を確認した．さらに，性能モデルを用いてバーストバッファの効果を調査し，バーストバッファが並列ファイルシステムへの I/O を仲介することで，I/O スループットが 4 倍になれば，128 台の計算ノードでのソート処理を 1/4 の計算ノード数 (32 台) で実行したとしても 1.06 倍の性能で処理することが可能であることを確認した．

2. 大規模分散ソート

2.1 スプリッタを用いた分散ソート

現在の高速な大規模分散ソートの多くは、Sample Sort [5], [17] や Histgram Sort [9] などスプリッタを用いたアルゴリズムが採用されている．これは，各プロセスが担当するレコードの範囲をスプリッタを用いて限定することで，プロセス間通信やストレージへの I/O などのレコードの移動に伴うデータ量を削減し，性能低下を抑えることができるためである．例えば，分散環境での典型的な Sample Sort では以下のような処理を行う．

STEP1 各プロセスが担当するレコードをソートする．

STEP2 各プロセスが担当するレコードの中で $0, n/p^2, 2n/p^2, \cdots, (p-1)/(n/p^2)$ 番目のインデックスのレコードをサンプリングし，サンプルとする．

STEP3 各プロセスがサンプリングしたレコードを 1 つのプロセスへ集約し，それらをソートする．この際，全てのサンプルを合わせた個数は p^2 個となる．

STEP4 ソートされたサンプルの集合において，$p + \lfloor p/2 \rfloor - 1, 2p + \lfloor p/2 \rfloor - 1, \cdots, (p-1)p + \lfloor p/2 \rfloor$ 番目のインデックスのレコードをスプリッタとする．この際，スプリッタの個数は $p-1$ 個となる．

STEP5 選ばれた $p-1$ 個のスプリッタを各プロセスへ転送する．

STEP6 各プロセスは転送されたスプリッタを用いてレコードを分割し，対応するプロセスにレコードを転送する．このとき，各プロセスはスプリッタで定められた範囲のレコードを担当する．

STEP7 各プロセスは転送されてきたレコードをソートすることにより，ソート列を得る．

スプリッタを用いた大規模分散ソートでは，プロセス間のデータの移動量を削減できる反面，プロセス内で担当するレコードに対するローカルなソート処理を行う必要があり，性能上のボトルネックとなる．

2.2 今後のスパコンアーキテクチャの動向

近年，Titan, Piz Daint, TSUBAME2.5 など数千〜数万台の GPU アクセラレータを搭載したスパコンが一般的になってきている．コモディティデバイスである GPU は，CPU と比較して大量のスレッドによるデータ並列処理が適用でき，高い演算ピーク性能やメモリバンド幅を持ち，性能に対する価格や消費電力が低いという利点がある．このため，今後も ORNL Summit や LLNL Sierra など GPU アクセラレータを搭載した Pre-exa 級のスパコンが計画されている．

一方，スパコンのストレージに関しては，従来は Lustre や GPFS などの並列ファイルシステムを用いてストレージを構成し，並列 I/O を行うことで I/O 性能を達成してきた．しかし，近年，計算ノードと並列ファイルシステムの I/O 性能の差を埋めるために SSD を活用し，スパコンを構成するということが行われている．例えば，LLNL Catalyst，東工大 TSUBAME2.5, SDSC Gordon, Dash [8] では，計算ノード上に SSD を搭載することで I/O 性能のアクセラレーションを行っている．また，昨今では，EMC aBBa, DDN IME など 計算ノードと並列ファイルシステムの中間に SSD で構成されたバーストバッファを設けることで並列ファイルシステムに対する I/O 性能をアクセラレーションする技術 [10], [13] も登場し，今後多くのスパコンで導入されていくことが期待される．

このように，今後のスパコンアーキテクチャでは，容量や帯域にトレードオフの関係があるメモリ・ストレージ階層が増加するため，それらをソフトウェア技術によりうまく活用することでアプリケーションの性能を向上させることが求められる．

2.3 既存の大規模分散ソートの問題点

大規模分散ソートでは，複数の計算ノードを用いることで利用可能なホストメモリの容量を増やし，大規模なデータセットを高速にソートすることができる．しかしながら，今後の DRAM の容量や半導体技術の深化を考慮すると巨大なデータセットをホストメモリのみでソートしていくことは現実的でない．そのため，並列ファイルシステムなど外部記憶に保存されているデータを高速にソートする Disk-to-Disk ソートに関する研究がなされている．一般的に，Disk-to-Disk ソートでは並列ファイルシステムへのファイル I/O は他の計算処理に比べて実行時間が非常に大きいため，近年，ホストメモリと比べて I/O 性能では劣るが容量の面では優れている SSD を用いて性能向上を目指す手法がとられる．例えば，文献 [19] では，並列ファイルシステムから読み込んだデータを計算ノードのローカルに搭載された SSD にキャッシュすることで並列ファイルシステムへの I/O の性能低下を軽減し，高速化を達成する．しかしながら，このアルゴリズムではローカルの SSD の

容量を超えてデータをキャッシュすることができず，ローカル SSD の合計量を超えるような巨大なデータセットをソートすることはできない．

また，将来のスパコンアーキテクチャに向けても，GPU アクセラレータやバーストバッファなどプロダクション環境で利用可能な現実的な新しいソリューションが登場している．特に，バーストバッファの多くの実装では，並列ファイルシステム上のファイルのメタデータは透過的にアクセスすることができるため，上記のような容量の制限による問題は基本的にはないものの，一方で，アプリケーション側からは，GPU アクセラレータの活用など，容量や帯域にトレードオフを持つメモリ階層を用いて高速化することが求められるが，このような多階層のメモリ・ストレージを効率的に活用する手法や効果については明らかでない．

3. 関連研究

多くのソートはメモリバンド幅がボトルネックになる処理であるため，メモリバンド幅が大きい GPU を使用したアルゴリズムが広く研究されている．1 台の GPU デバイスを対象としたソートアルゴリズムは数多く提案されており，GPU radix sort [14] や GPU を使った比較ソートである GPU Quicksort [2]，GPU Sample sort [11]，GPU Bitonic sort [7]，GPU Merge-based string sort [4] 等がある．GPU メモリを超えるようなサイズのデータに関するソートも研究されており，サンプルベースの外部ソート [21] や，マージベースの外部ソート [15] が提案されている．これらのソートでは、1 つの GPU デバイスを各プロセスが活用しているが，1 つのプロセスが複数の GPU デバイスを活用するようなソートの研究もなされている [20]．このソートでは複数の GPU デバイスを活用することができるが，対象としている環境は単一ノードであり，複数ノード向けのアルゴリズムではない．Spafford らは GPU を活用した分散ソートとして，radix sort をベースとしたソートを提案している [18]．彼らはノード内では複数 GPU を使用した radix sort を行い，更にデータを MPI の all-to-all 通信を行うことで複数ノード間でのソートを実現している．これらのアルゴリズムは，ソート処理を高速に行うことを可能にしているが，GPU のメモリを超過するような巨大なデータセットを処理することはできない．

4. 提案手法

4.1 提案アルゴリズムの概要

我々は，GPU のデバイスメモリ，ホストメモリ，SSD で構成されたバーストバッファ，並列ファイルシステムの 4 つのメモリ階層を考慮し，I/O 性能のボトルネックを解消するためにデータ圧縮を積極的に取り入れた大規模分散ソートを提案する．提案手法は，2.1 節で述べた Sample Sort を基にし，大きく分けて，(1) 各プロセスで入力レコードをスプリッタが定めた境界値で分割するフェーズと，(2) 各プロセスでスプリッタにより分割したレコードを 1 つのバケットとして集約し，各バケットのレコードをソートする，という 2 つのフェーズに分けられる．図 1 に提案手法が前提とするスパコンのアーキテクチャの概要を示す．計算ノードと並列ファイルシステム間に SSD を搭載したノードによりバーストバッファを構成し，バーストバッファを介して並列ファイルシステムからの I/O を行う．既存のバーストバッファの多くの実装では，並列ファイルシステム上のファイルのメタデータは透過的にアクセスすることができるため，提案手法でもバーストバッファを介しても透過的に並列ファイルシステム上のファイルへアクセスできることを想定している．具体的なアルゴリズムは以下のとおりである．

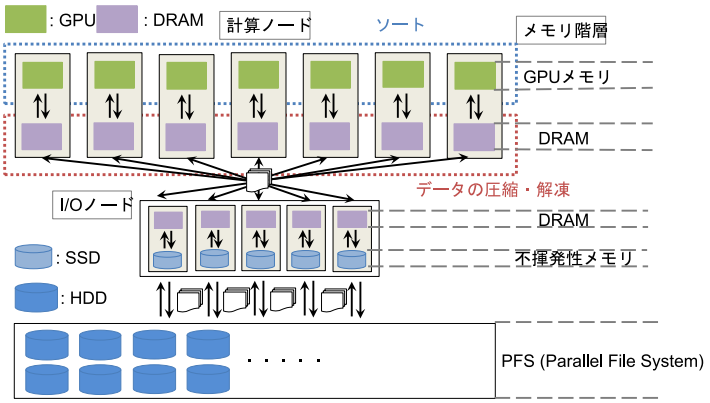

図 1　提案手法が前提とするスパコンのアーキテクチャの概要

Phase1

Step1-1　バケットへ分割する際の境界値となるスプリッタを選択する．(*select_splitters*)

Step1-2　ファイルからホストメモリへ予め決められたサイズのレコードをチャンクとして読み込む．(*rbw_read*)

Step1-3　ホストメモリから GPU のデバイスメモリへチャンクを転送し，GPU 上でチャンク内のレコードのソートを行う．その後，デバイスメモリからホストメモリへチャンクを転送する．(*rbw_sort*)

Step1-4　各プロセスで，チャンク内のレコードをスプリッタによりいくつかのセグメントに分割する．この際にはレコードの位置情報のみを記録する．これらのセグメントは，Phase2 で各プロセスで同一のスプリッタで区切られたセグメントを集約し，バケットとする．(*rbw_bucket*)

Step1-5　各セグメントに対して圧縮を行う．(*rbw_compress*)

Step1-6　ホストメモリ上にある圧縮されたセグメントを中間ファイルとして書き出す．(*rbw_write*)

Step1-7　Step1-2 から Step1-6 までを未処理のチャ

ンクがなくなるまで繰り返す．

Phase2

Step2-1 各プロセスが担当するバケットを決定する．

Step2-2 プロセス毎に，担当するバケット (セグメントの集合) を Phase1 で生成した中間ファイルからホストメモリへ読み込む．(*rsw_read*)

Step2-3 圧縮されたバケットのデータを解凍し，元のレコードへ復元する．(*rsw_uncompress*)

Step2-4 ホストメモリから GPU のデバイスメモリへ復元されたバケットを転送し，GPU 上でバケット内のレコードをソートする．その後，デバイスメモリからホストメモリへバケットを転送する．(*rsw_sort*)

Step2-5 Step2-2 から Step2-4 までを各プロセスが担当するバケットがなくなるまで繰り返す．

4.2 データ圧縮

一般的に，Disk-to-Disk ソートでは，実行時間の大部分が並列ファイルシステムへの I/O の実行時間で占められ，性能上のボトルネックとなる．そこで，提案手法では並列ファイルシステムへ行う I/O のデータ量を削減することで I/O 性能の向上を試みる．提案アルゴリズムにおいては，Phase1 の中間ファイルへの書き出し，及び，Phase2 の中間ファイルからの読み込みが I/O 性能のボトルネックとなる．そこで，Step1-4 のチャンクのセグメントへの分割の直後に圧縮をかけ，書き込みを行う．また，Step2-2 でバケットの読み込みの際に，読み込んできたセグメント毎に解凍を行い，元のバケットに復元する．

現在の実装では，SIMDComp [12] という整数圧縮用のデータ圧縮ライブラリを採用している．そのため，現在は 32bit 整数のみ対応している．Step1-4 でセグメント化された地点ではレコードはソートされた状態であるため，連続した要素間のデルタ値を使用し圧縮することで，ランダムなレコードに対しても高い圧縮率を達成する．64bit 整数や他のデータ型への対応に関しては，Snappy [6] や LZ4 [3] をはじめとする他の既存の圧縮ライブラリへ入れ替えることで対応することが可能である．

4.3 ファイルアクセスパターン

文献 [1] において指摘されているように，並列ファイルシステムに対する I/O は，そのアクセスパターンによって性能が大きく変化する．そのため，我々は，(1) 入力ファイル，中間ファイル，出力ファイルが 1 ファイルずつで管理される 1-1-1 パターンと，(2) 入力ファイル，中間ファイルがプロセス数と同数の n 個，出力ファイルがバケット数と同数の b 個であるの n-n-b パターンとの 2 通りのシナリオを想定した．前者は，単一のファイルに対するアクセスとなるため，ファイルの管理が楽であり，並列ファイルシステムにとって負荷の高い細粒度なファイルアクセスを避けることができるが，各プロセスがファイルへアクセスする際のオフセットを正確に求める必要がある．一方，後者は，ファイルアクセスの際のオフセットを容易に求めることができるが，多くのファイルを管理する必要があり，並列ファイルシステムにとって負荷の高い細粒度なファイルアクセスを伴う．

(1) 1-1-1 パターンの実装は次のようになる．まず，ソートの対象となる 1 つのファイルのファイルサイズを取得し，それをプロセス数で均等なチャンクへ分割する．各プロセスが分割されたチャンクを読み込み，GPU 上でソートを行った後，ホストメモリ上でチャンクをスプリッタでセグメントに分割する．セグメントの分割は実際にはオフセットのみを記録するため，中間ファイルへはソートされ圧縮されたチャンクを書き込みすることになる．そのとき，ファイルの読み込みを行ったオフセットに対して書き込みを行う．Phase1 が終わった後，バケットのフェイズで記憶しておいたオフセットを元に，バケットされたデータを各プロセスが読み込み，ソートを行う．Phase2 の開始時に各バケットのサイズを計算しておき，バケットサイズに基づい出力ファイルのバケットごとのオフセットを計算しておく．ソートされたデータはそのオフセットに書き込まれ，1 つの出力ファイルとしてソートされた状態となる．

(2) n-n-b パターンでは，まず入力ファイルがプロセス数分のファイルに分割されているという状態からソートを始める．それぞれのプロセスが対応するファイルに対してデータの読み込み，ソート，バケットを行っていく．バケット後の書き込みサイズは，各プロセスの入力ファイルと同じであり，そのファイルと同じサイズのファイルが中間ファイルとして生成される．この処理は各プロセスが行い，全中間ファイル数はプロセス数と同じになる．Phase2 において，各プロセスは担当するバケットのデータを全中間ファイルからそれぞれ読み込んで来た後，それらをまとめてソートし，ソートされた状態のバケットを得る．ソートされたバケットはそのまま出力ファイルとして書き込まれる．よって，最終的な出力ファイルはバケット数と同数となる．この実装では，Phase2 でのデータ読み込みの際に 1-1-1 パターンと同様な I/O パターンになってしまうのを避けるため，一度にアクセスするファイルはできるだけバラバラになるようにアクセスの順番を設定している．

5. 予備評価

5.1 実験設定

まずはじめに，提案アルゴリズムの基本的な性能を既存のスパコンで評価するための実験を行った．実験は東京工業大学のスーパーコンピュータ TSUBAME2.5 で行った．各計算ノードは，Intel Xeon X5670 (12 コア) の CPU を 2

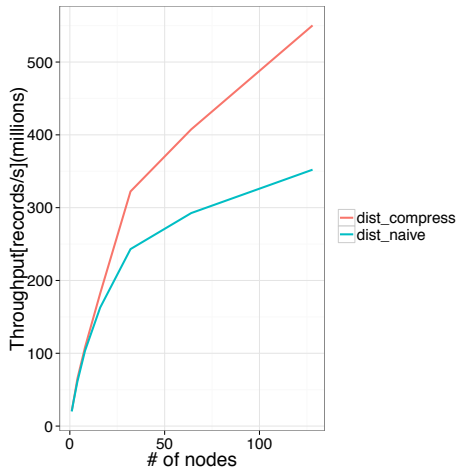

図 2　提案アルゴリズムのスケーラビリティ

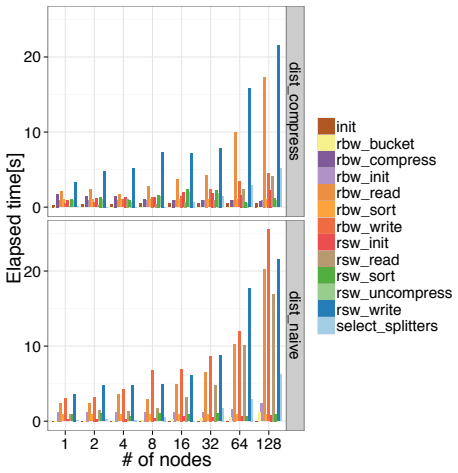

図 3　提案アルゴリズムの実行時間のブレイクダウン

ソケット，NVIDIA Tesla K20X の GPU を 3 台を演算装置とし，ホストメモリとして計 54GB の DDR3 メモリを搭載する．ネットワークへは計算ノード上の 2 つの 40Gbps QDR InfiniBand の HCA (Host Channel Adapter) を介して Dual rail，フルバイセクション・ファットツリー構成のインターコネクトへ接続される．並列ファイルシステムは Lustre の version 2.1.6 であり，同じインターコネクトへ接続されている．計算ノードでは Linux 3.0.76 OS が動作し，コンパイラ・ライブラリは，OpenMPI v1.8.3，CUDA v6.0，Thrust v1.7.1，SIMDComp v0.0.3 を用いた．

5.2　基本性能

提案アルゴリズムの基本的な性能を評価するために，計算ノードを 1 台から 128 台まで増加させたときのスケーラビリティについて実験を行った．1 台の計算ノードあたり 1 プロセス，1 プロセスあたり 1GB のメルセンヌ・ツイスタにより生成した一様ランダムな 32bit の整数からなるレコードを扱い，弱スケーリングを評価した．ファイルへの I/O は n-n-b パターンとしている．図 2 に結果を示す．図中で dist_compress は 4.2 節で述べたデータ圧縮手法を導入したもの，dist_navie はデータ圧縮手法を導入しないものを表す．x 軸は計算ノードの台数を表し，y 軸はスループットを表す．dist_naive では計算ノードが 32 台程度で性能が飽和しはじめているのに対し，dist_compress では計算ノードが 128 台でも比較的良好にスケールしていることが伺える．今回の評価では，総プロセス数や 1 プロセスあたりのレコード数が比較的小さい場合を行っているが，これらの数をさらに大きくした場合は，現状よりも処理効率が上がるため性能が向上することが期待される．

次に，上記の実行の際の実行時間のブレイクダウンについても調査した．図 3 に結果を示す．上図 (dist_compress) がデータ圧縮を行った場合で，下図 (dist_naive) がデータ圧縮を行わない場合の結果である．図中の凡例は， 4.1 節

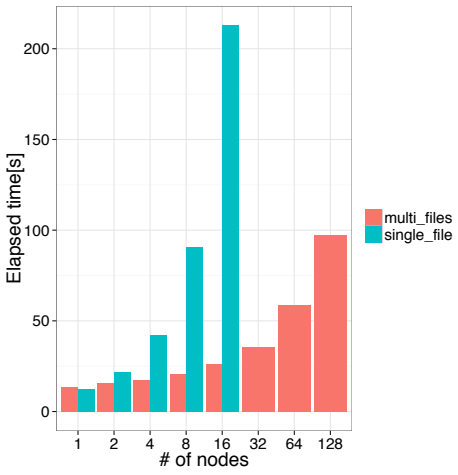

図 4　I/O パターンの違いによる性能比較

で述べた提案アルゴリズムの個々の手順に対応する．ただし，init は全体処理の初期化，rbw_init は Phase1 の初期化，rsw_init は Phase2 の初期化を表す．Phase1 において中間ファイルへの書き込みを行う rbw_write と Phase2 で中間ファイルからの読み込みを行う rsw_read の部分は，データ圧縮を導入することによりそれぞれ 5.59 倍, 4.18 倍の性能向上を示している．これにより，128 台の計算ノードの実行では 1.56 倍の性能向上を達成している．

最後に，4.3 節で述べた 2 種類の I/O パターンについて比較実験を行った．図 4 に 16 ノードまでの結果を示す．これより，n-n-b パターンの方が良好な結果を示すことを確認した．一般的に，単一ファイルへの多数のセグメントとなるアクセスは，ファイル I/O の性能低下を引き起こす．以降，ファイルアクセスパターンは n-n-b パターンであるとする．

6.　将来のスパコンアーキテクチャに向けた性能予測

前節では，既存のスパコン上の並列ファイルシステムを

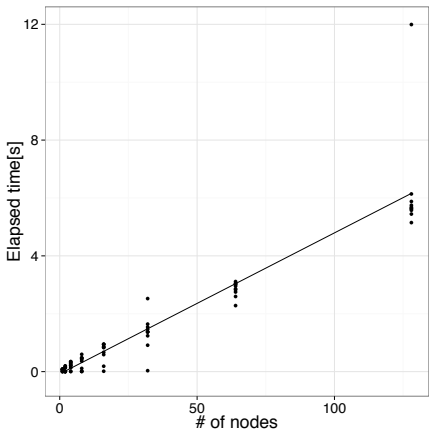

図 5　スプリッタ選択の実行時間の予測結果

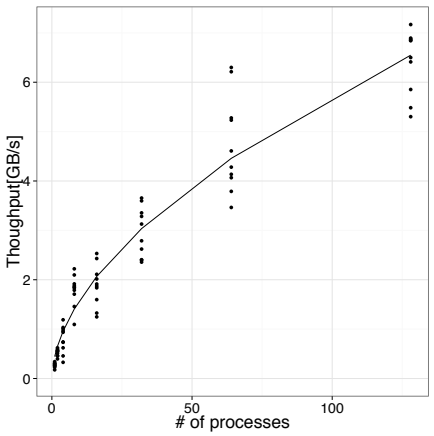

図 6　並列ファイルシステムへの書き込み I/O スループットの予測結果

用いて，提案アルゴリズムの性能の予備評価を行った．この節では，不揮発性メモリを用いたストレージアクセラレーション技術を用いた場合にどの程度の性能が期待できるかを評価するために，提案アルゴリズムについての性能モデルを構築し，将来のスパコンアーキテクチャを想定した性能予測を行う．提案アルゴリズムでは，並列ファイルシステムへの I/O を行わない部分は，各プロセスが同期を取る必要がなく，計算ノードあたりが扱うデータ量が等しければ 1 ノードで処理する時間と全ノードで処理する時間は変わらないはずである．これは図 3 を見ても明らかで，I/O 以外のフェイズは計算ノード数の増減によってあまり変化していない．そこで，今回は，スプリッタ選択の実行時間と，ファイル I/O が行われる並列ファイルシステムへの読み込み/書き込みの際の I/O スループットについての性能モデリングを行い，それ以外のステップは 1 ノードの時のスループットを使用して性能予測を行う．

6.1　スプリッタ選択の実行時間の性能モデル

現在の実装では，スプリッタの選択の際の実行時間はプロセス数に大きく依存する．これは，スプリッタを選択するためのサンプル数がプロセス数の定数倍となっているためである．サンプル数と同じ回数だけランダムな読み込みが行われ，サンプリング数と同数のレコードについてソート処理が行われる．よって，我々の実装におけるスプリッタ選択のステップはプロセス数に依存すると仮定し，線形回帰によってモデル式を得た．結果は図 5 のようになる．ほぼすべての場合で中央値を取るような直線となっており，やはり実行時間が線形に伸びていることが確認できた．今回の実装では，プロセス数の増加により実行時間が大幅に大きくなってしまうため，サンプルを一度の読み込みで複数取得する等の工夫により実行時間を削減することなどの対策が考えられる．

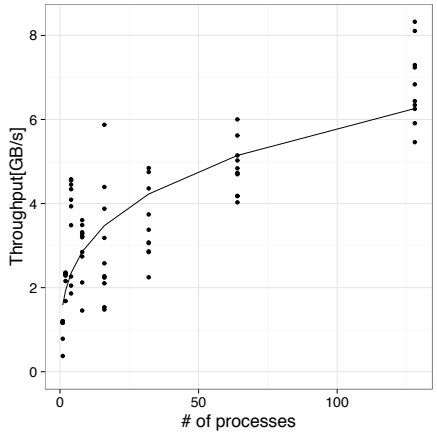

図 7　並列ファイルシステムへの読み込み I/O スループットの予測結果

6.2　I/O スループットの性能モデル

提案アルゴリズムでは並列ファイルシステムに対する I/O が読み込み，書き込み各々 2 回ずつあり，それらの性能予測をすることが必要である．そこで，実際にソート処理で計算ノード数をスケールさせたときの I/O のスループットの変化を測定し，モデル式にフィッティングすることで，性能予測を行う．並列ファイルシステムに対する I/O 性能は，基本的には実行の並列度が上がるほど向上するが，ストレージが許容する I/O 性能を超える場合は I/O 性能が飽和する．そこで，実際に測定した値を当てはめるモデルとして，説明変数が大きくなるほど目的変数が上がりづらくなるという性質を表現できるモデルである累乗モデルを採用した．図 6 に書き込みの I/O スループット，図 7 に読み込みの I/O スループットの予測結果を示す．書き込み I/O，読み込み I/O ともに同様の傾向を示している．

6.3　バーストバッファを前提とした提案アルゴリズムの性能モデル

スプリッタ選択の実行時間は 6.1 節の性能モデル，I/O

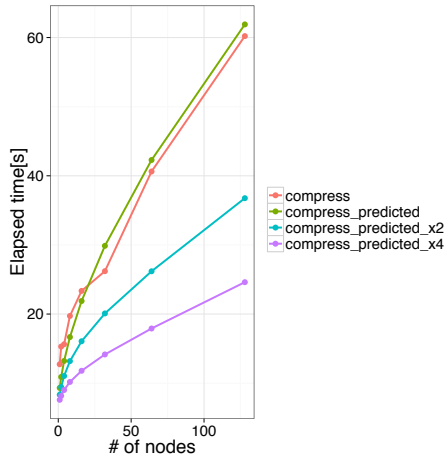

図 8　バーストバッファの I/O スループット向上による提案アルゴリズムの実行時間の予測結果

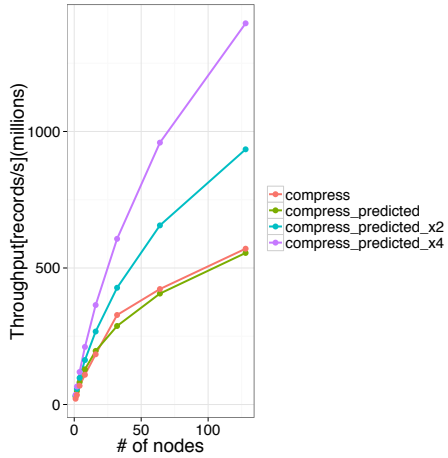

図 9　バーストバッファの I/O スループット向上による提案アルゴリズムの性能 (スループット) の予測結果

の実行時間は 6.2 節の I/O スループットのモデルをもとに算出し，それ以外のステップの実行時間は 1 ノードの時のデータを元にすることで，提案アルゴリズムのソート処理全体の実行時間を推定することが可能である．ここで，将来のスパコンアーキテクチャを想定したいので，並列ファイルシステム上のファイルへアクセスする際に SSD で構成されたバーストバッファを介することで単純に I/O のスループットが向上すると仮定する．図 8 に実行時間の予測結果，図 9 に図 8 を基に算出した性能 (スループット) の予測結果を示す． 5 節の図 3 の結果により，SIMDComp によるデータ圧縮によって Phase1 の書き込み，Phase2 の読み込みのデータ量がほぼ 4 分の 1 となるとし，I/O の実行時間を変動させる．図中で，compress は実測値で，compress_predicted は実測値をもとにした性能モデルによる推定値になっている．また，compress_predicted_x2, compress_predicted_x4 はそれぞれ I/O スループットが 2 倍，4 倍になった時を仮定した実行時間の推定値である．計算ノード数が少ない時は実測値と推定値の差が大きいが，これは累乗モデルが説明変数が小さい時の目的変数の上がり幅が大きくなってしまうことが要因であると考えられる．これに対し，計算ノード数が大きくなると，性能の上がり幅が小さくなることが表現できており，実測値との乖離が小さくなり，よくモデルにあっていると言える．

図 9 の結果より，バーストバッファにより，同じ計算ノードの台数で同じデータ量を高速に処理できることがわかる．例えば，バーストバッファの I/O スループットが 2 倍になると 128 台の計算ノードのソート処理で 1.64 倍の性能を達成し，I/O スループットが 4 倍になると 2.44 倍の性能を達成する．また，128 台の計算ノードでのソート処理に対し，I/O スループットが 2 倍になると半分 (64 台) の計算ノードで実行したとしても 1.14 倍の性能でソート処理することが可能であり，I/O スループットが 4 倍になると 1/4(32 台) の計算ノードで実行したとしても 1.06 倍の性能でソートすることが可能である．バーストバッファを導入しメモリ・ストレージの階層性を考慮することにより計算ノードの台数を減らしたとしても著しい性能低下は起こらず，効率的にソート処理を行うことが可能であることが伺える．

これまで，バーストバッファによる I/O の仲介によってソート処理全体の性能が向上することを述べた．一方で，ある一定以上バーストバッファのスループットが向上しても，I/O の高速化による提案アルゴリズムの性能の変化は小さくなる．図 10 は，計算ノードの数を 128 台に固定して，バーストバッファによって I/O のスループットが 1 倍から 15 倍まで高速化した場合の性能を表している．2 倍になった時には 1.64 倍の I/O スループットを達成しているが，3 倍以降はその性能の伸びは非常に小さくなることがわかる．I/O スループットを 15 倍にした時の性能の 80%の性能は I/O スループットが 7 倍の時に既に達成されており，この傾向はスループットが増え続けても変わらないことから，これ以上スループットが向上することによるソート処理への利点は非常に少ないと考えられる．

7. おわりに

GPU のデバイスメモリ，ホストメモリ，SSD で構成されたバーストバッファ，並列ファイルシステムの多階層のメモリ・ストレージを考慮し，ファイル I/O の性能低下を抑えるためにデータ圧縮を積極的に取り入れた Disk-to-Disk の大規模分散ソートを提案した．提案手法の有効性について TSUBAME2.5 上で検証し，128 台の計算ノード上の 128GPU を使用した 128GB の 32bit 整数のレコードのソートでデータ非圧縮の手法と比較して 1.56 倍の性能向上を確認した．さらに，性能モデルを用いてバーストバッファの効果を調査し，バーストバッファが並列ファイルシステムへの I/O を仲介することで，I/O スループットが 2 倍になれば，128 台の計算ノードでのソート処理を 1/4 の計

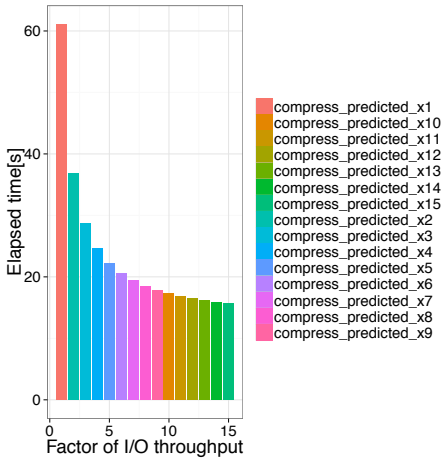

図 10　計算ノード台数が一定 (128 台) の場合のバーストバッファのスループット向上による提案アルゴリズムの実行時間の予測

算ノード数 (32 台) で実行したとしても 1.06 倍の性能で処理することが可能であることを確認した．これらの結果から，今後の大規模なデータ処理のアプローチとして，計算ノード数を増加させるだけでなく，バーストバッファなど階層的なメモリ・ストレージの積極的な活用が重要であることが伺える．

今後の課題としては，提案アルゴリズムの実装の最適化をはじめとして，一様ランダムなレコードの分布以外のデータセットへの対応，32bit 整数以外のデータ圧縮手法の適用などが挙げられる．

謝辞　本研究の一部は JST CREST「EBD：次世代の年ヨッタバイト処理に向けたエクストリームビッグデータの基盤技術」，「ポストペタスケールシステムにおける超大規模グラフ最適化基盤」，及び，JSPS 科研費 26540050 の助成を受けたものである．

参考文献

[1] Bent, J., Gibson, G., Grider, G., McClelland, B., Nowoczynski, P., Nunez, J., Polte, M. and Wingate, M.: PLFS: a checkpoint filesystem for parallel applications, *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ACM, p. 21 (2009).

[2] Cederman, D. and Tsigas, P.: A practical quicksort algorithm for graphics processors, *Algorithms-ESA 2008*, Springer, pp. 246–258 (2008).

[3] Collet, Y.: LZ4, https://github.com/google/snappy.

[4] Davidson, A., Tarjan, D., Garland, M. and Owens, J. D.: Efficient parallel merge sort for fixed and variable length keys, *Innovative Parallel Computing (InPar), 2012*, IEEE, pp. 1–9 (2012).

[5] Frazer, W. D. and McKellar, A.: Samplesort: A sampling approach to minimal storage tree sorting, *Journal of the ACM (JACM)*, Vol. 17, No. 3, pp. 496–507 (1970).

[6] Google: Snappy, https://github.com/google/snappy.

[7] Govindaraju, N., Gray, J., Kumar, R. and Manocha, D.: GPUTeraSort: high performance graphics co-processor sorting for large database management, *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, ACM, pp. 325–336 (2006).

[8] He, J., Jagatheesan, A., Gupta, S., Bennett, J. and Snavely, A.: Dash: a recipe for a flash-based data intensive supercomputer, *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE Computer Society, pp. 1–11 (2010).

[9] Kale, L. and Krishnan, S.: A comparison based parallel sorting algorithm, *Proceedings of the 1993 International Conference on Parallel Processing (ICPP'93)*, Vol. 3, pp. 196–200 (1993).

[10] Kimpe, D., Mohror, K., Moody, A., Van Essen, B., Gokhale, M., Ross, R. and de Supinski, B. R.: Integrated in-system storage architecture for high performance computing, *Proceedings of the 2nd International Workshop on Runtime and Operating Systems for Supercomputers*, ACM, p. 4 (2012).

[11] Leischner, N., Osipov, V. and Sanders, P.: GPU sample sort, *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, IEEE, pp. 1–10 (2010).

[12] Lemire, D.: The SIMDComp library, https://github.com/lemire/simdcomp.

[13] Liu, N., Cope, J., Carns, P., Carothers, C., Ross, R., Grider, G., Crume, A. and Maltzahn, C.: On the role of burst buffers in leadership-class storage systems, *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, IEEE, pp. 1–11 (2012).

[14] Merrill, D. and Grimshaw, A.: High Performance and Scalable Radix Sorting: A case study of implementing dynamic parallelism for GPU computing, *Parallel Processing Letters*, Vol. 21, No. 02, pp. 245–272 (2011).

[15] Peters, H., Schulz-Hildebrandt, O. and Luttenberger, N.: Parallel external sorting for CUDA-enabled GPUs with load balancing and low transfer overhead, *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, IEEE, pp. 1–8 (2010).

[16] Shamoto, H., Shirahata, K., Drozd, A., Sato, H. and Matsuoka, S.: GPU-Accelerated Large-scale Distributed Sorting Coping with Device Memory Capacity, *IEEE Transactions on Big Data*, Vol. PP, No. 1, pp. 1–14 (2016).

[17] Shi, H. and Schaeffer, J.: Parallel sorting by regular sampling, *Journal of Parallel and Distributed Computing*, Vol. 14, No. 4, pp. 361–372 (1992).

[18] Spafford, K. L., Meredith, J. S. and Vetter, J. S.: Quartile and outlier detection on heterogeneous clusters using distributed radix sort, *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, IEEE, pp. 412–419 (2011).

[19] Sundar, H., Malhotra, D. and Schulz, K. W.: Algorithms for high-throughput disk-to-disk sorting, *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, ACM, p. 93 (2013).

[20] Tanasic, I., Vilanova, L., Jordà, M., Cabezas, J., Gelado, I., Navarro, N. and Hwu, W.-m.: Comparison based sorting for systems with multiple GPUs, *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, ACM, pp. 1–11 (2013).

[21] Ye, Y., Du, Z. and Bader, D. A.: Gpumemsort: A high performance graphic co-processors sorting algorithm for large scale in-memory data (2010).