Vol.2016-HPC-153 No.22
2016/3/2

Multiple PVAS を用いた MapReduce におけるファイル I/O 処理

佐藤 未来子[†1]　島田 明男[†2]　吉永 一美[†3]　辻田 祐一[†3]
堀 敦史[†3]　並木 美太郎[†1]

概要: XeonPhi 搭載計算機を活用した MapReduce アプリケーションにおいて，ファイル I/O を搭載していない XeonPhi 上で大量の MapReduce データを処理するためには，リモートファイルをマウントするなどホストや別ノードと連携してデータを取得する必要がある．本研究では，メニーコア混在型計算機向けプログラム実行基盤 Multiple PVAS(M-PVAS)を用いて，MapReduce フレームワーク内にメニーコア側から発行されたファイル I/O 要求をホスト側で代行処理する機能を設け，MapReduce データをファイルから効率よく取得する方式を提案する．本提案方式および既存方式との方式性能を比較評価した結果，M-PVAS の空間共有を活用したファイル I/O 処理により MapReduce 実行性能が約 2 割改善するという結果が得られた．

キーワード：MapReduce，XeonPhi，メニーコア，ファイル I/O

1. はじめに

近年，アプリケーションプログラムの高速化のためにメニーコアを採用し，システムに搭載するコア数を増やし性能向上を図る傾向にある．GPGPUやIntel XeonPhiなどのメニーコアを有効に活用する際には，GPGPUであればCUDAに代表される様なアクセラレータに特化した並列プログラミングにより，計算性能を最大限に引き出すための並列アプリケーションの開発が必要となる．一方で，近年ではMapReduceの分散処理フレームワークが注目されている[1]．MapReduceはプログラマが定義したMapとReduceと呼ばれる計算を並列分散処理させるためのフレームワークである．分散環境におけるHadoop[2]の他，京コンピュータにおけるKMR[3]，GPGPUやXeonPhiなどのメニーコア向けMapReduceフレームワーク[4][5][6][7][8]など，各種のプラットフォームでMapReduce実現方式が提案されている．これらのMapReduceのフレームワークは，OSやランタイムライブラリが提供する並列実行基盤を応用して実現されている．したがって，並列プログラミングに長けていないプログラマであっても，並列化したい処理をMapとReduceで構築することで並列分散処理を簡単に行えるという特徴がある．

本研究では，Intel XeonPhiを搭載するヘテロジニアスアーキテクチャシステムを対象としたMapReduceフレームワークの実現方式を提案している[9]．既存のXeonPhi向けMapReduceフレームワーク[6][7][8]では，XeonPhi上のメモリにMapReduce処理対象データを設定して実行するフレームワーク設計となっている．すなわち，XeonPhiのデバイス上メモリの容量を超える大量データを扱えないという問題が残されている．XeonPhiを活用して大量のMapReduceデータを処理するためには，XeonPhiにおいてホストや別ノードと連携した，ファイルI/Oを用いたデータ処理が必要だと考える．本研究では，メニーコア混在型計算機向けプログラム実行基盤Multiple PVAS (M-PVAS) [10]を用いて，XeonPhi側で行うMapReduce処理からのファイルI/O要求をホスト側で代行処理する機能を設け，MapReduce対象データをファイルから効率よく取得する方式を提案する．本提案方式および既存OSでの実現方式との方式性能を比較評価し，M-PVASの空間共有を活用したファイルI/O処理の有効性を示す．

2. 関連研究

MapReduceは並列分散処理のためのフレームワークとして様々な並列計算機環境で実現されている．本研究のように，メニーコアによるアクセラレーションを適用したMapReduce フレームワークもいくつか提案されている．Phoenix++[6] や MRPhi[7] は汎用 CPU を対象とした MapReduce フレームワークであり，Intel XeonPhi 上での並列処理も可能である．スレッドベースで master/worker モデルで Map 処理, Reduce 処理を並列化する．Mars[4]は GPGPU へ並列計算をオフロードするための MapReduce フレームワークを提案しており，近年では複数の GPGPU を対象とする計算オフロード方式も提案されている[5].

本研究で対象としている Intel XeonPhi では，MRPhi [7] においてマルチコア CPU 向けに提案された Phoenix++ [6] を XeonPhi 向けに Map 処理のベクトル化，SIMD 命令によるハッシュ管理などの最適化が提案されている．これにより XeonPhi 上での Map 処理，Reduce 処理の高速化が実現されている．そして MrPhi[8]において，MPI による分散オペレーションによりホストから複数台の XeonPhi 上の MapReduce 処理を分散させ，データ処理させる方式が提案されている．これらの研究では，既存の汎用 OS で提供される POSIX thread や MPI といった並列実行基盤を用いて，メモリに配置したデータを処理対象とする MapReduce フレームワークを提供している．すなわち，これらの MapReduce フレームワークでは，ファイル I/O 処理については言及されておらず，XeonPhi のデバイス上メモリの容

†1 東京農工大学
†2 (株)日立製作所
†3 理科学研究所 AICS

量を超える大量データを扱えないという問題が残されている．

3. XeonPhi におけるファイル I/O の問題点

XeonPhi 搭載計算機を活用した MapReduce 処理において，ファイル I/O を搭載していない XeonPhi 上で大量の MapReduce データを処理するためには，ホストや別ノードと連携してファイルを扱うことで大量のデータ処理が可能となる．既存の XeonPhi 向けフレームワークでファイルからデータを取得する方式として，次の二種類をとりうる．

(1) XeonPhi でのリモートファイル共有方式
(2) ホストでファイルアクセスを代行する方式

以下，それぞれの方式とその問題点について示す．

3.1 XeonPhi でのリモートファイル共有方式

XeonPhi上で稼動するLinuxでは，Intel Manycore Platform Software Stack (MPSS)のサポートにより，XeonPhiからホスト上のファイルをNFSでアクセスすることが可能である．また，XeonPhi搭載計算機にInfiniband Host Channel Adapter（IB HCA）を装備している場合には，NFSやLustreサーバとのファイル共有も可能である[11].

しかし，XeonPhi 搭載計算機において，XeonPhi 側のプロセスからホストあるいは別ノード上のファイルを読むことを想定した予備評価を行ったところ，表 1 に示すとおり，ホストにおける read 性能の約 200MB/s の半分にも満たないことが判明した．このことは MapReduce 処理において XeonPhi 上でファイル I/O を実行した場合に I/O 性能がボトルネックとなり MapReduce 処理性能の低下の要因となりうると考えた．

表 1 XeonPhi でのリモートファイルアクセス性能

アクセス方式		性能
PCIe 経由ホストアクセス性能	NFS	20MB/s
Infiniband HCA 経由リモートサーバアクセス性能	NFS	31MB/s
	Lustre	77MB/s

（評価環境：WESTERN DIGITAL 製 WD1000CHTZ, Data transfer rate (max)：200MB/s (Host to/from drive (sustained))）

3.2 ホストでファイルアクセスを代行する方式

本研究の先行研究として，メニーコアを並列演算処理に専念させるために，メニーコア上プロセスからのファイルアクセスをホスト側で代行させるI/O デリゲーション方式を提案している[12]．このI/O デリゲーション方式における課題は，処理の委託に伴うデータ転送やプロセッサ間通信に伴うオーバヘッドであり，これらを可能な限り削減することが課題となる．

既存の Xeon Phi 搭載計算機において，Xeon Phi からファイル I/O の要求をホストへ送信し，ホストが I/O を代行する場合のアクセス方式を図 1 に示す．既存のシステム構成では，ホストと XeonPhi ではそれぞれ別 OS がアドレス空間を管理しているため，XeonPhi 側プロセスの I/O バッファに対して，直接ホスト側からデータ転送をかけることはできない．そのため，既存システムでは，ホスト上プロセスでファイルから読み出したデータを一旦メモリ上に保持した後，MPI 等のプロセッサ間通信によりホストから Xeon Phi へデータ転送する必要がある．このファイル I/O デリゲーションに伴うデータコピーのオーバヘッドを削減し，ファイル I/O 処理性能を向上させることが課題となる．

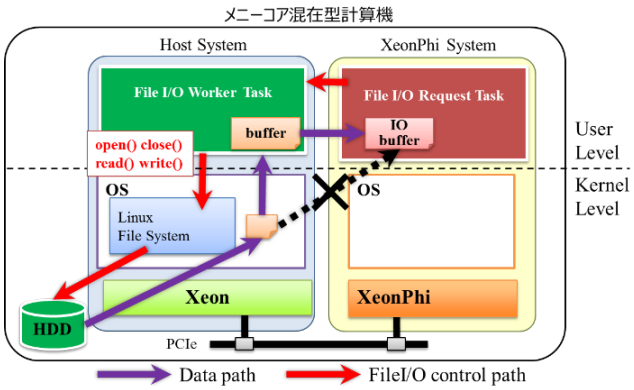

図 1 既存システムにおける XeonPhi 側プロセスからホスト側ファイルへのアクセス方式

4. 目標

本研究では，メニーコア混在型計算機において，

- メニーコアの並列度を生かした MapReduce 処理
- 効率のよいファイル I/O を用いた MapReduce 処理

を可能とする MapReduce フレームワークを目指す．メニーコアの並列度を生かした MapReduce 処理については，既存研究の MapReduce フレームワーク MRPhi[7]で提案されている XeonPhi 向けの最適化手法を採用することとし，本研究では，MapReduce フレームワークモデルにおける効率のよいファイル I/O 処理方式を提案することを目標とする．3.1 節で述べたように，XeonPhi 上でリモートファイルを共有する方式は性能面で問題があるため，本研究では，3.2 節で述べたホストでのファイル I/O 代行方式を採用する．そして，ホスト側から直接 XeonPhi のデバイスメモリへファイル I/O を行う「ダイレクトファイルアクセス方式」を提案し，I/O デリゲーション方式の課題であるプロセッサ間データ転送処理オーバヘッドの削減を目指す．本提案方式は，3.2 節で述べたように，既存システム構成ではアドレス空間の制約により達成できないため，次章で述べる Multiple PVAS プログラム実行基盤において本提案方式を設計し実現する．

5. システムモデル

5.1 Multiple PVAS（M-PVAS）

M-PVAS は，マルチコアプロセッサとメニーコアプロセッサ上の個々のタスクが同一の仮想アドレス空間上で動作

するプログラム実行基盤である．図 2 にシステム構成，図 3 に M-PVAS で管理する仮想アドレス空間の概念図を示す．図 2 に示すように，M-PVAS では各 CPU 上で OS が資源を管理し，OS 間でメモリ管理情報を交換しあうことで，“M-PVAS Address Space”という単一の大域的な仮想アドレス空間を形成する．M-PVAS における各 OS は，PVAS（Partitioned Virtual Address Space）のアドレス空間を管理する．PVAS においてコアを割り当てる実行実体であるプロセスを“PVAS Task”と呼び，一つの仮想アドレス空間“PVAS Address Space”上に，各 PVAS Task のアドレス空間“PVAS Partition”を配置する．これにより，PVAS Task 間では，既存の汎用 OS における共有メモリを要すること無く，仮想アドレス参照により Task 間でデータを共有することができる．M-PVAS では，プロセッサ間でこの大域的な仮想アドレス空間をアクセスできる．これにより，異なる PVAS 空間に属する PVAS Task 同士が仮想アドレスを用いて直接メモリアクセスができるので，異なるプロセッサ間でメモリベースのデータ転送や制御を行える．

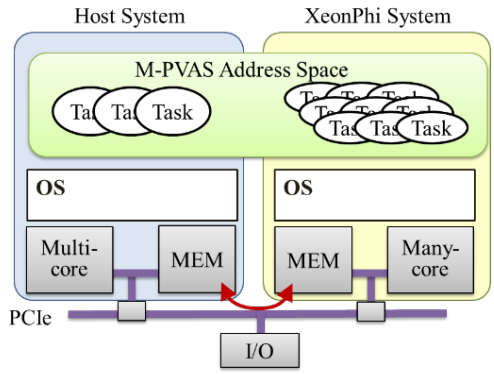

図 2　M-PVAS システム構成

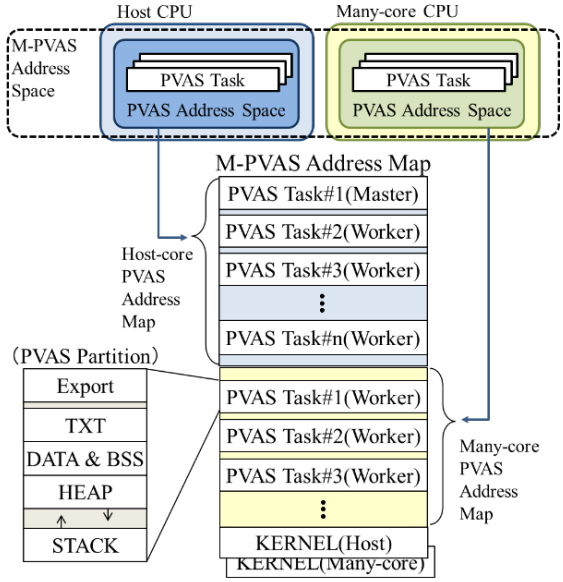

図 3　M-PVAS の仮想アドレス空間の概念図

5.2 XeonPhi 向け MapReduce フレームワーク

図 4 に XeonPhi 側で MapReduce 処理を行い，ホスト側でファイルアクセス処理を行う MapReduce フレームワークのタスクモデルを示す．XeonPhi 側では一つのタスクが Master となり，MapReduce 処理対象データの管理および Worker 制御を担う「Master/Worker モデル」を構成する．MapReduce の処理対象データを Master が管理し，Worker へ本データを分配し，担当領域の先頭アドレスとサイズを Worker へ通知することで，フレームワーク内部での MapReduce 対象データのメモリコピーを抑制し，各 Worker で効率よく MapReduce 処理を実行できる設計としている．なお，既存の MRPhi[7]のフレームワークではスレッドモデルで実装することで Master/Worker 間でデータ共有し，本研究の M-PVAS でも，Master タスクと Worker タスクを同一の M-PVAS 空間上で実行させることでデータ共有可能である．さらに M-PVAS では Master をホスト上で稼動させて，XeonPhi 側の Worker をホスト側の Master から制御するモデルも構成できる[9]ため，MapReduce 処理効率のよい方で Master を実行させることが可能である．

本タスクモデに対して本研究では新たに「File I/O タスク」を追加する．File I/O タスクは，Master からの指示によりホスト側でファイルを読み出し，XeonPhi 側へデータを転送する役割を担う．3.1 節で述べたように，XeonPhi 側ではファイルアクセス効率が悪いため，本研究では File I/O タスクをホスト側で稼動させる．

本タスクモデルにおいて，高効率な MapReduce 処理を実現するためには，XeonPhi 側の MapReduce 処理をできる限り待たせることなくホスト側から XeonPhi 側へ順次データを転送することが性能的に重要な課題となる．次章では，M-PVAS の実行基盤において高効率な MapReduce 処理を実現するためのファイル I/O 処理の設計について述べる．

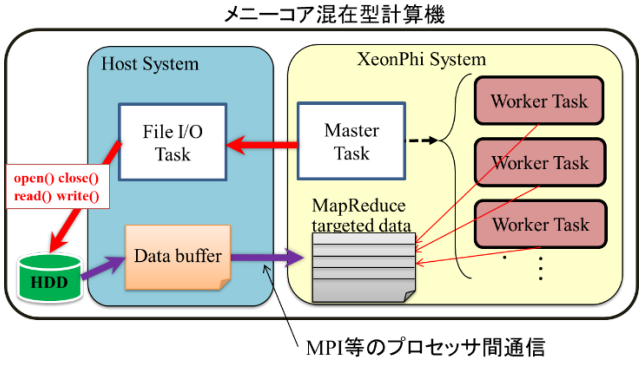

図 4　XeonPhi 向け MapReduce フレームワークモデル

6. M-PVAS MapReduce フレームワークにおけるファイル I/O 処理

ホスト側から効率よくデータを転送して XeonPhi 側の MapReduce 処理をできる限り待機させずに効率良く実行させるための設計として，以下の二項目が検討課題となる．

- ホストから XeonPhi へのデータ転送方式
- ホストから XeonPhi へのデータ転送処理と MapReduce 処理とのオーバラップ制御方式

以下，それぞれの項目に関する設計について述べる．

6.1 ホストから XeonPhi へのデータ転送方式

5.1 節で述べたように，M-PVAS では，XeonPhi 側タスクの仮想アドレスに対してホスト側の別タスクがアクセス可能なプログラム実行基盤である．そのため，3.2 節で述べた従来 OS におけるファイル I/O デリゲーション方式の他に，直接ホスト側から XeonPhi 上の I/O バッファに対して File I/O を実施する方式を提案することができる．これら二つの転送方式について述べる．

(a) ダイレクトファイルアクセス方式

Worker の MapReduce 処理時に参照する I/O バッファの仮想アドレスおよび I/O バッファサイズを Master から File I/O タスクへ通知し，ホスト上の File I/O タスクが直接 XeonPhi 上の I/O バッファへデータを書きこむ方式である（図 5）．本方式では，次項目の方式(b)で説明するホストデータコピー方式に比べて，メモリコピー回数が一回少ない．そのため，データ転送性能の向上が期待できる方式である．

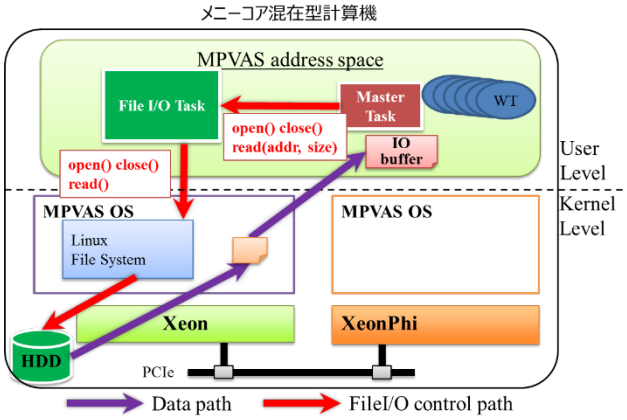

図 5　ダイレクトファイルアクセス方式

(b) ホストデータコピー方式

3.2 節で説明した従来 OS で実現されるファイルアクセスを想定した方式である．方式(a)と同様に，Worker の MapReduce 処理時に参照する I/O バッファの仮想アドレスおよび I/O バッファサイズを Master から File I/O タスクへ通知し，それを受けたホスト上の File I/O タスクがファイルからデータを読み，メモリに一旦データを保持し，そのデータを XeonPhi へ転送する方式である（図 6）．File I/O タスクから XeonPhi へのデータ転送方式に関しては，従来 OS では MPI 通信などのプロセッサ間通信を利用する必要がある．M-PVAS の場合，仮想アドレスを指定して標準ライブラリ関数「memcpy()」を用いたデータ転送が行えるため，MPI 等のプロセッサ間通信に要する制御オーバヘッドを抑制できる可能性がある．

ファイル I/O に対するアクセス性能と，PCIe バスを介したプロセッサ間のデータ転送に関しては方式(a)と方式(b)ともに同様の時間がかかると想定できることから，両方式の違いは，ホストから XeonPhi へデータ転送するまでに File I/O タスク内のバッファを経由する点となる．そして，本メモリコピーのオーバヘッドがデータ転送性能に影響すると予想される．したがって，定性的に方式(a)が有効だと考える．

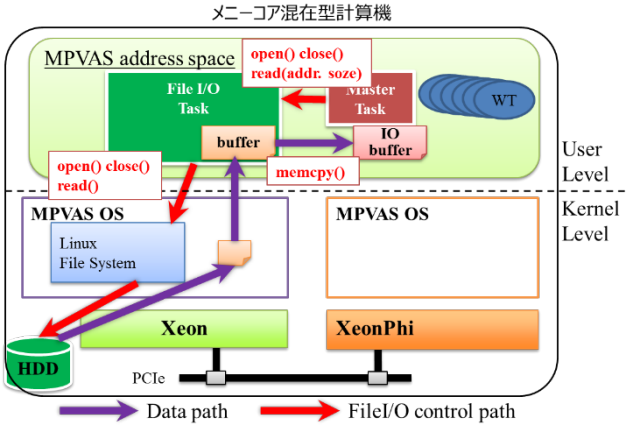

図 6　ホストデータコピー方式

6.2 MapReduce 処理とのオーバラップ制御方式

MapReduce 処理では，MapReduce アプリケーションプログラマが定めた処理対象データをある処理単位で分割し，複数の Worker で Map 処理を並列処理する．本処理対象データには特に依存関係はないため，本処理対象データを，Worker の MapReduce 処理と平行して順次先読みしておくことが可能である．そこで本研究では，MapReduce 処理とデータの先読みをオーバラップさせることにより，Worker タスクのデータ待機時間をできる限り減らし，MapReduce 処理効率の向上を図る．

本オーバラップ制御については，一つの File I/O タスクから二面の I/O バッファを介してデータを転送する方式を基本とする．さらに，I/O バッファ数や File I/O タスク数を増やすことによりオーバラップ制御効率が向上できると考える．以下，本研究で検討した File I/O タスクオーバラップ制御方式について述べる．

(I) 基本構成の FileI/O 処理方式

二面の I/O バッファに対して，片面を Worker による MapReduce 処理に使用し，もう片面を File I/O タスクからのデータ転送に使用する（図 7）．Master は，Worker による I/O バッファに対する MapReduce 処理の終わりを監視しつつ，File I/O タスクへ次のデータに対する転送を依頼する．Worker タスクの MapRedcue 処理の方が File I/O タスクのデータ転送処理よりも速く完了した場合にはデータ到着までの待ち時間が発生する．

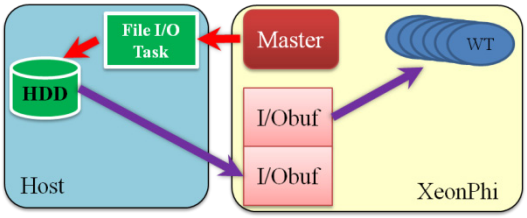

図 7　基本構成による FileI/O 処理方式

(II)　I/O バッファを分割して File I/O を依頼する方式

方式(I)を補う方式として，一つの I/O バッファに対するファイル I/O 処理を複数の File I/O タスクで分担する方式である（図 8）．I/O バッファに対して複数のタスクで並列に処理することで，方式(I)よりも I/O バッファへのデータ転送までの時間を短縮できる可能性がある．実際には，HDD デバイスやプロセッサ間をつなぐ PCIe バスに対する同時アクセス時にボトルネックが生じる可能性がある．しかし，その他の File I/O 処理の制御部分を並列化したことによる高速化を期待できる方式である．

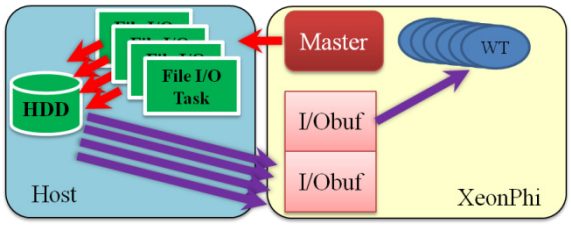

図 8　I/O バッファを分割して File I/O を依頼する方式

(III)　I/O バッファごとに FileI/O を依頼する方式

I/O バッファを複数用意し，I/O バッファ単位で順次 File I/O タスクへデータの先読みを依頼する方式である（図 9）．方式(II)に比べ，XeonPhi 側の I/O バッファのために消費するメモリ容量が増えるが，先読みのサイズを増やすことで，Worker のデータ待機時間を抑制できる可能性が増えると考える．本方式も方式(II)と同様に，HDD デバイスやプロセッサ間をつなぐ PCIe バスに対する同時アクセス時にボトルネックが生じる可能性がある．

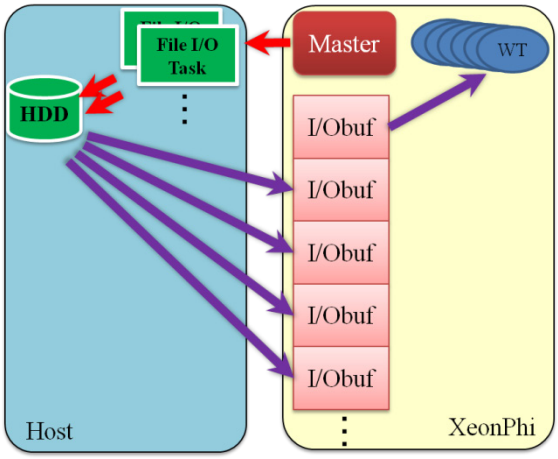

図 9　I/O バッファごとに File I/O を依頼する方式

これらの方式は Master による File I/O タスクに対するリクエスト制御方式であり，6.1 節で述べたダイレクトファイルアクセス方式(a)やホストデータコピー方式(b)によるデータ転送方式と組み合わせて構成できる．次章の評価では，これらの組み合わせでオーバラップ制御方式の効果を検証する．

7.　評価

6 章で述べたファイル I/O 処理方式を備えた MapReduce フレームワークを M-PVAS 上に実装し，MapReduce アプリケーション実行時の MapReduce 処理時間を比較した．

本評価で用いた計算機環境の仕様を表 2 に示す．また，本評価では，文献[7]のベンチマークプログラムの中から，XeonPhi 向け最適化[7]による効果の高い Monte Carlo ベンチマークプログラムを用いた．本ベンチマークプログラムに使うファイルサイズを 4GByte とし，I/O バッファサイズ，すなわち MapReduce 処理単位を 256MByte とした．本評価環境では，XeonPhi のデバイスメモリ上に 4GB の対象データを配置した場合の MapReduce 処理時間に約 1 秒，M-PVAS におけるホストから XeonPhi に対する 256MByte 単位の memcpy()性能が約 3.50GB/s であり，プロセッサ間のデータ転送よりも MapReduce 処理時間のほうが速い環境であった．そこで，比較的大きな単位でファイルの先読みをするために上記サイズで評価を行うことにした．また，ファイル I/O に関しては，OS のファイルキャッシュにヒットする場合での評価とし，I/O 性能を含まないファイルアクセスの制御オーバヘッドで評価した．

なお，5.2 節で述べたように M-PVAS では Master や Worker をどちらのプロセッサで実行するかということを選択することができる．本評価では，既存研究[7]と同じ構成，すなわち Master/Worker を共に XeonPhi 上で実行させて評価することにした．Worker 数については，XeonPhi 上で Monte Carlo ベンチマークの予備評価をしたときに最短の処理時間を示した時の Worker 構成である 236 個とした．

表 2　評価に使用したメニーコア混在型計算機仕様

Many-core	CPU	Intel Xeon Phi 5110P (60 cores, 240 threads, 1.053GHz)
	Memory	GDDR5 8GB
	OS	Linux 2.6.38
Multi-core	CPU	Intel Xeon E5-2650 x2 (8 cores, 16 threads, 2.6GHz)
	Memory	DDR3 64GB
	HDD	WD1000CHTZ (Data transfer rate (max) : 200MB/s)
	OS	Linux 2.6.32 (CentOS 6.3)
Intel CCL	MPSS	Version 3.4.3

7.1 評価結果

File I/O を含む MapReduce 実行時間の比較結果を図 10 に示す．図 10 において，method(I)～method(III)は 6.2 節で述べた方式(I)～(III)に相当し，method(I)では File I/O タスクを一つ稼働させ，method(II)と(III)では File I/O タスクを二つ稼働させた結果を示している．また，method(a)，(b)は 6.1 節で述べた方式(a)，(b)に相当するデータ転送方式である．なお，「method(b)MPI」とは，既存の MRPhi[7]などに File I/O 処理を追加したことを想定した方式である．すなわち，従来 OS ではプロセッサ間では MPI 等の通信方式を用いる必要があるため，M-PVAS での方式(b)における「memcpy()」の部分を別途測定したプロセッサ間の MPI 転送性能に置き換えて机上算出した実行時間を示している．

本評価では，Worker の MapReduce 処理単位でのデータ転送と MapReduce 処理をオーバラップさせながら実行し，Worker による MapReduce の処理時間（MapReduce processing time）と，次の MapReduce 処理のためのデータが I/O バッファに届くのを待つ時間（data waiting time）を Master 内で各々計測したものを合計している．その結果，データ転送方式によらず MapReduce 処理時間には同じとなり，どのオーバラップ制御方式を選択した場合でも本研究で提案するダイレクトファイルアクセス方式(a)が全体の実行時間の削減に寄与した．その中でも，I/O バッファの処理を複数の File I/O タスクで分担する方式(II)との組み合わせが最も高い効果を示した．

また，予備評価により，M-PVAS のプロセッサ間での memcpy 性能は約 3.50GB/s，MPI 通信の性能は約 3.88GB/s と MPI の方が 1 割ほど高い通信性能であったことから，方式(b)＋MPI 方式が方式(b)のホストデータコピー方式よりも高い性能を示した．方式(b)＋MPI 方式を従来方式であると想定した場合，提案する方式(a)により，最大で約 21%の実行時間削減が可能であるという結果が得られた．

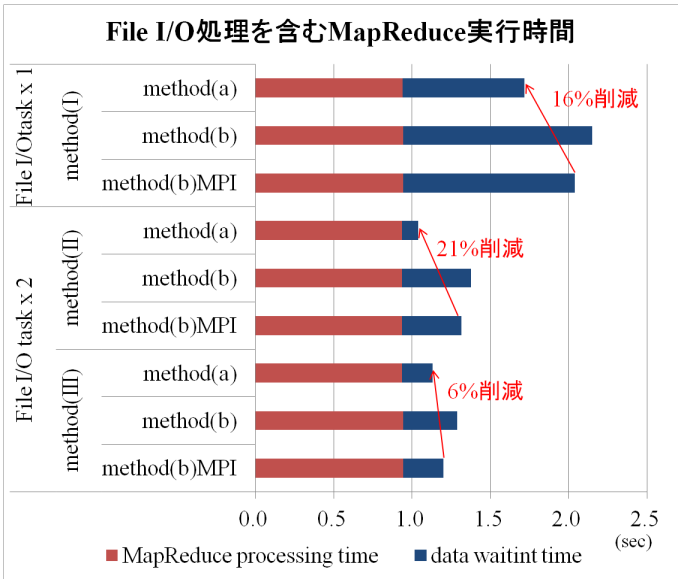

図 10　方式比較の結果

7.2 考察

MapReduce フレームワークでは，MapReduce 全体の実行時間の中で MapReduce 処理の実行を阻害する要因を見極め，待ち時間をいかに減らすかが効率向上につながる．本研究の場合，ホスト側のファイルのデータを XeonPhi へ送るファイル I/O 処理の遅れが待ち時間となっていると考え，各種方式を提案した．その結果を確認するために，File I/O タスク側で計測した，Master からのファイル I/O リクエストに対する処理時間の内訳を調べた．図 11 に示すように，方式(a)では，ホストから XeonPhi へ直接「read()」を実行しているため，この「read()」処理時間の合計値を示している（Direct file I/O）．方式(b)では，ホストから内部のバッファへの「read()」時間（Host file read）と，内部バッファから XeonPhi 側の I/O バッファへの「memcpy()」時間（Memcpy）の合計値をそれぞれ示している．方式(b)＋MPI 方式は，方式(b)の「read()」時間（Host file read）と机上算出した MPI 通信時間を示している．これらの結果から，データ転送処理時間が図 10 で示した MapReduce 実行時間とほぼ同等の時間を占めており，File I/O タスクにおけるデータ転送処理時間を削減することが MapReduce 実行時間全体の削減につながったことが明らかとなった．

最もデータ転送処理時間を削減できた方式(II)のオーバラップ制御においては，内部バッファを介さずにホスト側で直接ファイル I/O を行う方式(a)により，方式(b)の約 25%のデータ転送処理時間を削減できた．本データ転送処理時間は MPI を用いても約 7%しか削減できないことから，本ダイレクトファイルアクセス方式によりバッファコピーを減らした効果が高かったといえる．

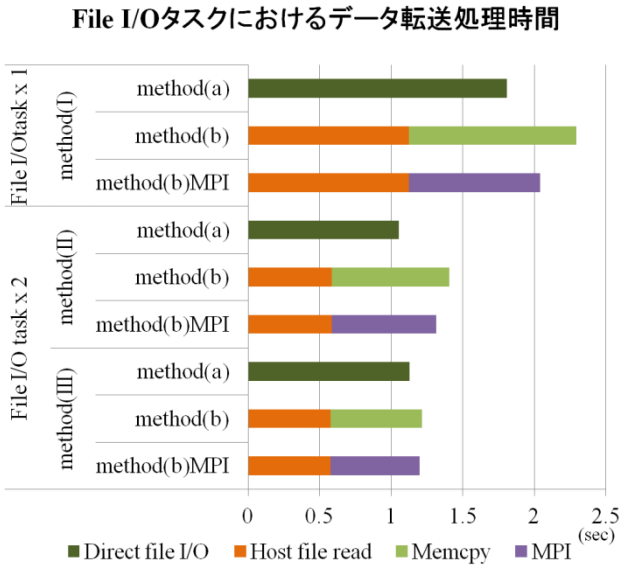

図 11　データ転送処理時間の内訳

最も効率のよいファイル I/O 処理方式であった，方式(II)と方式(a)の組み合わせでも，まだ MapReduce 処理の待ち時間が若干発生している．本評価では File I/O タスクを 2 個

までで評価しているが，この数を増やして File I/O 処理をさらに分割することで，定性的には効果が上がると考えている．なお，本研究で提案した方式は，ホスト上の大量データを XeonPhi 側で処理するフレームワークに有効であり，ホスト上のファイルから XeonPhi へデータを転送する各種システムへ応用可能な方式である．

8. おわりに

本研究では，Intel XeonPhi を搭載するヘテロジニアスアーキテクチャシステムを対象とした MapReduce フレームワークにおいて，XeonPhi 側の MapReduce 処理からのファイル I/O 要求をホスト側で代行処理する機能を設け，MapReduce 対象データをファイルから効率よく取得する方式を提案した．特に，ホスト上の File I/O タスクが直接 XeonPhi 上の I/O バッファへデータを書きこむ「ダイレクトファイルアクセス方式」，および，一つの I/O バッファを複数のタスクで分担して並列にファイル I/O を代行する「I/O バッファを分割して File I/O を依頼する方式」を組み合わせた効率のよいファイル I/O 処理により，従来研究[7]のモデルを想定した MPI 通信を用いて実現する方式よりも約 20%もの MapReduce 実行時間を削減できる見通しを得た．

今後は，さらに本データ転送処理時間を削減するために，File I/O タスクをさらに増やすなど，I/O バッファの先読みを工夫することでさらなる効果が得られるかどうかを確認する必要があると考えている．また，今回の評価では，従来研究のモデルとの比較を机上評価で行ったため，より正しい比較結果を示すために，MPI 通信を用いた従来モデルの実装・評価を続ける．

謝辞 本研究は，科学技術振興機構(JST) の戦略的創造研究推進事業「CREST」における研究領域「ポストペタスケール高性能計算に資するシステムソフトウェア技術の創出」によるものである．

参考文献

[1] J. Dean and S. Ghemawat: Mapreduce: Simplified data processing on large clusters, in OSDI, 2004.
[2] Welcome to Apache Hadoop (online), available from http://hadoop.apache.org.
[3] M. Matsuda, N. Maruyama, and S. Takizawa: K MapReduce: A scalable tool for data-processing and search/ensemble applications on large-scale supercomputers, in CLUSTER , IEEE Computer Society, pp. 1-8, 2013.
[4] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wnag: Mars: a mapreduce framework on graphics processors, in PACT, pp. 1-8, 2008.
[5] J. A. Stuart and J. D. Owens: Multi-gpu mapreduce on gpu clusters, in IPDPS, pp. 1068-1079, 2011.
[6] J. Talbot, R. M. Yoo, and C. Kozyrakis: Phoenix++: modular mapreduce for shared-memory systems, in Proc. of the second international workshop on MapReduce and its applications, pp.9-16, 2011.
[7] M. Lu, L. Zhang, H. P. Huynh, Z. Ong, Y. Liang, B. He, R.S.M. Goh, and R. Huynh: Optimizing the MapReduce framework on Intel Xeon Phi coprocessor, IEEE International Conference on Big Data, pp.125-130, 2013.
[8] Lu, M., Liang, Y., Huynh, H., Liang, O., He, B., and Goh, R.: MrPhi: An Optimized MapReduce Framework on Intel Xeon Phi Coprocessors, IEEE Transactions on Parallel and Distributed Systems, vol.PP, no.99, pp.1-14, 2014.
[9] 佐藤未来子，島田明男，吉永一美，辻田祐一，堀敦史，並木美太郎: メニーコアプロセッサにおける PVAS タスクモデルによる MapReduce アプリケーションの性能評価, 情報処理学会「ハイパフォーマンスコンピューティング研究会」第 150 回研究報告, Vol.2015-HPC-150, No.17, pp.1-5, 2015.
[10] M. Sato, G. Fukazawa, A. Shimada, A. Hori, Y. Ishikawa, and M. Namiki: Design of Multiple PVAS on InfiniBand Cluster System Consisting of Many-core and Multi-core, in EuroMPI/ASIA '14, pp.133-138, 2014.
[11] Intel Developer Zone: Compiling, Configuring and running Lustre on Intel® Xeon Phi™ Coprocessor, https://software.intel.com/en-us/blogs/2014/11/06/lustre-on-intel-xeon-phi.
[12] 深沢 豪, 長嶺精彦, 坂本龍一, 佐藤未来子, 吉永一美, 辻田祐一, 堀 敦史, 石川 裕, 並木美太郎: マルチコア・メニーコア混在型計算機における軽量 OS 向け I/O ライブラリの提案, 情報処理学会「システムソフトウェアとオペレーティング・システム研究会」第 122 回研究報告，Vol. 2012-OS-122, No. 6, pp. 1-8, 2012.