Vol.2018-HPC-163 No.11
2018/3/1

「京」の集団型MPI-IOにおけるプロセス配置を考慮した高速化実装

辻田 祐一[1,a)]　堀 敦史[1]　宇野 篤也[1]　石川 裕[1]

概要：
MPI における並列 I/O インタフェースである MPI-IO の実装では，集団型 MPI-IO において Two-Phase I/O と呼ばれる高速化実装が広く利用されているが，計算ノードに複数のプロセスを配置する場合に，Two-Phase I/O は必ずしも最適化が十分ではない．本稿では，各計算ノードに複数のプロセスが配置される場合においても，プロセス配置や使用するノード形状，さらには配下の並列ファイルシステムの構成や特性等も考慮した Two-Phase I/O の改変実装である EARTH を提案する．本実装を用いることにより，プロセス配置に関係無く，使用する環境に最適な集団型 MPI-IO が可能になる．実装機能の検証の一環として，スーパーコンピュータ「京」において，EARTH における集団型 MPI-IO の読み込み機能の高速化実装を行い，1,536 ノード上に起動した 6,144 プロセスによる HPIO ベンチマークを用いた性能評価において，「京」のオリジナルの MPI-IO 実装に比べて最大で約 15%の性能向上を確認した．

キーワード：MPI-IO，Two-Phase I/O，アグリゲータ，Lustre，スーパーコンピュータ「京」，FEFS

1. はじめに

近年のスーパーコンピュータでは並列ファイルシステムにより，大規模データの高速なファイルアクセスを実現している．また，並列計算で標準的に用いられている通信インタフェースである MPI(Message Passing Interface) [1] においては，このような状況に鑑み，早くから並列 I/O を含む入出力インタフェースである MPI-IO [2] が策定されている．代表的な MPI-IO 実装である ROMIO [3] では，様々な並列ファイルシステムに向けた高速化実装が実現されてきた．特に Lustre [4] は，近年のスーパーコンピュータや PC クラスタシステムで広く利用されており，Lustre 向けに最適化された MPI-IO 実装が利用できる．

MPI-IO は HDF5 [5] や PnetCDF [6] のようなアプリケーション向けの I/O ライブラリにおける並列 I/O 機能としても利用されており，性能向上において集団型 MPI-IO 機能の高速化が重要である．ROMIO においては，集団型 MPI-IO 向けの高速化実装である Two-Phase I/O（以下，TP-IO） [7] が利用されている．上記のようなアプリケーション向け I/O ライブラリなどで行われる不連続なファイルアクセスにおいて，個々のプロセスが対象領域を飛び飛びにアクセスするとファイル I/O 時間が長期化する問題がある．一方，TP-IO を利用した場合，プロセス間通信によるデータ交換とファイル I/O を交互に行うことで，ファイル I/O において連続なファイルアクセスを可能にし，通信による性能低下に対してファイル I/O での性能向上が大きく上回ることで，大幅な性能向上を実現している．

ROMIO では，プロセスの一部あるいは全部にファイルアクセスを行う処理を割り当てており，これらのプロセスをアグリゲータと呼んでいる．各計算ノードに複数のプロセスを配置するケースで，アグリゲータも複数割り当てる場合，ノード毎に詰めたレイアウトになるために，使用するノード群の形状やファイルシステムとの接続状況によっては，TP-IO による高速化が十分に発揮できない場合がある．他にも，使用するファイルシステムへの I/O 要求が一度に大量に発行されると，ファイルシステム側の処理が混雑し，その結果，性能低下を招く場合もある．さらに TP-IO におけるデータ通信がファイルアクセスと交互に行われるが，大量のプロセス群が相互のデータ通信を一度に発行することにより，通信の混雑を招き，性能低下に繋がる可能性もある．以上のような問題は実行規模の増加に伴い，より顕著になるため，大規模化における大きな問題である．

以上のような問題を解決するために，我々はプロセス配置や使用するノード群の形状，さらにはファイル

[1] 国立研究開発法人理化学研究所　計算科学研究機構
[a)] yuichi.tsujita@riken.jp

システムの構成や特性などを考慮した最適化実装である EARTH(**E**ffective **A**ggregation **R**ounds with **TH**rottling) を提案している [8–11]. EARTH では，ユーザが指定したプロセス配置とは関係なく，TP-IO におけるファイルアクセスに最適なアグリゲータ配置やファイルアクセスにおける I/O throttling 機能，さらには通信混雑を緩和する段階的通信機能などを実装している．以上のような最適化により，大規模なプロセス数やノード数による集団型 MPI-IO において，EARTH の優位性が発揮される．これまで集団型書き込みでの高速化を行ってきたが，今回，集団型読出しにおいても同様の最適化実装をスーパーコンピュータ「京」(以下，「京」) の上で実現し，1,536 ノード上に起動した 6,144 プロセスによる HPIO ベンチマークにより，「京」のオリジナル実装に比べて最大で約 15%の性能向上を確認した．

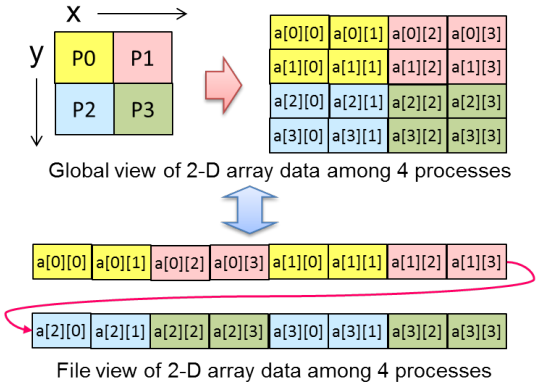
以下，第 2 章において，本実装の開発背景にある MPI-IO の機能と現行の TP-IO 実装について説明した後に，EARTH で提案している最適化実装について第 3 章で説明する．次に第 4 章において今回実装を行った「京」のファイルシステムを含むシステム構成と「京」向けに行った実装について説明する．第 5 章において今回実施した性能評価試験について報告する．第 6 章では関連研究について本提案手法との比較検討を行い，最後に第 7 章において本稿のまとめと EARTH の今後の開発の方向性について述べる．

2. MPI-IO

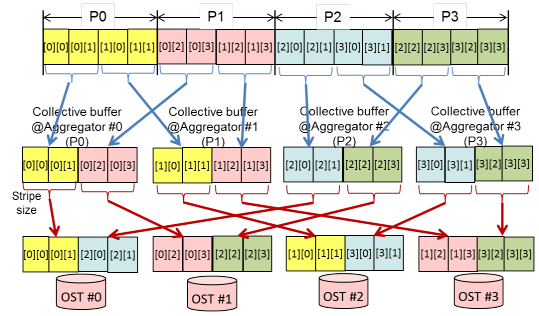
MPI における入出力インタフェースとして策定された MPI-IO には，大きく分けて，プロセス個々に独立に I/O を行う非集団型と，同じファイルに対してプロセス全体で分散してアクセスを行う集団型の 2 種類のインタフェースが用意されている．MPI-IO は，それ自身を直接利用するだけでなく，既に述べたように，HDF5 や PnetCDF のようなアプリケーション向け I/O ライブラリ実装における並列 I/O 機能としても利用されている．特に，HDF5 や PnetCDF では，集団型 MPI-IO インタフェースが多用されており，MPI-IO 実装側での高速化実装の良し悪しが性能に大きな影響を及ぼす．

代表的な MPI-IO 実装である ROMIO においては，様々なファイルシステムに対応する実装を実現するために，ADIO [12] と呼ばれるファイルシステム依存部分と上位の MPI-IO インタフェースレイヤとを仲介するインタフェース実装を提供している．近年のスーパーコンピュータで広く利用されている Lustre ファイルシステム [4] も ADIO でサポートされており [13]，Lustre の特性を活かした高速化実装が提供されている．

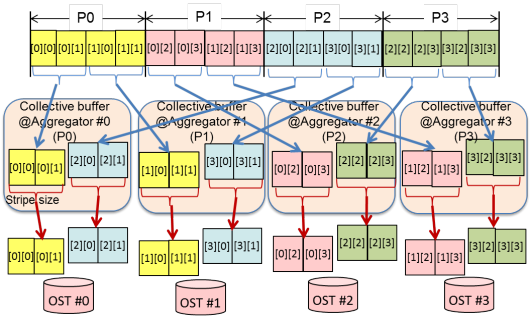
集団型 MPI-IO においては，**図 1** に示すような TP-IO による高速化が行われる．図 1(a) では，2 次元配置のデータを 4 プロセス間でブロック分割した結果，書き込み処



(a) 2 次元データを 4 プロセス間で分割した例



(b) TP-IO による処理フロー（ファイルビュー指向）



(c) TP-IO による処理フロー（ストライピングアクセス指向）

図 1: 2 次元データに対し，4 プロセス間で集団型 MPI-IO を行う例

理でのレイアウト（ファイルビュー）では，各プロセス (P0~P3) でのアクセス領域が不連続になる様子を示している．不連続なアクセスを繰り返し行うと，ファイル I/O 性能の著しい性能低下に繋がる．そこで，TP-IO によって，データ通信により連続するファイルアクセス領域をできる限り大きくすることにより，通信処理による性能低下を上回るファイルアクセスの性能向上により，全体として大きな性能向上が達成される．TP-IO には大きく分けて図 1(b) に示すファイルビュー指向の実装と図 1(c) に示すストライピングアクセス指向の実装がある．前者は一般的なファイルシステム向けの実装で，後者は現在の Lustre 向け ADIO で採用されている手法である．Lustre 向け ADIO では，以前は前者を利用していたが，その後，集団型書き込みにおいて後者の実装に変更されて性能が改善された経緯があ

る [13]．一方で集団型読出しでは，書き込みとは逆の処理フローになる前者の手法を継続して利用している．

図 1(b) に示すファイルビュー指向の実装では，ファイルビューの上でファイルアクセスを担当するプロセス（アグリゲータ）間で均等にアクセス担当領域を分割する．この手法を Lustre に対して適用した場合，各アグリゲータが複数の OST（Object Storage Target）へアクセスするため，ロック競合や OST までのアクセス経路の混雑などを招きやすく，十分な性能向上が望めない．

一方，図 1(c) に示すストライピングアクセス指向の実装では，ファイルビューとは無関係にストライピングアクセスパターンに配慮して，各アグリゲータが特定の OST にアクセスするように，ストライピング情報を元にデータを並び替えてからアクセスを行う．この手法によって，前者に比べてロック競合やアクセス経路の混雑等を軽減でき，大幅な性能向上が達成できる．この手法は集団型読出しに対しても同様の性能向上効果が期待される．

3. 集団型 MPI-IO の高速化

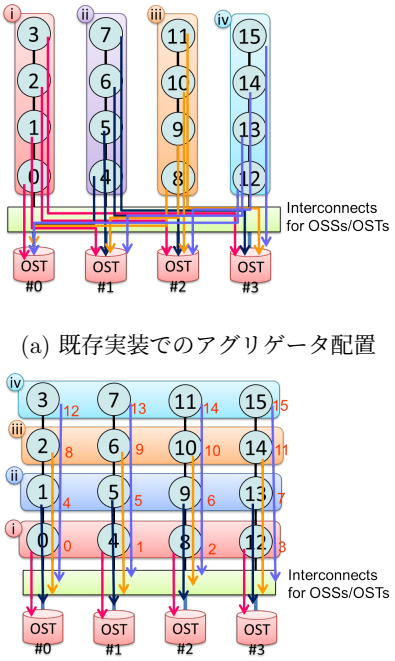
集団型 MPI-IO では，第 2 章で述べた TP-IO により高速化を実現している．しかしながら，計算ノード群と並列ファイルシステム間の相互結合状況や計算ノード内・計算ノード間のプロセス配置等を考慮した設計になっておらず，例えば計算ノードに複数のプロセスを配置した場合に十分な性能を引き出せない問題などがある．そこで，我々は以下の最適化実装を提供する高速化実装である EARTH の開発を進めている．

- ストライピングアクセスに適した TP-IO 実装
- アグリゲータ配置の最適化
- I/O throttling によるファイルアクセス最適化
- 段階的データ通信による混雑緩和
- I/O throttling と段階的データ通信に適したアグリゲータ配置（ノード内に複数のアグリゲータがある場合）

なお，1 番目の実装は，既に説明した通り，Lustre 向け ADIO において，集団型書き込みで実装済み [13] であり，EARTH においても同様の実装を集団型書き込みで用いているが，今回新たに集団型読出しにおいても実装した．以下，残りの 4 項目について，それぞれの実装の機能説明を行う．

3.1 アグリゲータ配置の最適化

並列ファイルシステムと計算ノード群との相互接続方式によっては，アグリゲータの配置方法が性能に大きな影響を与える可能性がある．例として**図 2** に示すような直接結合網で相互接続された計算ノード群と並列ファイルシステムの環境でのアグリゲータ配置を考える．この図では，4 ノードと 1 台の OST が同じ通信リンク上に配置され，全体で 16 ノードを用いて 16 プロセスにより 4 つの OST にアクセスするケースを示している．図中の丸はプロセスを表し，丸の中の数字は MPI ランクを表している．図 2(a) では，ランク順にアグリゲータが配置されたケースを示している．図中の”i”から”iv”はストライピングアクセスのラウンド順を示しており，この場合，最初のラウンドでランクが 0 から 3 の 4 プロセス，次のラウンドで 4 から 7 のプロセス，というようにファイルアクセスが行われる．このようなアクセスパターンにおいては，個々のラウンドで同じ通信リンク上にデータ通信が重なり混雑する一方，アクセスしていないプロセス群の通信リンクはデータ通信が行われず，全体として，通信リンクの使われ方にも偏りがある．



(a) 既存実装でのアグリゲータ配置

(b) EARTH におけるアグリゲータ配置

図 2: アグリゲータ配置に応じたファイルアクセスの流れの違い

一方，図 2(a) では，ランク順とは無関係にプロセスのレイアウトに配慮し，各通信リンク毎に順に下から処理が進むようにアグリゲータが配置されている．なお，図中の丸の右にある赤字の数字がアグリゲータ配置順を示している．この場合，最初のラウンドではランクが 0，4，8 及び 12 のプロセスのアクセスが行われ，次のラウンドでは，1，5，9 および 13 のランクのプロセスからのアクセスが行われるため，各々のラウンドにおいて各通信リンクの混雑が緩和され，かつ全通信リンクが均等に利用される．その結果，前者に比べてファイルアクセス性能が向上する可能性が高くなる．

3.2 I/O throttling

ファイルシステムに対し，一度に大量の I/O 要求を送ると，ファイルサーバ側での処理が追いつかなくなり，その

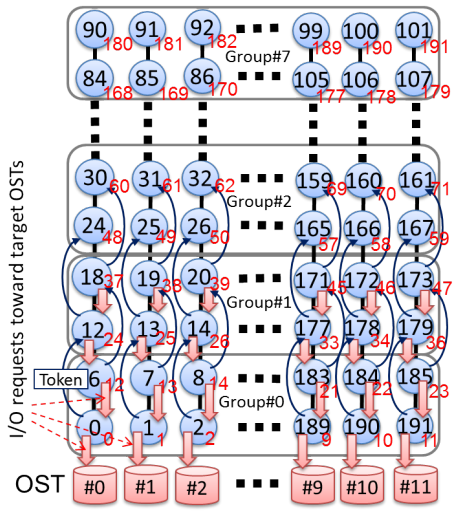
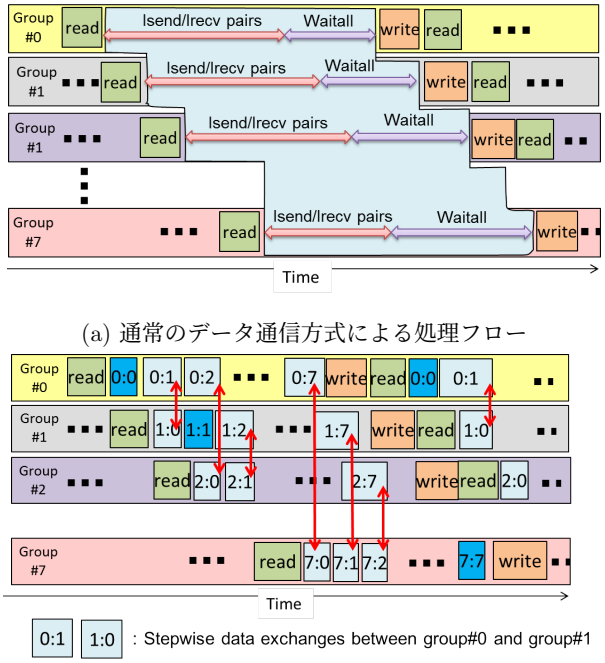


図 3: I/O throttling の実行パターン例. 図中の丸の中の数字および右横の赤字の数字はそれぞれ MPI ランク並びにアグリゲータ配置順を表す.



(a) 通常のデータ通信方式による処理フロー

(b) 段階的データ通信方式による処理フロー

図 4: 通常のデータ通信方式と段階的データ通信方式を適用したケースでの処理フロー

結果，ファイルアクセス性能が低下する可能性がある．このような状況を改善するために，I/O 要求発行を段階的に行う I/O throttling という手法が用いられている [14, 15].

第 3.1 章で述べたアグリゲータ配置の最適化に対し，同様の手法を付加的に適用することで，更なる性能向上が期待できる．**図 3** は，各 OST へのアクセスを 2 リクエストずつに絞って実行している様子を示している．我々の実装では，OST 群へのストライピングアクセスにおける連続する複数ラウンド単位でグループを形成し，グループ間のトークン受け渡しによって，段階的に I/O 要求を発行させるようにしている．

3.3 段階的データ通信

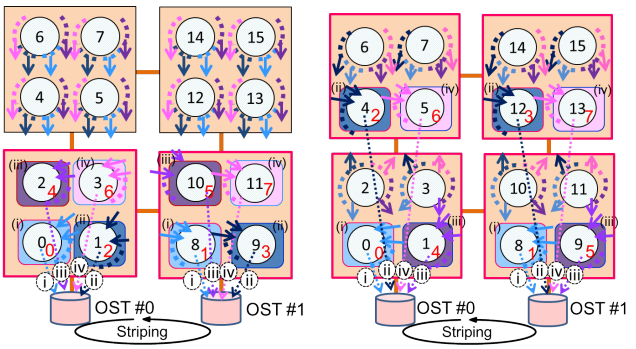
アグリゲータへデータを集める全対全通信に関して，一度に全プロセスとの通信を開始すると，通信混雑が発生する恐れがある．そこで，通信混雑の軽減による性能向上のために，データ通信フェーズに対し，I/O throttling のステップ数に合わせて段階的に通信を行う機能を実装している．

図 4 に 8 グループ間（Group #0 ～ #7）でのデータ通信フェーズを含む TP-IO 内部の処理フローを示す．図 4(a) では，各グループに属するプロセスが read 処理の後に一度に通信関数を発行した場合の処理の流れを示している．一度に通信関数を発行することによる混雑により，通信処理が遅れ，さらに，ステップ順に次々と各プロセスが通信関数を一度に発行するために，混雑がさらに進み，通信性能低下を招く恐れがある．なお，オリジナルの TP-IO では，プロセス配置に関係なく機械的に通信相手ランクを 0 から順に `MPI_Irecv()` および `MPI_Isend()` を発行している．これに対して EARTH では，ファイル I/O での I/O throttling による処理フローに沿ったプロセス配置に配慮したランク順で通信関数を発行することで，ファイル I/O フェーズの終了順に合わせた効率の良い通信を可能にしていたが，この方式でも一度に通信関数を発行していることに変わりはなく，オリジナルの TP-IO と同様に混雑による性能低下を招く可能性があった．

一方，図 4(b) に示す段階的データ通信方式では，ファイル I/O での I/O throttling 処理とプロセス配置に配慮したランク順に従って，各グループに属するプロセスが段階的に通信を開始するために，段階的なファイル I/O 処理の有効性を保ちつつ，通信混雑を緩和することが可能になり，性能向上に寄与できる可能性がある．

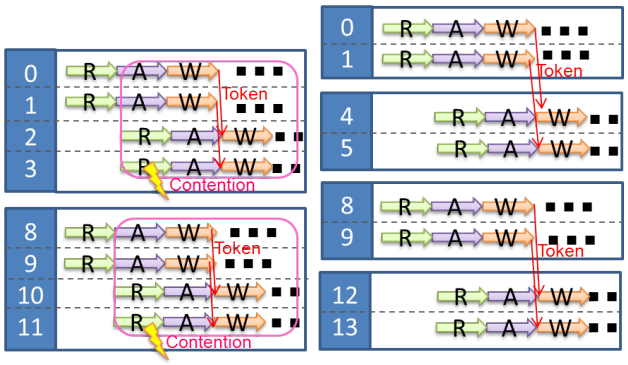
3.4 I/O throttling と段階的データ通信に適したアグリゲータ配置

計算ノードあたりにプロセスが複数配置されている際に，ROMIO では，`MPI_Info_set()` により `cb_config_list` および `romio_lustre_co_ratio` の設定を調整することで，ノード数よりも多くアグリゲータを割り当てることができる．これにより性能向上を狙うことも可能となる．しかしながら，各ノードに順に詰めて配置すると，ノード間のアグリゲータ配置のバランスが崩れるだけでなく，ノード内の複数のアグリゲータ間の処理の干渉が顕著になり，性能低下を招くおそれがある．そこで，ノード間でラウンドロビン配置でアグリゲータを配置することで，上記の 2 つの問題を解消できる．



(a) マルチプロセス配置の考慮なし　(b) マルチプロセス配置の考慮あり

図 5: I/O throttling と段階的データ通信におけるマルチプロセス配置の考慮の有無によるアグリゲータ配置の違い



(a) マルチプロセス配置に配慮無し　(b) マルチプロセス配置に配慮あり

図 6: I/O throttling と段階的データ通信におけるマルチプロセス配置の配慮をしないケースと配慮をしたケースにおける集団型書き込みでの TP-IO 内部の処理の流れと混雑発生

一例として，**図 5** に，ノードあたり 4 プロセスを配置して，全体で 16 プロセスによりファイルアクセスをするケースを示している．図 5(a) では，下の列の 2 ノードにアグリゲータが詰めて配置されている．この場合，OST 群へのアクセスやデータ通信がこの 2 ノードに集中し，混雑することが想定され，その結果，性能低下を招いてしまう．

一方，図 5(b) では，ノード間でラウンドロビンでアグリゲータを配置させている．これにより全ノードが 2 個ずつアグリゲータを均等に有し，さらにアグリゲータから OST へのアクセスやデータ通信で混雑する状況が前者に比べて緩和される．

上記の 2 種類の配置に関して，TP-IO の内部処理の流れを**図 6** に示す．図 6(a) は図 5(a) での処理フローを示しており，この場合，データ通信（図中の”A”）とファイル読み込み（図中の ”R”）やファイル書き込み（図中の ”W”）との干渉が発生する可能性がある．一方，図 6(b) は図 5(b) での処理フローになっており，前者における干渉を回避できることが分かる．なお，この図では集団型書き込みでの TP-IO の処理フローを示しているが，集団型読出しの場合には，この図でのファイル書き込み（図中の”W”）を除いたものとなり，ファイル読み込みとデータ通信の間で同様の問題が前者のレイアウトでは発生するが，後者のレイアウトにすることで干渉を緩和できる可能性がある．

ここで，I/O throttling と段階的通信を併用する際のアグリゲータ配置に関して，ノード毎に詰めて配置する場合とノード間でラウンドロビン配置させる場合とで，各ステップあたりのノード内の TP-IO での異なる処理フェーズの重複度とファイルアクセスを行うノード数について検証を行う．パラメタとして，ノード内に配置されたプロセス数を C_{node} とし，同一 OST にアクセスする総ノード数を N_g とする．さらに I/O throttling でのリクエスト数を n_{req}，TP-IO 内の処理フェーズ数を k_{TP} とした場合に，それぞれのケースでの見積り量を**表 1** に示す．各ステップでリクエスト数 n_{req} 分のアグリゲータが同じステップで TP-IO 内の同じ処理フェーズを行うが，ノード毎に詰めた配置では，ノード内のプロセス全てが同じステップで同じ処理フェーズを行う場合を除き，異なる処理フェーズ間の干渉が発生する．異なる処理フェーズの多重度を，同一ノード内にある処理フェーズ数で表すと，$n_{req} \times k_{TP}$ か C_{node} のいずれか小さい方が 1 ステップ分での 1 ノードが持つ TP-IO の処理に関与するプロセス数になる．これを C_{node} と n_{req} のいずれか小さい方で割った値が 1 ノードが持つ TP-IO 内の異なる処理フェーズの個数となる．この値は k_{TP} を越えない範囲の値となるため，この表に記載のように記述できる．一方，アグリゲータをノード間でラウンドロビン配置させた場合，TP-IO 内の処理に関わるプロセス数は $n_{req} \times k_{TP}$ となり，これを N_g で割った値の切り上げ値か k_{TP} のいずれか小さい方が，ノード内の異なる処理フェーズ数の最大値になる．

また，ファイル I/O に関わるノード数に関しては，詰めて配置させた場合には，n_{req}/C_{node} を切り上げた値か N_g のいずれか小さい方の値になり，ラウンドロビン配置させた場合では $min(n_{req}, N_g)$ となる．通常 C_{node} の方が N_g に比べて小さいため，後者の方が大きくなる．I/O throttling での最適なリクエスト数の条件にもよるが，多重度をできるだけ小さく，かつファイル I/O に関わるノード数を多めに取りやすい観点で見た場合，後者の方が性能向上させやすい構成になると考えられる．

4. EARTH on K の実装

本稿では，EARTH の集団型 MPI-IO の高速化実装の検証のために「京」を用いた．我々は EARTH の枠組みの中で，「京」向けの実装である EARTH on K を開発してお

表 1: 各々のアグリゲータ配置での各ステップでのノード内の TP-IO での異なる処理フェーズの多重度と OST あたりのファイル I/O に関与するノード数

配置方法	ノード内の TP-IO における異なる処理フェーズの多重度	OST あたりのファイル I/O を行うノード数
詰めて配置	$min(min(C_{node}, (n_{req} \times k_{TP}))/min(C_{node}, n_{req}), k_{TP})$	$min(\lceil n_{req}/C_{node} \rceil, N_g)$
ラウンドロビン配置	$min(\lceil (n_{req} \times k_{TP})/N_g \rceil, k_{TP})$	$min(n_{req}, N_g)$

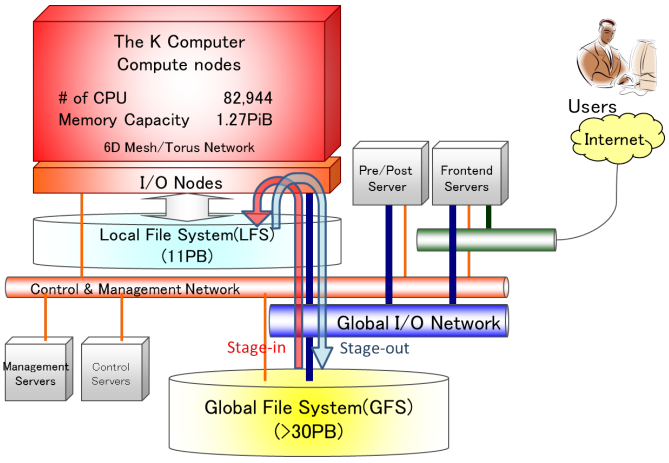


図 7: スーパーコンピュータ「京」のシステム構成

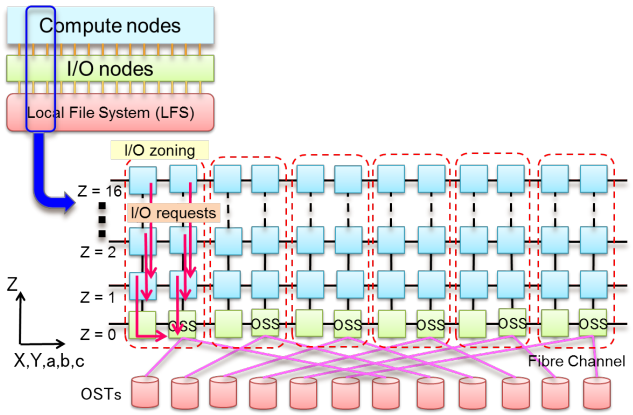


図 8: I/O zoning による計算ノードからの LFS へのファイルアクセス

り，現在の Lustre 向け ADIO に倣い，「京」のファイルシステムの構成とストライピングアクセスに最適化した高速化実装を集団型書き込みにおいて実現している [8, 11]．なお，EARTH on K は開発ベースとなる FEFS 向け ADIO のソースコードが富士通の著作物であるため，バイナリ形式のライブラリのみ「京」の上で利用公開をしている．

EARTH on K の実装の説明の前に，ファイルシステムを含む「京」のシステム構成を**図 7** を用いて説明する．約 8 万 3 千ノードの計算ノードはファイル I/O 等で利用される I/O ノードと共にノード間インターコネクトである Tofu [16] を介して相互に結合されている．「京」は非同期ステージング [17, 18] を用いた二階層のファイルシステムを採用しており，ユーザプログラムやデータ等が通常保管されるグローバルファイルシステム（GFS）と計算ノードからの高速なファイル I/O を実現するローカルファイルシステム（LFS）で構成されている．LFS および GFS 共にファイルシステムとして富士通により Lustre をベースに開発された FEFS(Fujitsu Exabyte File System) が利用されている．Lustre と同様に複数の OST を用いた高速なファイルアクセスを可能にしている．

図 8 に計算ノードから LFS に対してファイルアクセスを行っている様子を示す．IO zoning と呼ばれる機構によって，計算ノードからのファイルアクセス要求は，ノードが繋がっている Tofu の z 軸を共有する I/O ノード（LIO）を経由して対象の OSS（Object Storage Server）に送られる．これにより，計算ノード間の I/O 要求の干渉を軽減している．

「京」の MPI ライブラリは OpenMPI をベースに開発されており，MPI-IO ライブラリは ROMIO をベースに開発されているが，若干古い ROMIO をベースに開発された経緯があり，TP-IO 実装はファイルビュー指向になっている．さらに，現在の Lustre 向け ADIO と同様に，アグリゲータ配置やアグリゲータへのデータ通信方法など，利用環境である「京」や FEFS の特性を活かしきれていない．そこで，EARTH on K では EARTH が提供する最適化機能を提供するにあたり，LFS の OST 群と計算ノード群の間の Tofu による接続状況を「京」のプロセストポロジー関数群等を用いて検出することで，上記の I/O zoning の特性も活かしながら集団型 MPI-IO の高速化を実現している．これまでの集団型 MPI-IO における書き込み実装をベースに，今回新たに読出し機能についても高速化に向けた実装を行った．

5. 性能評価

今回実装した集団型読出し関数に対して以下の評価試験を行った．

- ファイル I/O における I/O throttling 機能の検証
- データ通信フェーズにおける段階的通信の検証
- I/O throttling によるファイル I/O とデータ通信フェーズの混在環境の検証
- HPIO ベンチマークによる性能評価

以下，各評価試験の結果と分析について述べる．

5.1 I/O throttling 機能の検証

まず初めにノードあたり 1 プロセスの配置で 3,072 プロセスを用い，各プロセスがストライプサイズ単位で POSIX

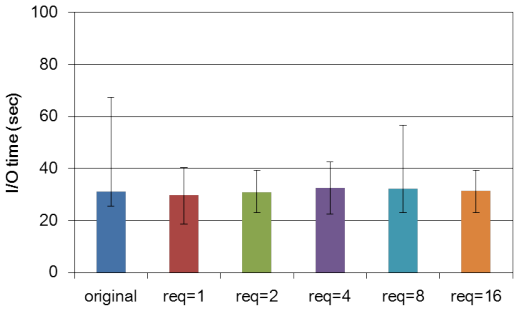


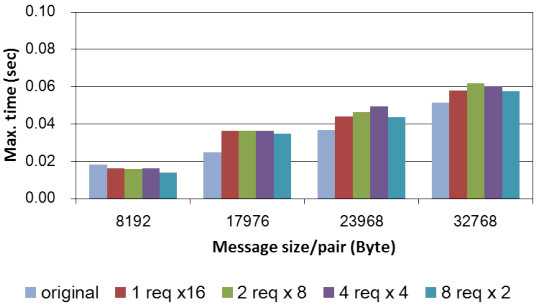
図 9: 3,072 プロセスによる各プロセスからの POSIX read 関数による読み込みに要した時間（ストライプサイズ =16 MiB，1 プロセス/ノード）

read 関数による読み込みを 50 回繰り返し行った際の 1 回あたりに要した時間の平均値，最大値，並びに最小値を**図 9** に示す．この図で最大値と最小値は縦方向のバーで平均値と共に示している．”original”はオリジナル TP-IO の処理を模擬した方式で，全プロセスが一斉にアクセス対象の OST に対してアクセスを行っている．一方，”req=X” は I/O throttling を模擬したファイルアクセスになっており，”X”は各 OST に同時にアクセスしているプロセスの数を表している．

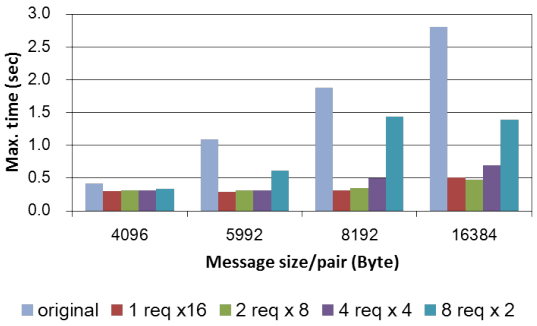
I/O throttling 機構により，同時に同じ OST に到達するリクエスト数を 1 個（req=1）あるいは 2 個（req=2）と絞ることで，I/O throttling を用いなかった”original”のケースと比較して，最大値および最小値でファイル I/O 時間の短縮が確認でき，平均値でも若干の時間短縮が確認できた．

5.2 データ通信フェーズにおける段階的通信機能の検証

アグリゲータにデータを集める通信フェーズでの段階的通信処理の効果を TP-IO で行っているプロセス間の `MPI_Isend()`/`MPI_Irecv()` 対による全対全のデータ通信を模擬したプログラムを用いて検証した．1 ノードあたりに 1 プロセスの配置で，768 プロセスおよび 6,144 プロセスによりメッセージ長を変えて通信に要する時間を計測した結果を**図 10** に示す．シンプルな構成で評価する目的で，全プロセスがアグリゲータとなる構成を想定し，またメッセージ長は毎回同じ設定で評価している．ここで，”original”は段階的通信機能を用いずに全プロセスが同時に全てのプロセスとの通信を行ったケースであり，”X req × Y”と表記しているのは，段階的通信機構を用いたケースで，”X”は I/O throttling でのリクエスト数に相当するもので，”Y” はステップ数である．例えば 3,072 プロセスのケースでは，$8 \times 12 \times 32$ のノード形状で実施しており，OST 数は 192 個となるため，2 リクエストずつの場合には 1 ステップあたり，$2 \times 192 = 384$ 個のプロセスが各々のステップで当該ステップに対象となる 384 個の通信相手のプロセスと通信を行っている．



(a) 768 プロセス（1 プロセス/ノード）



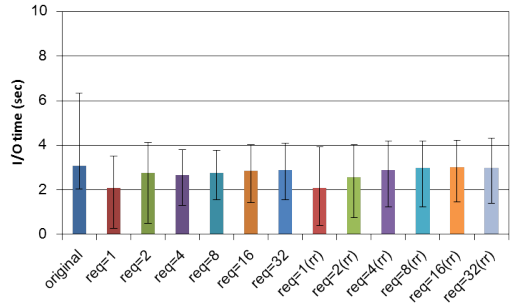
(b) 6,144 プロセス（1 プロセス/ノード）

図 10: `MPI_Isend()`/`MPI_Irecv()` 対による 768 および 6,144 プロセスにおける全対全通信の性能（1 プロセス/ノードの配置）

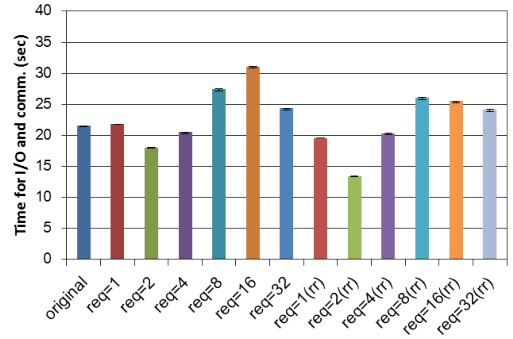
評価結果から，プロセス数およびノード数が少なめの 768 プロセスのケースでは，段階的通信を行う効果は評価したメッセージ長では確認できなかった．一方，プロセス数およびノード数を大きくした 6,144 プロセスのケースでは，段階的通信機構の効果が見られており，一度に通信を発行するプロセス数を絞ることによる通信時間短縮が確認できた．

5.3 I/O throttling によるファイル I/O とデータ通信フェーズの混在環境の検証

TP-IO での処理を模擬して I/O throttling によるファイル I/O と全プロセス間のデータ通信フェーズを交互に行うプログラムを作成し，プロセス個々の配置に配慮したファイル I/O やデータ通信の効果を検証した．ここではノードあたりに 4 プロセスを起動し，ノードあたりに複数のプロセスを起動するケースを想定した評価を実施した．また，ストライピングアクセスに合わせたアグリゲータ配置における順に倣い，I/O throttling を用いた場合のファイル I/O を行う順番については，ノード毎に詰めた配置順とノード間でラウンドロビンに配置した順の 2 パターンを実施した．データ通信に関しては，TP-IO での処理順に従い，書き込み処理ではファイルアクセスの前，読出し処理ではファイルアクセスの後に行っており，`MPI_Isend()`/`MPI_Irecv()` 対の発行の際の通信相手のプロセスは EARTH on K で採



(a) TP-IO を模したストライプサイズ単位のファイル I/O の所要時間



(b) TP-IO を模してストライプサイズ単位のファイル I/O とデータ通信フェーズを混在させた場合の所要時間

図 11: 1,536 ノード上に起動した 6,144 プロセスによるファイルアクセス評価試験の結果

用している方式に倣い，ファイル I/O の発行順と同じにした．なお，通信とファイル I/O を混在させたため，繰り返し回数は 20 回と，第 5.1 章に比べ少なめに設定した．

図 11 に 1,536 ノード（形状：8 × 6 × 32）上に起動した 6,144 プロセスにより実施した計測結果を示す．第 5.2 章での評価に合わせて，全プロセスがアグリゲータとなる構成を想定して評価している．各プロセスでストライプサイズと等しい 16 MiB ずつをプロセス間で重ならない様にオフセットをずらしてファイルから読み込んだ計測と，読み込みデータ通信を交互に混在させた計測について，それぞれで 20 回繰り返し，1 回あたりの平均値，最大値，並びに最小値を求めた．

この図において，いずれも平均値と共に，縦方向のバーで最大値と最小値を示している．”original”はオリジナルの TP-IO 実装を模擬した方式で，ファイル I/O およびデータ通信はプロセス全体で同時に実行している．一方，”req=X”とあるのはファイル I/O で I/O throttling 機構を用いたケースで，”X”は，各々の OST へ同じステップに発行する I/O 要求数であり，EARTH の TP-IO でのストライピングアクセス指向の実装に倣い，ノード毎に詰めたストライピング方向単位で Tofu の z 軸座標の小さいノード上のプロセスから I/O 要求を発行している．また，”req=X(rr)”の”rr”は，TP-IO でのアグリゲータ配置におけるノード間

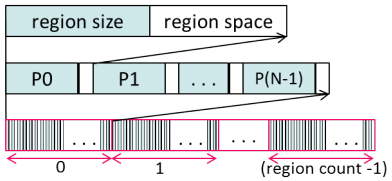


図 12: HPIO ベンチマークによるアクセスパターン生成の例

ラウンドロビン配置に倣い，前者の方式に対して，ノード間をラウンドロビンで I/O 要求を発行する方式を表している．前者と同様に”X”は同一ステップで発行する OST あたりの I/O 要求数である．

図 11(a) では，ストライプサイズ単位でのファイル読み込み時間を示している．これに対し，図 11(b) ではプロセス間のデータ通信を混在させており，この結果からファイル I/O とデータ通信を混在させた場合に，通信時間が占める割合が高いことが分かる．

ファイル I/O のみであった図 11(a) の結果からは，同時発行するリクエスト数を 1 個に絞ったケースが最も時間が短縮されたが，データ通信と混在させた図 11(b) では，リクエスト数が 2 個の時に最も時間が短縮できており，ファイル I/O と通信の組み合わせにより，最適な条件に変更が生じたと考えられる．また，ファイル I/O とデータ通信の干渉の軽減にラウンドロビン配置が有効であることが確認できた．

5.4 HPIO ベンチマークによる性能評価

EARTH on K の性能を HPIO ベンチマーク [19] により評価した．HPIO ベンチマークは，派生データ型を用いた集団型 MPI-IO を含む多様なアクセスパターンをサポートするベンチマークである．**図 12** に示す 3 つのパラメタ（region size，region space および region count）により空白を含んだ不連続なアクセスパターンが利用できる．HPIO ベンチマークの 3 つのパラメタ設定は，region size および region space をそれぞれ 5,992 B および 256 B に，さらに region count を 30,729 とし，ノードあたり 4 プロセスの配置で，1,536 ノードで 6,144 プロセスにより集団型 MPI-IO の読み込み性能を計測した．なお，TP-IO でのアグリゲータはノード間のアグリゲータ配置の検証も兼ねて，半分の 3,072 プロセスとした．

表 2 に示すパラメタ設定の組み合わせを評価した結果を**図 13** に示す．この図では計測性能の平均値と共に縦方向のバーで標準偏差を示している．この結果から，EARTH で提案する実装において，I/O throttlig 並びにアグリゲータ配置をノード間でラウンドロビンにすることによる性能向上が確認できた．さらに，段階的データ通信機構を取り入れることによって，さらに性能が向上することも確認できた．今回の評価では，オリジナルの実装に比べて最大で

表 2: 評価したパラメタ設定一覧（各パターンとも，チェックのある機能を有することを意味する.）

設定パターン	ストライピング指向 TP-IO	アグリゲータ配置		I/O throttling	段階的通信
		ストライピング指向	ノード間ラウンドロビン		
original					
B	✓				
B,agg	✓	✓			
B,agg,rr	✓	✓	✓		
E(req={2,4,8,16,32})	✓	✓		✓	
E,rr(req={2,4,8,16,32})	✓	✓	✓	✓	
E,rr,step(req={2,4,8,16,32})	✓	✓	✓	✓	✓

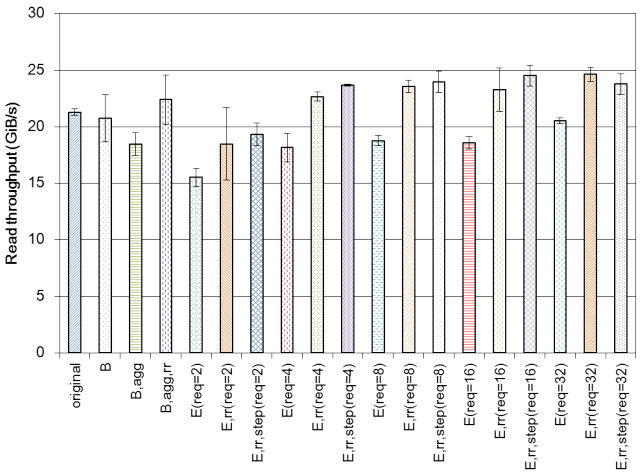


図 13: HPIO ベンチマークによる性能（6,144 プロセス，$8 \times 6 \times 32$ ノード使用の下で，アグリゲータを 3,072 プロセスに設定）

約 15%の性能向上を達成した.

6. 関連研究

集団型 MPI-IO の高速化に関しては，様々な研究が行われてきており，ターゲットの計算機やファイルシステムの構成や特性を活かした実装が提案されている．例えば View-based I/O [20] では，TP-IO でのファイル I/O とデータ通信の繰り返し毎にファイルアクセスパターンに関する情報交換をプロセス間で行っている部分を，開始前に全てのパターン情報を集計しておくことで性能向上を実現している．ただし，プロセス数の増加に伴い，この前処理のコストが増加する問題がある．一方，我々の実装はオリジナルの TP-IO と同様に毎回アクセスパターン情報集計のためのデータ通信を行うが，アグリゲータ配置の最適化によりファイル I/O およびデータ通信の高速化を図っており、これによる性能向上のメリットは，特に使用するノード数やプロセス数の増加に伴い向上することが期待される.

各アグリゲータが担当するファイルアクセス領域を並列ファイルシステムでのストライピングパターンに配慮したレイアウトにすることで高速化を狙ったものとして LACIO [21] がある．並列ファイルシステムの物理的な分散データレイアウトに合わせた最適化を行うことでファイルシステム側へのアクセス時の通信経路の混雑緩和を実現している．同様の試みは Lustre 向け ADIO 実装 [13] でも行っており，我々が提案する EARTH でも採用している．また，[22] では，LIOProf と呼ばれる Lustre でのプロファイリングツールの検証の中で，集団型読み込み関数に対し，ストライピングアクセス指向の TP-IO 実装を施したものを使用しているが，その実装に関する詳細は不明である．EARTH でも同様と思われるストライピングアクセス指向の実装を行っているが，EARTH ではノードあたりに複数のプロセスが配置される場合での最適なアグリゲータ配置方法や I/O throttling 機構，段階的通信機構等も提案しており，プロセスやノード群のレイアウトを考慮している点で，これらの研究とは異なる.

EARTH において採用している I/O throttling 機構と同様の取り組みは，これまでにもファイル I/O における高速化の一手法として提案されてきている．例えば，[14] では，I/O サーバへのアクセスにおいて，I/O ストリームの throttling 機構を取り入れ，書き込みや読出し処理を行うタスク数を調整することで高速化を実現している．ADIOS [23] では，POSIX-I/O によるファイルアクセスにおいて，I/O 要求の調停機能による throttling 処理による高速化を実現している．各々のノードの I/O 処理は調停役のプロセスにより発行タイミングが制御され，throttling による性能向上を実現している．一方，EARTH では，ファイルアクセスのみならず，データ通信でも段階的通信機構による処理時間の短縮を行っており，アグリゲータ配置の最適化と併用することで更なる性能向上を実現している点でこれらの研究とは異なる.

7. まとめ

各計算ノードに複数のプロセスが配置される場合においても，プロセス配置や使用するノード形状，さらには配下の並列ファイルシステムの構成や特性等も考慮した EARTH を提案した．EARTH の実装の一貫として，「京」の上で実装した EARTH on K による性能向上をこれまでに集団型書き込みにおいて確認していたが，今回，新たに集団型

読み込みに対しても同様の最適化を行った．今回行った性能評価から，プロセス配置を考慮したアグリゲータの配置方法やデータ通信順の最適化の有用性が確認できた．さらに I/O throttling 並びに段階的データ通信を組み合わせることにより，さらなる性能向上が可能になることも確認できた．

今後の課題としては，I/O throttling および段階的データ通信において，同時に発行する最適なリクエスト数の選定方法がある．例えば TP-IO 内部の主要な 3 つの処理であるファイル読み込み，データ通信，およびファイル書き込みの所要時間などから最適な設定を見つけることなどが考えられる．今後，実現可能性も含め検討したい．また，「京」におけるファイル読出しでの先読み機能におけるキャッシュメモリサイズの上限がノードあたり 128 MiB となっているが、このようにキャッシュサイズの上限の制約が厳しい環境に対応した，アグリゲータからの読出しサイズの調整機能についても検討する必要があると考えている．さらに，TP-IO でのファイル I/O とデータ通信の干渉緩和に関して，通信経路等の事前予測によるデータ通信の発行順のさらなる最適化の可能性等の検討を進める．

謝辞 MPI-IO ライブラリの拡張実装を進めるにあたり，志田直之氏をはじめ，富士通株式会社の関係者のご支援を頂きました．また，本論文の結果は，理化学研究所のスーパーコンピュータ「京」を利用して得られたものです．

参考文献

[1] MPI Forum: `http://www.mpi-forum.org/`.
[2] Message Passing Interface Forum: *MPI-2: Extensions to the Message-Passing Interface* (1997).
[3] Thakur, R., Gropp, W. and Lusk, E.: On Implementing MPI-IO Portably and with High Performance, *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, pp. 23–32 (1999).
[4] Lustre: `http://lustre.org/`.
[5] The National Center for Supercomputing Applications: `https://www.hdfgroup.org/`.
[6] Parallel netCDF: `http://cucis.ece.northwestern.edu/projects/PnetCDF/`.
[7] Thakur, R., Gropp, W. and Lusk, E.: Optimizing noncontiguous accesses in MPI-IO, *Parallel Computing*, Vol. 28, No. 1, pp. 83–105 (2002).
[8] Tsujita, Y., Hori, A. and Ishikawa, Y.: Locality-Aware Process Mapping for High Performance Collective MPI-IO on FEFS with Tofu Interconnect, *Proceedings of the 21st European MPI Users' Group Meeting*, EuroMPI/ASIA '14, pp. 157–162 (2014).
[9] Tsujita, Y., Hori, A. and Ishikawa, Y.: Striping Layout Aware Data Aggregation for High Performance I/O on a Lustre File System, *High Performance Computing - 30th International Conference, ISC High Performance 2015, Frankfurt, Germany, July 12-16, 2015, Proceedings*, pp. 282–290 (2015).
[10] Tsujita, Y., Hori, A., Kameyama, T. and Ishikawa, Y.: Topology-Aware Data Aggregation for High Performance Collective MPI-IO on a Multi-Core Cluster System, *Proceedings of 2016 Fourth International Symposium on Computing and Networking*, IEEE Computer Society, pp. 37–46 (2016).
[11] Tsujita, Y., Hori, A., Kameyama, T., Uno, A., Shoji, F. and Ishikawa, Y.: Improving Collective MPI-IO Using Topology-Aware Stepwise Data Aggregation with I/O Throttling, *Proceedings of HPC Asia 2018 (to be published)*, ACM (2018).
[12] Thakur, R., Gropp, W. and Lusk, E.: An Abstract-Device Interface for Implementing Portable Parallel-I/O Interfaces, *Proceedings of the Sixth Symposium on the Frontiers of Massively Parallel Computation*, pp. 180–187 (1996).
[13] Lustre: Lustre ADIO collective write driver, Technical report, Lustre (2008).
[14] Seelam, S., Kerstens, A. and Teller, P. J.: Throttling I/O Streams to Accelerate File-IO Performance, *High Performance Computing and Communications, Third International Conference, HPCC 2007, Houston, USA, September 26-28, 2007, Proceedings*, Lecture Notes in Computer Science, Vol. 4782, Springer, pp. 718–731 (2007).
[15] Ma, S., Sun, X.-H. and Raicu, I.: I/O Throttling and Coordination for MapReduce, Technical report, Illinois Institute of Technology (2012).
[16] Ajima, Y., Inoue, T., Hiramoto, S., Takagi, Y. and Shimizu, T.: The Tofu Interconnect, *IEEE Micro*, Vol. 32, No. 1, pp. 21–31 (2012).
[17] 宇野篤也，庄司文由，横川三津夫：ファイルステージングのあるジョブスケジューリングの評価，情報処理学会研究報告，Vol. 2012-HPC-136, No. 22 (2012).
[18] Hirai, K., Iguchi, Y., Uno, A. and Kurokawa, M.: Operations Management Software for the K computer, *Fujitsu Sci. Tech. J.*, Vol. 48, No. 3, pp. 310–316 (2012).
[19] Ching, A., Choudhary, A., keng Liao, W., Ward, L. and Pundit, N.: Evaluating I/O Characteristics and Methods for Storing Structured Scientific Data, *Proceedings 20th IEEE International Parallel and Distributed Processing Symposium*, IEEE Computer Society, p. 49 (2006).
[20] Blas, J. G., Isaila, F., Singh, D. E. and Carretero, J.: View-Based Collective I/O for MPI-IO, *CCGRID*, pp. 409–416 (2008).
[21] Chen, Y., Sun, X.-H., Thakur, R., Roth, P. C. and Gropp, W. D.: LACIO: A New Collective I/O Strategy for Parallel I/O Systems, *Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium (IPDPS '11)*, IEEE Computer Society, pp. 794–804 (2011).
[22] Xu, C., Byna, S., Venkatesan, V., Sisneros, R., Kulkarni, O., Chaarawi, M. and Chadalavada, K.: LIOProf: Exposing Lustre File System Behavior for I/O Middleware, *2016 Cray User Group Meeting* (2016).
[23] Lofstead, J., Zheng, F., Liu, Q., Klasky, S., Oldfield, R., Kordenbrock, T., Schwan, K. and Wolf, M.: Managing Variability in the IO Performance of Petascale Storage Systems, *2010 International Conference for High Performance Computing Networking, Storage and Analysis (SC'10)*, IEEE, pp. 1–12 (2010).