Vol.2014-HPC-146 No.17
2014/10/3

集団型MPI-IOの高速化に向けたI/Oリクエスト発行方式の最適化

辻田　祐一[1,2,a)]　堀　敦史[1,2,b)]　石川　裕[3,1,c)]

概要：京や FX10 で利用されている並列ファイルシステムである FEFS を用いた MPI-IO による並列入出力の性能向上のために，我々は，FEFS でのストライピング処理に最適化された MPI-IO ライブラリの実装を行っている．本実装は，特に，空白データ領域を含む不連続アクセスパターンに対する集団型書き込み処理に対する性能向上を目指しており，これまでにストライピングパターンに合わせた最適化や FEFS や Tofu インターコネクトの構成に応じた入出力を行うプロセスの最適化配置などを実装し，その有用性を確認してきている．本稿では，これまでに行った実装に対して，更なる入出力性能向上を目指し，各クライアントから FEFS の OST へのアクセス方式の最適化の検討と実装を行った．今回行った実装に対するベンチマークプログラムによる性能評価の結果から，今回提案するトークンリレー方式による I/O 要求の発行方式の有用性を確認すると共に，同一 OST に送られる I/O 要求数を，ストライピングサイズに応じ調整することで，常に高い I/O 性能が維持できる可能性を確認した．

キーワード：MPI-IO, Lustre, ROMIO, two-phase I/O, アグリゲータプロセス

1. はじめに

京や FX10 において，Lustre [1] をベースに開発された FEFS （Fujitsu Exabyte File System) [2] が並列ファイルシステムとして利用されている．MPI[3] による並列計算に対しては，Open MPI [4] をベースに富士通で開発された MPI ライブラリ [5]（FJMPI) が提供されており，MPI-IO の機能は MPI-IO 実装の一つである ROMIO [6] をベースに FEFS 向けに実装されたものが利用できる．

MPI-IO は様々な I/O アクセスパターンをサポートしているが，特徴的なアクセスパターンの一つとして派生データ型を用いた集団型 I/O が挙げられる．多次元のデータをプロセス間で分割するようなアプリケーションでは，ファイルへのアクセスが不連続パターンになることが多く，多数の小さいデータ領域を個別にアクセスするのは効率が悪いため，ROMIO においては Two-Phase I/O [7] と呼ばれる高速化手法が用いられている．Two-Phase I/O では，プロセス間でアクセス対象のファイル領域全体を個々のプロセスのアクセス対象データとは無関係にプロセス間で連続に均等に分割し，read-modify-write の手法により，ファイルから読み込んだデータを一時的にメモリ上に保管し，書き込むデータをこのメモリ上に上書きしてファイルに書き戻しすることで空白領域は変更せずに非連続データ領域に書き込みができる．

FEFS は，Lustre と同様にメタデータを管理する Meta Data Server（MDS) と実際のデータの入出力を管理する複数の Object Storage Server(OSS) によって構成され，MDS および OSS それぞれの配下にストレージとしての Meta Data Target(MDT) ならびに Object Storage Target(OST）が接続されている．FJMPI による FEFS への並列入出力においては，使用する FEFS の OST の数とは無関係に，I/O を行うプロセス（アグリゲータ）間でデータを分割し，割り当てられた OST 群に対し，各々のプロセスがストライピング設定に基づいてアクセスしている．この場合，アグリゲータとなるプロセス群と OST 群の間のデータ通信パターンによっては，通信経路の混雑や OST へのアクセス集中などが発生し，性能低下に繋がる可能性がある．

著者らはこれまでに集団型書き込みの性能向上に向けた FEFS 向け MPI-IO 実装の改変を行っており [8], [9]，既に Lustre 向けには採用されているストライピング処理でのアクセスパターンに合わせたアグリゲータ間データ分割 [10]

[1] 理化学研究所　計算科学研究機構
[2] JST CREST
[3] 東京大学
[a)] yuichi.tsujita@riken.jp
[b)] ahori@riken.jp
[c)] ishikawa@is.s.u-tokyo.ac.jp

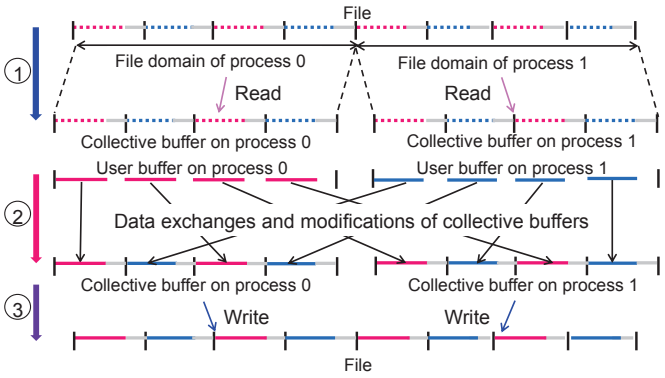


図 1 Two-Phase I/O による集団型書き込みの処理の流れ

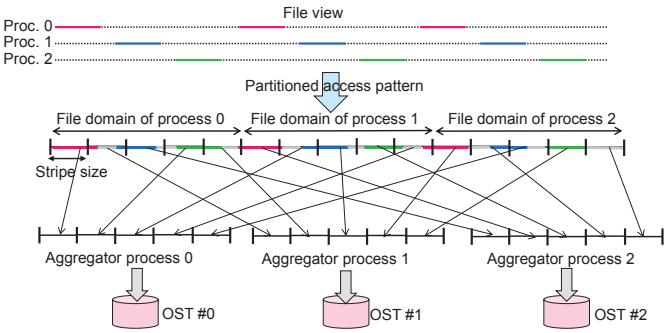


図 2 TP-IO におけるストライプパターンに合わせたアグリゲータ間のファイル I/O 処理の例

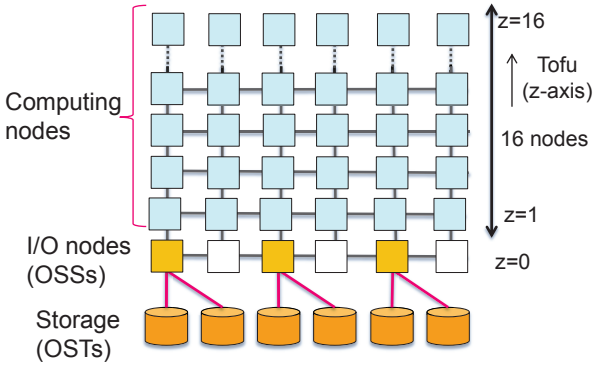
や，計算ノード群と Tofu 並びに FEFS のネットワーク構成に応じたアグリゲータ配置の最適化等を行ってきている．本稿では，更なる高速化に向けた取り組みとして，ファイル I/O を行うアグリゲータから OST への I/O リクエストの発行方法に関する最適化について報告する．今回行った実装に対するベンチマークプログラムによる性能評価の結果から，ストライピングサイズに応じて，同一 OST に同時に送信する I/O リクエスト数を調整することによって常に高い性能を維持できる可能性を確認した．ストライプサイズが小さい時は個々の I/O リクエストのデータ長も短くなるため，同時に送信する I/O リクエスト数を多くし，逆にストライピングサイズが大きく，個々の I/O リクエストのデータ長が長くなる時には，同時に送信する I/O リクエスト数を少なくすることが I/O 性能の向上に繋がった．以下，第 2 章で，本実装の対象である Two-Phase I/O のメカニズムについて説明し，第 3 章で実装対象である京および FEFS についてストライピング機能と Tofu インターコネクトとの関係について説明する．その後，FEFS に対する Two-Phase I/O に対して行った最適化実装に関して第 4 章で説明し，さらに本実装に対して行った性能評価と考察について第 5 章で報告する．その後，関連研究について第 6 章で本研究との比較検討を行い，最後に第 7 章で本稿のまとめを行う．

2. Two-Phase I/O

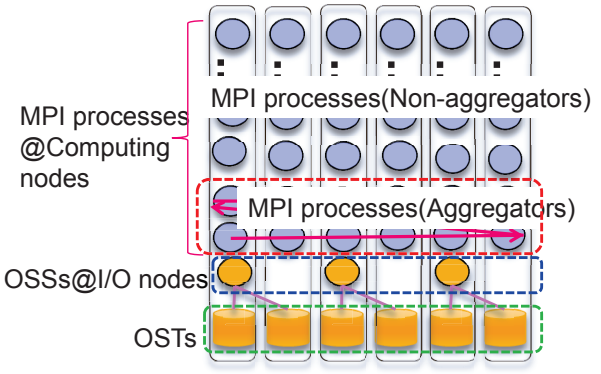
Two-Phase I/O（以下，TP-IO）による集団型書き込みでの処理の流れを**図 1** に示す。I/O 処理を行う MPI プロセス間で対象のファイル領域を連続に均等に分割し，担当領域から一時的なメモリ領域へのデータ読出しを行う（図中の 1）．なお，この一時的なバッファ領域を TP-IO では，Collective buffer（以下，CB）と呼んでおり，ファイルアクセスやプロセス間のデータアクセスの基本サイズになる．ファイル上の空白部分を含む読み込まれたデータに対し，予め計算しておいた書き込むべきデータに関するオフセットおよびデータ長の情報を基にして，個々のデータ領域を担当するプロセス間のデータの送受信と CB でのデータ書き込みが行われる（図中の 2）．書き込むべきデータが CB に上書きされた後，各プロセスは CB 内のデータをファイル上の対象領域に書き戻す（図中の 3）．実際には CB は有限の大きさのため，この一連の処理（以下，サイクル）を複数回繰り返して I/O 処理を完了させる．なお，ファイルアクセスを行う MPI プロセスを TP-IO ではアグリゲータと呼んでいる．この図では全ての MPI プロセスがアグリゲータの役割を担っているが，一部のプロセス群にのみアグリゲータを担当させることもできる．

FJMPI による派生データ型を用いた FEFS での集団型 MPI-IO においても，TP-IO により，高速な I/O 処理を実現しているが，オリジナルの FJMPI による TP-IO は，アグリゲータ個々に集められるデータは，ファイル上のデータレイアウト（ファイルビュー）をベースに分割されたレイアウトになっている．この方式では各アグリゲータはアクセス対象の全ての OST に対しストライピング処理をするため，アグリゲータと OST 間のデータ送信が混雑し，かつ OST 上でのファイルアクセスも非連続なアクセスが頻発し，全体として処理性能が十分に出せない可能性が想定される．さらに，アグリゲータプロセスの選出に関しては，FEFS のストライピングパターンや Tofu 上の各 MPI プロセスの配置とは無関係に MPI のランク順にアグリゲータとして割り当ててゆくため，MPI プロセスの配置パターンによってはストライピング処理によるアクセスに対して十分な性能を発揮できない可能性も考えられる．

一方で，最近の Lustre 向けの ADIO では，ストライピングパターンに合わせたアグリゲータ間のデータ並び替えにより高速化が実現されている [10]．この動作例を**図 2** に示す．アグリゲータ上に集められるデータは既にストライピングパターンに合わせているため，各アグリゲータは特定の OST へのファイルアクセスのみとなり，OST へのファイルアクセスの混雑が解消される．さらに OST 上でのファイルアクセスも連続なデータレイアウトになり，全体の I/O スループット向上が実現されている．



(a) 京における計算ノードとローカルファイルシステム用 FEFS の一部



(b) 京の FEFS 向けに最適化したアグリゲータ配置の一例

図 3 京における (a) ローカルファイルシステムでの FEFS と計算ノード群とのネットワーク結合例並びに (b)FEFS 向けに最適化したアグリゲータ配置の一例

3. FEFS における MPI-IO 実装

京においては，フロントエンド・計算ノード双方からアクセスできるグローバルファイルシステムと計算ノードからのみアクセスできるローカルファイルシステムがある．両者ともに FEFS が利用されているが，前者は恒常的にファイル等が保管される場所で，一方後者は計算処理でのデータ入出力に用いられるため，それぞれにおいて FEFS を構成するインターコネクトやストレージが異なっており，後者は特に入出力性能を重視した構成になっている．本実装は MPI-IO による並列入出力の高速化を目指しているため，後者のローカルファイルシステムでの高速化を目指している．

京のローカルファイルシステムは，**図 3**(a) に示すように，Tofu の z 軸単位で一つの OST を共有する構成になっている．そこで我々は図 3(b) に示すようなアグリゲータとなるプロセスの最適化配置機能を実装している．個々の OST に対応する同一の z 軸上にある MPI プロセスのグループ分けを行い，ストライピングパターンに沿って各グループからプロセス 1 個ずつアグリゲータとして割り当ててゆくことによって，ストライピングパターンレイアウトに合わせ，かつ個々の OST に特定のアグリゲータ群が対応する構成になるようにプログラムコードの改変をこれまでに行っている [8], [9].

この最適化配置とストライピングパターンに合わせたアグリゲータ間データ分割により，入出力性能が向上することをこれまでに確認しているが，アグリゲータ群から OST 群への I/O リクエストの発行方式に関しては，全プロセスが同時に I/O リクエストを発行するよりも，各 z 軸上で形成されたグループで，対応する OST に対し逐次的にリクエストを送る方が性能向上に繋がることを確認している．これは各 OSS や OST の処理能力を超えるの多数の I/O リクエストが同時に送信されることによって，かえって性能低下を招いてしまうのを防ぐ目的で行っており，またこれは POSIX-I/O による OST への書込み処理の高速化で確認されている結果 [11] にも基づいている．

なお，[11] においては，クライアントから同時には 2 個の I/O リクエストを送る時が最も性能が高いことが報告されているが，この研究報告では，ストライピング処理は行わず，MPI プロセス個々に割り当てられるランクディレクトリと呼ばれているプロセス個々の専用のローカルファイルシステムに対し，MPI プロセス毎に別々のファイルに対する書き込み処理を行うケースを対象としている．一方，我々の場合，POSIX-I/O ではなく MPI-IO によるローカルファイルシステムにおける共有ファイル生成とストライピング機能を用いた並列 I/O 処理を対象にしている．よって最適な I/O リクエスト数などが異なることが想定される．本稿では，この I/O リクエストの発行方式に着目し，FEFS における集団型 MPI-IO の高速化を目指した最適化実装を検討した．

4. 高速化に向けた I/O リクエスト発行方式に関する考察と実装

FEFS に対する I/O リクエストの発行は，**図 4** に示すような処理フローにより実現されている．FEFS における OSS と I/O リクエストを発行するクライアントの間では，この図に示すように，Remote Procedure Call (RPC) による通信が行われている．なお，この図では O_DIRECT モードによる処理フローの例を示している．このモードでは，クライアント側である計算ノードから I/O 要求が発行されると，直ちに OSS に RPC による OSS との通信が開始する．この例ではクライアントの数が 1 個であるが，複数のクライアントからの要求がある場合など，OSS に対する負荷に応じて，スレッドプールより予め設定されたスレッド数の最大値に達するまで，必要に応じて複数のスレッドが追加起動される．

よって，一度に多くのクライアントから大量の I/O 要求が発行されると，OSS 上の多数のスレッド間での資源の取り合いが発生するなど，OSS や OST で処理できる I/O リ

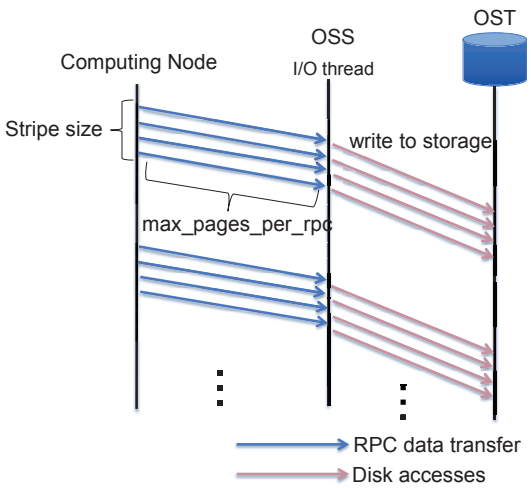


図 4 書込み処理における FEFS でのクライアント・OSS・OST 間での I/O 要求の処理フローの例

クエスト数あるいはデータ量を超えてしまう．その結果，待ち状態になる I/O リクエストが多数発生し，性能低下に繋がる可能性が懸念される．そこで，第 3 で述べたように，我々は既に各々の OST にアクセスするアグリゲータ群に対し，I/O リクエストを逐次的に発行する最適化を提案している．この実装により，同時に全てのアグリゲータが OST にアクセスする場合に比べて性能が向上する可能性を確認している．しかしながら [8] にて報告した際には，同時に 1 個ずつ発行する方式であったため，OSS や OST では，それより多い I/O リクエストを処理できる余裕があるものと考えられる．

そこで，本稿では，同時に発行する I/O リクエスト数の複数化を中心に最適化実装の可能性と検証を行うこととし，これまでの実装に対して，さらに以下の最適化実装を提案する．

- トークンリレー方式による I/O リクエスト逐次発行機能の実装
- 同時に発行できる I/O リクエスト数の複数化（書き込み処理および読み込み処理各々での対応の組合せ）

前者は，これまでの Tofu の同一 z 軸上にあるアグリゲータ間でのバリア同期による I/O リクエスト逐次発行機能に対する改善策として行うものである．Tofu の各 z 軸上で形成されているアグリゲータ群のグループ内で，バリア同期を行いながら次々と各プロセスから I/O リクエストを発行すると，リクエスト発行の度にバリア同期のオーバーヘッドが発生してしまう。更にこの同期方式では，グループ内の全てのアグリゲータが I/O リクエストを発行するまで次の処理フェーズに進むことができず，非効率的であった．そこで，I/O 要求を発行したアグリゲータが，次に発行するアグリゲータに MPI_Send/MPI_Recv 対によりトークンを渡してゆくトークンリレー方式を提案する．この方式では，リクエストを発行する際には，2 つのアグリゲータ間で

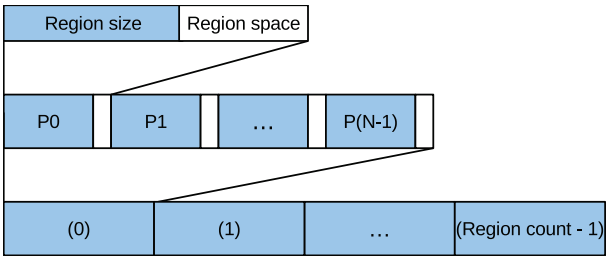


図 5 HPIO ベンチマークによる不連続データパターン生成

のみ通信を行うため，全体で同期をせずに済み，先に I/O リクエストを発行したアグリゲータは，I/O リクエストを発行していないアグリゲータを待たずに先の処理フェーズに進むことができ，高速化が図れるものと期待される．

次に後者については，TP-IO 内部の読み込みフェーズおよび書込みフェーズそれぞれにおいて OST 側への I/O リクエスト数の制御を行うためのものである．本稿では，書込みフェーズでの I/O リクエスト数の制御を基本とし，読み込みフェーズにおける I/O リクエスト数の制御機能を追加するか否かの 2 パターンを実装し，性能評価から最適な実装を探し出すことにした．

5. 性能評価

本稿では，Lustre での ADIO 実装に倣ったストライピングパターンに合わせたアグリゲータ間データ分割方式を取り入れた TP-IO を基準として，今回実装した最適化機能により，どの程度性能が向上するかを検証した．今回は，以下の最適化の組合せにより高速化実装の評価を行った．

(1) Tofu の 6 次元座標を基にしたアグリゲータの最適化配置（以下，OPT1）
(2) Tofu の同一 z 軸上のアグリゲータ群から OST へのバリア同期式により逐次化されたアクセス方式
  (a) 書き込みフェーズのみ対応（OPT2-1）
  (b) 書き込み・読み込み両フェーズで対応（OPT2-2）
(3) Tofu の同一 z 軸上のアグリゲータ群から OST へのトークンリレー方式により逐次化されたアクセス方式
  (a) 書き込みフェーズのみ対応（OPT3-1）
  (b) 書き込み・読み込み両フェーズで対応（OPT3-2）

改変を加えた実装に対し，空白データを含む不連続なデータパターンへの集団型並列 I/O の評価を HPIO ベンチマーク [12] により性能評価を行った．HPIO ベンチマークは**図 5** に示すように 3 つのパラメタ（region size，region space 並びに region count）により，様々なアクセスパターンをサポートしている．本稿では,region size=3,744 B，region space=256 B とし，さらに region count=192000 と設定することで，192 プロセス全体で約 137 GiB のデータの I/O を行うようにした．なお，出来る限り I/O の振舞いを調べ易くするために，O_DIRECT モードに設定し，アグリゲータから OST へのアクセス方法を変更して性能評価

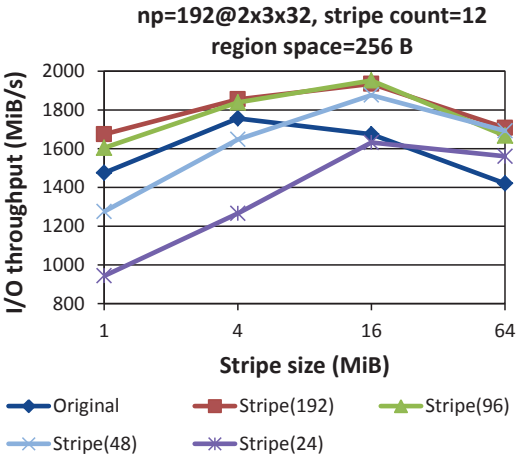


図 6 既存実装とストライピング向けアグリゲータ間データ並び替えのみを適用した実装での書込み処理性能

を行った．

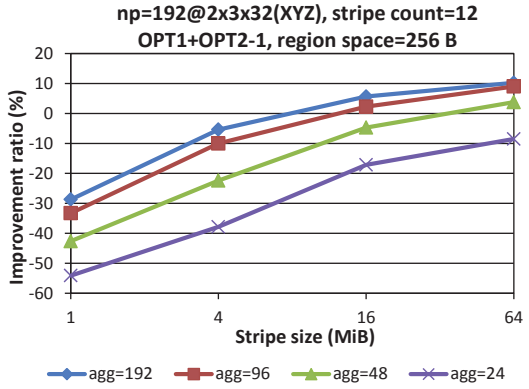
5.1 既存実装とストライピング向け実装での性能

まず始めに評価の基礎データとなる既存実装とストライピングパターン向けアグリゲータ間データ並び替えのみを施した実装での集団型書き込みの性能を**図 6** に示す．この図から，ストライピング向け最適化実装（図中の Stripe：括弧内数字はアグリゲータプロセスの数）において，特に全プロセスあるいはその半分がアグリゲータとなるケースでは既存実装（図中の Original）よりも高い性能を示しているのが分かる．しかし，アグリゲータの数をさらに減らしてしまうと既存実装よりも性能が低下してしまうため，アクセスパターンや使用する環境等を配慮した適切なアグリゲータ数の設定が必要であることも分かる．以下，このストライピング向け実装を比較対象として，さらに新たに実装した最適化手法によるスループット値での性能向上率を調べ，各々の最適化実装での特徴や適用可能性を検討した．

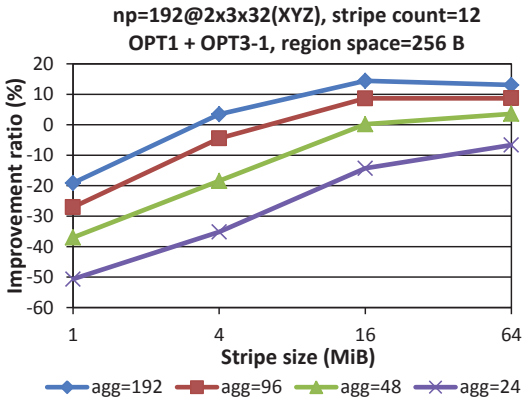
5.2 OST への逐次アクセス方法の最適化

前述のストライピング向け実装に対し，まず最初に，同一 OST にアクセスする Tofu の同一 z 軸上にあるアグリゲータ間で I/O リクエストを 1 個ずつ発行する実装について，これまでのバリア同期による実装（OPT2-1 および OPT2-2）と今回提案するトークンリレー方式での実装（OPT3-1 および OPT3-2）の比較検討を行った．なお，第 4 章で述べた通り，TP-IO における書込みフェーズへの実装を基本とし，読み込みフェーズへの適用の有無での性能比較を行った．

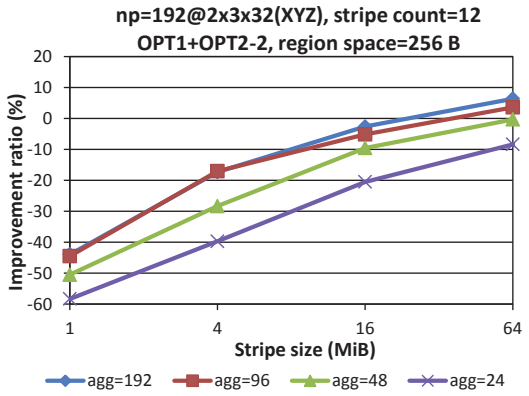
各方式でのアグリゲータ数の変化に伴う I/O スループットに関する性能向上率を**図 7** に示す．この結果から，トークンリレー方式の方がバリア同期式よりも高い性能が得られることが分かった．第 4 章で説明した通り，バリア同期式では，書込み要求を発行するたびに同一 z 軸上のアグリ



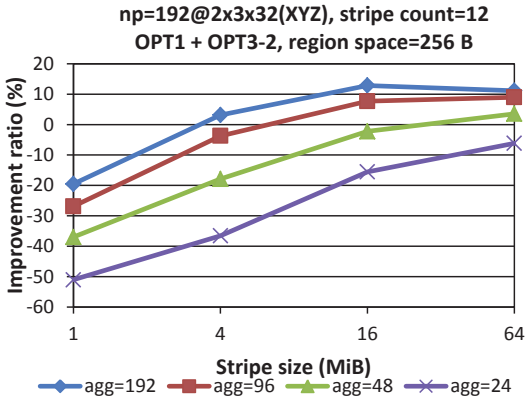
(a) 書込みのみ同期（バリア同期式）



(b) 書込みのみ同期（トークンリレー方式）



(c) 読み込み・書き込み共に同期（バリア同期式）



(d) 読み込み・書き込み共に同期（トークンリレー方式）

図 7 アグリゲータ数を変化させた時の性能向上度

ゲータ間でバリア同期を行い，更にグループ内の全てのアグリゲータがI/Oリクエストを発行するまで次の処理に進むことが出来ない．一方，トークンリレー方式においては，I/Oリクエスト発行が完了したアグリゲータと，次にI/O要求を発行するアグリゲータとの間でのトークン受け渡しでI/O発行のタイミングを制御しており，各々のアグリゲータはI/O発行完了後に直ちに次の処理に進むことが出来る．よって，後者の実装で更なる性能向上が実現できた．

さらに，いずれの方式においても読み込みフェーズでの同期を行わない方が性能上有利であることも分かった．これは，直前の書込みフェーズが早く完了したアグリゲータほど，より先に読み込みフェーズを開始するが，仮に書き込みより読み込みの方が速い場合には，先に読み込みが終了したアグリゲータはトークンを渡すべきアグリゲータが読み込みフェーズを開始するまで待ち状態になる可能性がある．実際に今回試験したケースでは読み込みの方が書き込みよりも速かったため，上記のような問題が発生していたと考えられる．

5.3 OSTへのI/Oリクエスト数に対する性能評価

これまでは，アグリゲータからOSTへのI/Oリクエストは同時には1個のみ送られる実装になっていたが，POSIX I/Oの書込み処理のFEFSへの性能向上に関する研究[11]では，同時に2個のI/Oリクエストを送信する時が最も高い性能を示すことが報告されている．ここでは，MPIプロセス個々にファイルを独立に生成し，POSIX I/Oによる書き込みを行っているため，我々が行っている実装と若干異なる点はあるが，OSTへのアクセスの点では共通する部分が多いため，同時に発行するI/Oリクエストの数を1個から増やしてゆき，性能向上に関する評価を行った．なお，前の章で確認した通り，トークンリレー方式による実装が性能上有利なので，以後の評価ではこの方式で評価を進めた．

図 8 は同時にOSTに送られるI/Oリクエストの数を変えた時の性能向上度を示している．この結果から，ストライピングサイズが小さい時にはI/Oリクエスト数を多めに，逆にストライピングサイズが大きい時にはI/Oリクエスト数を少な目にする方が性能向上に繋がりやすい傾向が分かった．

前者のケースではI/Oリクエストのデータ長が小さくなり，OSSやOST側が，より多くのI/Oリクエストを受け付けることが可能になる一方，後者のケースではI/Oリクエストのデータ長が長くなり，複数のリクエストを受け付ける余裕が無くなるためだと考えられる．

更に，これまでのスループットでの評価では分からない内部処理の振舞いを調べるため，TP-IO内部の各々の処理について個々のMPIプロセスで処理時間を計測し，各々の

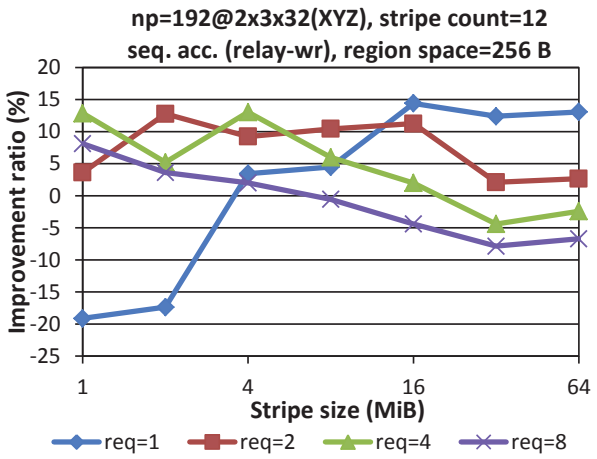
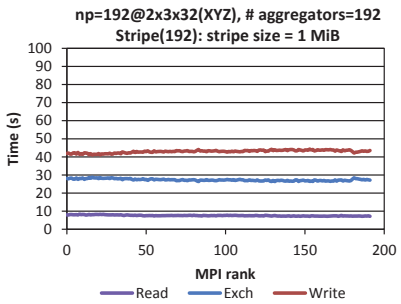


図 8 同時にOSTに送られるI/Oリクエスト数（*req*）を変えた時の性能向上度

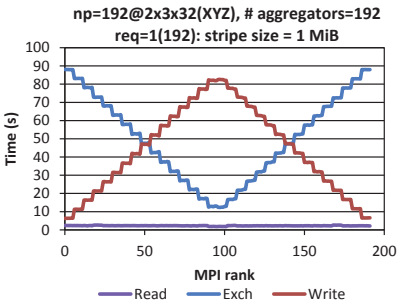
プロセス個々で求めた平均値から，I/Oリクエスト数の影響を調べた．ここでは，ストライピングサイズが小さい場合および大きい場合を比較するために，全てのMPIプロセスがアグリゲータになるケースにおいて，ストライピングサイズがそれぞれ1 MiBならびに64 MiBの時のTP-IO内部の処理時間をプロセス毎に平均値を取ったものを，横軸をランク番号として，それぞれ**図 9** および**図 10** に示した． なお，空白データ領域の有無をチェックするフェーズは他の3つのフェーズに比べて無視できる時間であるため，残り3つのフェーズについて調査した．比較のためにトークンリレー方式によるI/Oリクエスト発行最適化機構を取り入れたものに加え，性能評価の基準としているストライピングパターンに基づいたアグリゲータ間データ分割のみを取り入れた実装でも同様の計測を行った．

図9及び図10から，いずれのストライピングサイズにおいても，全体的に書き込みフェーズについてはI/Oリクエスト数の最適化が適用された実装では，ランク番号が真ん中あたり（96付近）で書き込み時間が最も長く，それよりも小さく，あるいは大きくなるにつれて書込み時間が短くなっているのが分かる．これは，今回の評価に用いたMPIプロセスのマッピング方式により，ランク番号が96付近のプロセスが，OSTへのI/Oアクセスが最も遅く発行されていたためである．それよりも小さいあるいは大きいランク番号になるにつれて，より先に書き込みのI/Oリクエストが発行されていたため，処理時間がより短くなっている．図中の階段状の分布の一定時間になっている平坦な領域にあるプロセスの数がI/Oリクエスト数の増加と共に増えているが，これは設定したI/Oリクエスト数の増加に伴い，同時にI/Oリクエストを発行するアグリゲータの数が増えたためである．

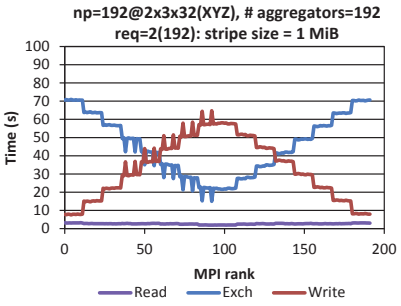
一方，この最適化を適用していないストライピング向け最適化のみのケースでは，全てのアグリゲータが同時にOSTにアクセスしにゆくため，全体的にプロセス間で同程



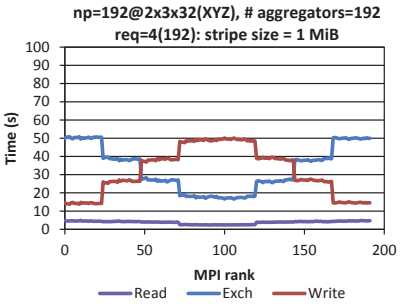
(a) ストライピング向け最適化のみ



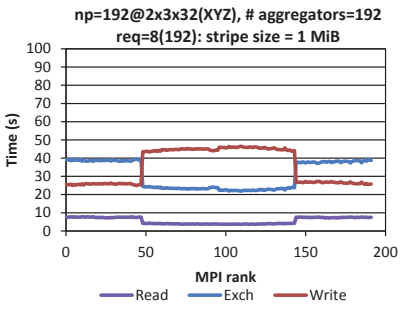
(b) トークンリレー方式（req=1）



(c) トークンリレー方式（req=2）

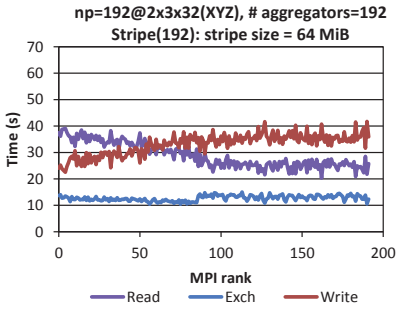


(d) トークンリレー方式（req=4）

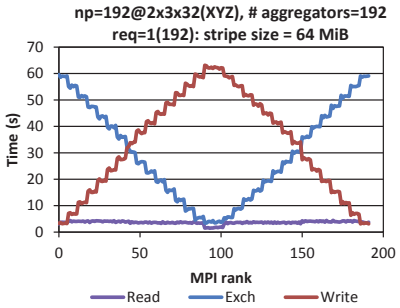


(e) トークンリレー方式（req=8）

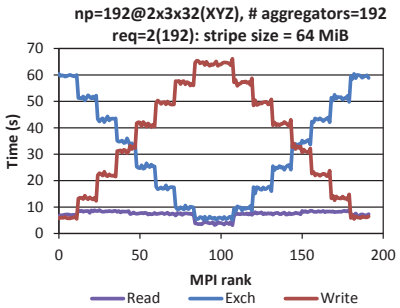
図 9 TP-IO 内部の処理時間（192 プロセス全てがアグリゲータ）：Stripe size=1 MiB



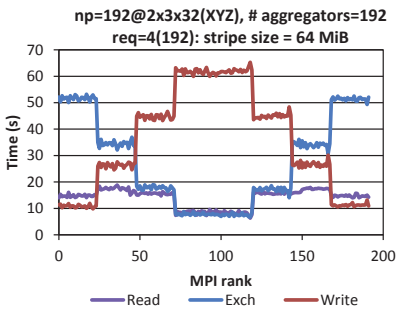
(a) ストライピング向け最適化のみ



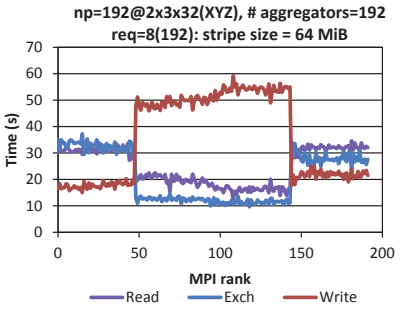
(b) トークンリレー方式（req=1）



(c) トークンリレー方式（req=2）



(d) トークンリレー方式（req=4）



(e) トークンリレー方式（req=8）

図 10 TP-IO 内部の処理時間（192 プロセス全てがアグリゲータ）：Stripe size=64 MiB

度の処理時間になっていたと考えられる．また，ストライピングサイズが 64 MiB と大きい場合，ランク番号が 0 での処理時間が最も短く，ランク番号が進むにつれて徐々に増加し，途中からはほぼ一定時間となっている．これはランク番号順に配置されたアグリゲータの配置により，ストライピングのデータレイアウト自体もランク番号順に並んでいたためで，ランク番号が小さいプロセスほどより先頭のデータを持つために，書き込み処理が先に終了したためと考えられる．

次に読み込み処理については，最適化が適用されたケースでは，逆にランク番号で 96 付近が最も処理時間が短くなる傾向が見える．これは直前の書込み処理が前述の通りランク番号で 96 付近が最も遅く処理され，そこから離れるに従って，より先に処理されたことによる影響を受けたためである．

この読出しの処理の完了を受けて次のデータ交換フェーズが始まるが，これは MPI_Isend/MPI_Irecv 対による全プロセス間での総データ交換になり，MPI_Waitall による完了待ちとなるため，先にこのフェーズに入ったアグリゲータが，より処理待ちの時間が増え，結果としてデータ交換に要した時間が増えたものと考えられる．それゆえにランク番号で 96 付近のプロセスでは処理時間が短く，ランク番号がそこから離れるほど時間が長くなったと考えられる．これは，プロセス間のデータ交換フェーズの処理時間の最大値と最小値の差がほぼ書き込みフェーズの処理時間の最大値と最小値の差とほぼ等しいことからも分かる．

次に，ストライピングサイズの違いでの振舞いに着目すると，ストライピングサイズが 1 MiB の図 9 では，同時に発行する I/O リクエストの数が増えるにつれ，各々の処理時間が短縮してゆく傾向が見える．一方，ストライピングサイズが 64 MiB の図 10 のケースでは，I/O リクエストの数を増やしても時間が短縮する様子が見られない．これは，前者では，ストライピングサイズが小さいことにより，OSS 側がより多くの I/O リクエストを処理できる可能性があるのに対し，後者ではストライピングサイズが大きいことによって，I/O リクエストの数を増やしても，OSS 側で処理しきれる量を超えてしまい，時間の短縮に繋がらなかったものと思われる．よって，図 8 において確認した，性能が最も良くなる I/O リクエスト数に関しては，以上のことが大きな要因になっていると考えられる．

いずれのケースにおいても，トークンリレー方式による実装の場合，先にデータ交換フェーズを開始するプロセスは書き込みや読み込みのフェーズも先に開始しており，順番はアグリゲータ最適化配置の過程で作成されたマッピングテーブルに基づいている．このテーブルの中にはアグリゲータのマッピング順を表すインデックスと MPI_COMM_WORLD でのランク番号が格納されている．マッピングの順番はストライピングサイズ単位でのデータレイアウト順にもなっているため，このテーブルの先頭から順にランク番号を参照すれば，書き込み，読み込みやデータ交換を先に開始しているプロセスの順になっている．よって，これを基に，先にデータ交換フェーズを開始したプロセスとの非同期通信を順に MPI_Wait により 1 つずつ完了確認をすれば，通信完了待ちの時間を削減できる可能性があるものと思われる．これに関しては今後，実装の設計や性能上のメリットなどを含めて検討する．

6. 関連研究

TP-IO における高速化に向けた様々な取り組みがある中で，特にファイルビューの取扱いに関する最適化による高速化を実現したものとして View-based I/O [13] がある．TP-IO では，処理サイクルの繰り返しにおいて，サイクルごとにプロセス間でデータを入れ替える際に必要な各プロセスでのオフセット・データ長の情報をプロセス相互に通信している．それに対し View-based I/O はファイル I/O 開始時に全サイクル分のオフセット・データ長の情報を集めておくことで，サイクルごとに行うプロセス間の通信を無くし，全体の性能を向上させている．

アグリゲータの動的な選定手法による TP-IO の高速化が OpenMPI で利用できる MPI-IO ライブラリの一つである OMPIO において実現されている [14]．この実装ではファイルビューやプロセス数，プロセスのノードへの配置トポロジ，データサイズなどに加え，並列ファイルシステムのストライピングなどを考慮した上でアグリゲータの数を動的に決定している．しかしながらこの実装では，計算ノードと通信回線の設定を反映させていない点で我々の実装とは異なる．

アグリゲータの最適化配置については，Blue Gene/P（BG/P）における実装において有効性が報告されている [15]．この実装では，MPI-IO による集団型 I/O を利用するプログラムにおいて，BG/P の計算ノード間のネットワークトポロジと MPI プロセスの配置を考慮して，アグリゲータとなるプロセスがサブグループ化された各々の領域に配置されるように工夫しており，既存の MPI-IO 実装に比べて性能が向上することが報告されている．我々が行っているアグリゲータ配置の最適化は，ネットワーク構成を配慮している点において同じであるが，それぞれが対象とするハードウェア環境に特化した最適化実装のため，基本的なアイデアは同じと言えるが，性能向上に向けた実装方法などは異なる．さらに，我々の研究では，Tofu の z 軸単位でサブグループ化されたアグリゲータ群からアクセス対象の OST への I/O リクエストの数の調整を通して性能向上を狙っている点でも異なる．

7. 本稿のまとめ

京の FEFS に対する MPI-IO による集団型書き込みの高

速化に向けて，FEFS のストライピング設定および Tofu との接続形状に応じ，ストライピングパターンに合わせたアグリゲータ間でのデータ分割やストライピングパターンのデータレイアウトに沿ってアグリゲータとなる MPI プロセスを選定するアグリゲータ最適化配置などの実装を通してこれまでに高速化の可能性を探ってきたが，本稿では，さらなる高速化の可能性を探るべく，アグリゲータから OST への I/O リクエストの最適な数や発行方式について検証を進めた．

まず，これまでに実装した Tofu の同一 z 軸上のアグリゲータ群からの逐次的な I/O 要求発行の実装ではアグリゲータ群の間でバリア同期を取りながら I/O 要求を順番に発行していたが，同一 z 軸上の全部のアグリゲータからの I/O 要求が完了するまで次の処理に進めない問題があったため，トークンリレー方式に切り替えて，性能向上が実現できることを確かめた．トークンリレー方式では，I/O リクエストを発行したアグリゲータが次に I/O 要求を発行するアグリゲータとの間でトークンの送受信をし，I/O 要求を発行したアグリゲータは次の処理に進むことができるため，これにより性能が向上したものと考えられる．

次にアグリゲータ群から OST への I/O 要求の複数化については，ストライプサイズに応じて適切な数が変わる傾向を確認した．今回行ったベンチマークによる評価では，ストライプサイズが短い時には，I/O リクエストの数を多めにする方が，逆にストライプサイズが長い時には I/O リクエストの数を少なくする方が性能が出やすかった．これは OSS や OST で同時に処理することができる I/O リクエストの許容量との関係があると考えている．ストライプサイズが小さい時にはアグリゲータから OSS に送信される RPC によるリクエストのサイズが小さいサイズのデータを扱うために多くの I/O リクエストを同時に受け付けることができたものと考えている．逆にストライプサイズが大きいと，1 つの RPC が扱うデータサイズも大きくなり，I/O リクエストを少なくした方が OSS や OST での負担を軽減し，高い処理能力を維持しやすかったものと思われる．

運用上の制約から，OSS や OST での負荷を直接調べることが出来なかったが，ベンチマークによる評価の中で，TP-IO 内部の 4 つの処理フェーズで，特に時間がかかる読み込み・データ交換並びに書き込みの 3 フェーズについてプロセス個々に時間を計測した．その結果，書き込み処理では，I/O リクエスト数の分ずつ各 z 軸上のアグリゲータ群からの処理が終了している様子が確認できた．これを受けて，その後に行われる読み込み処理やデータ交換処理の時間が決まってくる様子も確認できた．現状では，書き込みと読み込みの後に MPI_Isend/MPI_Irecv 対による全アグリゲータ間でのデータ交換が行われる際に，早くデータ交換に入ったアグリゲータほど，より多く時間がかかっていた．これは非同期通信を MPI_Waitall により終了検知を行っていることで，最も遅くデータ交換に到達するアグリゲータを待つ時間に依るものであった．今後の課題として，先にデータ交換フェーズに入ったプロセスからデータの送受信完了をすることで，さらなる時間短縮に繋げられるか確認する予定である．また，トークンリレー方式による I/O リクエスト発行において，I/O リクエスト数を変化させた時の OSS 並びに OST での振舞いの調査についても，例えば Lustre が利用できる環境を整備するなどの対応により，今後進めてゆく予定である．

謝辞　本研究の一部は JST CREST「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」の採択課題「メニーコア混在型混在型並列計算機用基盤ソフトウェア」の支援を受けております．本研究の開発対象である FEFS 向け ADIO は（株）富士通様よりご提供頂いたソースコードを用いており，本ソースコードに関する技術情報を頂きました．また理化学研究所　計算科学研究機構の運用技術部門の山本啓二氏には京の FEFS の技術的な情報を頂きました．さらに理化学研究所　計算科学研究機構　システムソフトウェア研究チームの亀山豊久氏にはコード開発環境に関する技術的なアドバイスを頂きました．この場をお借りして感謝を申し上げます．

参考文献

[1] Lustre: `http://wiki.lustre.org/index.php/Main_Page`.
[2] Sakai, K., Sumimoto, S. and Kurokawa, M.: High-Performance and Highly Reliable File System for the K computer, *Fujitsu Sci. Tech. J.*, Vol. 48, No. 3, pp. 302–309 (2012).
[3] MPI Forum: `http://www.mpi-forum.org/`.
[4] Open MPI: Open Source High Performance Computing, `http://www.open-mpi.org/`.
[5] Shida, N., Sumimoto, S. and Uno, A.: MPI Library and Low-Level Communication on the K computer, *Fujitsu Sci. Tech. J.*, Vol. 48, No. 3, pp. 324–330 (2012).
[6] 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).
[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] 辻田祐一, 堀敦史, 石川裕: FEFS におけるストライピング処理を考慮した集団型 MPI-IO の実装, 情報処理学会研究報告 2014-HPC-145, 35 (2014).
[9] Tsujita, Y., Hori, A. and Ishikawa, Y.: Locality-Aware Process Mapping for High Performance Collective MPI-IO on FEFS with Tofu Interconnect, *Proceedings of EuroMPI/ASIA'14*, ACM, pp. 157–162 (2014) (accepted).
[10] Lustre: Lustre ADIO collective write driver, Technical report, Lustre (2008).
[11] 大野善之, 堀敦史, 石川裕: 並列ファイルシステムに対する I/O リクエスト調停機構の提案, 情報処理学会研究報告 2014-HPC-144, 13 (2014).
[12] 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).

[13] Blas, J. G., Isaila, F., Singh, D. E. and Carretero, J.: View-Based Collective I/O for MPI-IO, *CCGRID*, pp. 409–416 (2008).

[14] Chaarawi, M. and Gabriel, E.: Automatically Selecting the Number of Aggregators for Collective I/O Operations, *2011 IEEE International Conference on Cluster Computing (CLUSTER 2011)*, IEEE, pp. 428–437 (2011).

[15] Vishwanath, V., Hereld, M., Morozov, V. and Papka, M. E.: Topology-aware Data Movement and Staging for I/O Acceleration on Blue Gene/P Supercomputing Systems, *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, ACM, pp. 19:1–19:11 (2011).