Vol.2014-HPC-145 No.35
2014/7/30

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

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 における MPI-IO による並列入出力の性能向上のために，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た．京や FX10 の MPI ライブラリでは，MPI プロセス間でファイル領域を等しく分割し，全プロセスが OSS を通して OST とアクセスするようになっている．各プロセスは複数の OST 間をストライピングしてアクセスするため、OST 群へのアクセスによる通信経路の混雑や，OST 上の不連続なアクセスが多発することによる処理性能低下が想定される．本実装では，事前に各プロセスでの書込み前のデータレイアウトをストライピング処理に合わせたものにしておくことで，各 MPI プロセスから OST までの通信経路の混雑を回避し，かつ OST 上でのデータアクセスが連続になるようにした．これらにより，派生データ型による集団型書き込み処理での性能向上を実現した．

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] による並列計算支援環境としては，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 がある．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非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．

京や FX10 における 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，並列ファイルシステムのストライピング設定とは無関係に，I/O を行うプロセス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，I/O を行うプロセス群と 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，性能を上げにくい問題が発生しているものと考えられる．

一方で最近の ROMIO での Lustre 向け実装では，Lustre のストライピング設定を配慮した高速化実装が行われている [8]．この実装では，各クライアントプロセスからは１つの OST に対するアクセスになり，クライアントプロセス群と OST 群の間の通信経路の混雑を避けることができる．さらに OST への効率の良いアクセスも可能になり，

[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

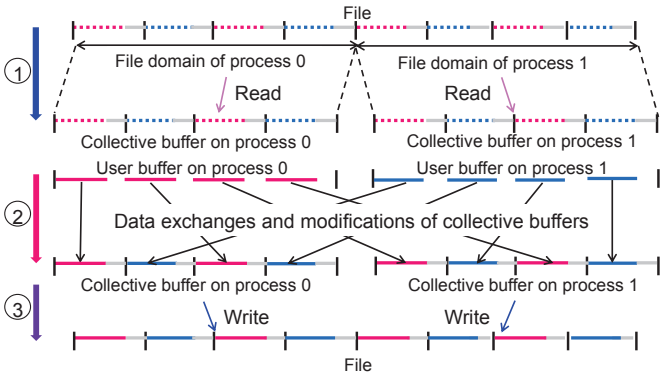


図 1 Two-Phase I/O による集団型書き込みの処理の流れ

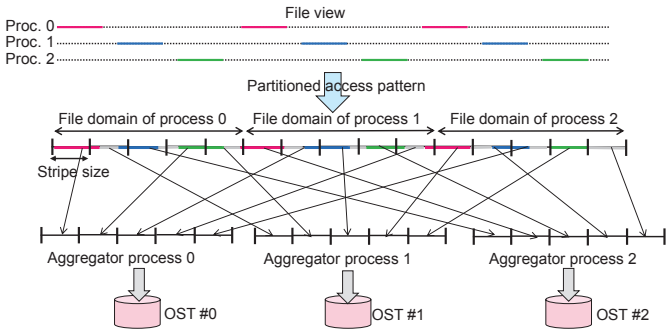


図 2 TP-IO におけるストライプパターンに合わせたアグリゲータ間のファイル I/O 処理の例

スループットの向上が実現されている．

本稿では，ROMIO の Lustre 向け実装に倣い，FEFS での MPI-IO 実装の改変を行うと共に，更なる高速化に向けて，計算ノードと I/O ノードの Tofu での接続形状を考慮した最適化実装を提案する．改変した実装を京においてベンチマークプログラムにより性能評価を行ったところ，派生データ型による集団型書き込みにおいて，Lustre で利用されているストライプパターンに合わせた Two-Phase I/O の方式のみを取り入れた実装に対して，最大で 10%程度の性能向上を確認した．以下，Two-Phase I/O の仕組みと本実装の手法をそれぞれ第 2 章および第 3 章で説明する．次に第 4 章で今回行った性能評価の結果について報告する．関連研究と本研究の相違点などについて第 5 章で述べた後に，最後に本稿のまとめを第 6 章で行う．

2. Two-Phase I/O

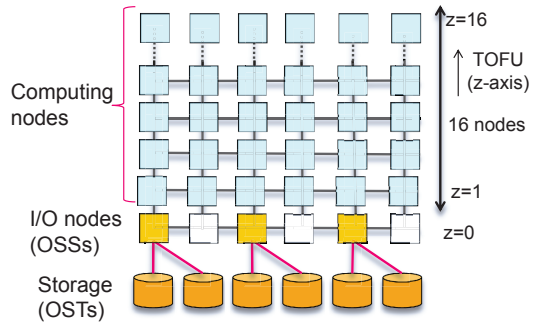
Two-Phase I/O（以下，TP-IO）による集団型書き込みでの処理の流れを**図 1** に示す。I/O 処理を行う MPI プロセス間で対象のファイル領域を連続に均等に分割し，担当領域から一時的なメモリ領域へのデータ読出しを行う（図中の 1）．なお，この一時的なバッファ領域を TP-IO では，Collective buffer（以下，CB）と呼んでおり，ファイルアクセスやプロセス間のデータアクセスの基本サイズになる．読み込まれたファイル上の空白部分を含むデータに対し，予め計算しておいた書き込むべきデータに関するオフセットおよびデータ長の情報を基にして，個々のデータ領域を担当するプロセスへの送信と 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ようになっている．

FJMPI による FEFS での派生データ型を用いた集団型 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書き込まれるデータは，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 ID 順にアグリゲータとしてゆくため，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．

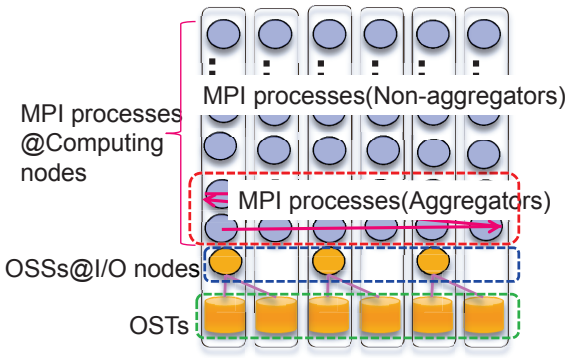
一方で，最近の Lustre 向けの ADIO では，ストライピングパターンに合わせたアグリゲータ間のデータ並び替えにより高速化が実現されている [8]．この動作例を**図 2** に示す．アグリゲータ上に集められるデータは既にストライピングパターンに合わせているため，各アグリゲータは特定の OST へのファイルアクセスのみとなり，さらに OST 上でのファイルアクセスも連続なデータレイアウトになり，全体の I/O スループット向上が実現されている．そこで本稿では，この方式に，さらに Tofu 上のノード配置と FEFS のストライピング情報を考慮したアグリゲータ割り当て方式を加えた最適化実装を提案する．

3. FEFS のストライピング設定を配慮した ADIO 改変実装

京の場合，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り，本実装はローカルファイルシステムでの最適化を目指している．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



(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例

図 3 京における (a) ローカルファイルシステムの FEFS と計算ノードの結合と (b)FEFS 向けに最適化されたアグリゲータ配置の例

ている．

そこで図 3(b) に示すように，個々の OST 毎に，対応する同じ z 軸上にある計算ノード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一つずつアグリゲータとなるプロセスを抽出し，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コードの改変を行った．

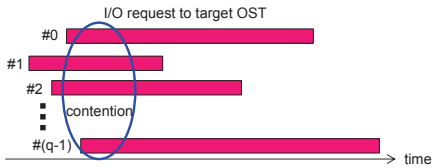
我々の提案手法では，ストライピングアクセスに合わせた TP-IO 処理を導入するにあたり，京や FX10 での FEFS 向け機能として，以下の最適化を実装した．

(1) Tofu および FEFS と使用する計算ノードに最適化したアグリゲータ配置
(2) 同一 z 軸のアグリゲータ間における OST への逐次 I/O アクセス

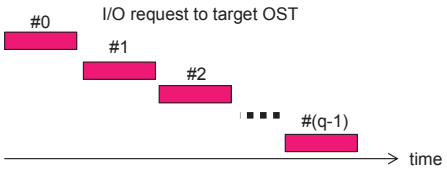
以下，上記の実装の詳細について説明する．

3.1 Tofu および FEFS との接続形態に合わせたアグリゲータ配置の最適化

FJMPI で利用されている FEFS 向け ADIO 実装では，全ての MPI プロセスが OST への I/O アクセスを行うアグリゲータの役割を担う設定になっている．アグリゲータの割り当ては，通常はランク ID が 0 番から昇順に設定されているため，プロセスが起動されているノードの配置は考



(a) 同一 z 軸上のアグリゲータ間で独立に OST にアクセスする場合



(b) 同一 z 軸上のアグリゲータ間で逐次的に OST にアクセスする場合

図 4 同一 z 軸上のアグリゲータ間から対象の OST へのアクセス手法

慮されていない．この場合，MPI プロセスの配置によっては OST への Tofu を介した通信に衝突・混雑が生じ，Tofu の性能を十分に発揮できない可能性がある．

このような衝突や混雑を可能な限り回避し，ストライピングパターンに合わせた高効率な OST へのアクセスを実現するために Tofu と FEFS 並びに計算ノードの接続形態に合わせたアグリゲータ配置手法を実装した．この実装では，京や FX10 で提供されている計算ノードの 6 次元 Tofu 座標とランク ID の変換を行う API を利用し，1 ノードに 1 プロセスの起動を想定している．

アグリゲータは OST に近い方から順にストライピングパターンに沿って順に配置されてゆき，かつ各アグリゲータでのデータレイアウトがストライピングパターンに合わせて並べ替えられているため，各アグリゲータは対応する OST に対して効率の良いファイルアクセスが期待できる．このように，同じ OST へアクセスするプロセス間で協調して逐次的に I/O アクセスさせる手法は，既に [9] で行われており，OST の負荷を低減することや I/O リクエスト間の競合を回避して性能向上を狙う意味で，本研究で行っているものと同じである．しかしながら，本研究ではアグリゲータ間で同じファイルに対するアクセスを行っており，文献 [9] における，プロセス個々にファイルを独立にアクセスする環境とは異なっている．

3.2 同一 z 軸のアグリゲータ間での OST への逐次 I/O アクセス

ストライピングパターンレイアウトに基づいてアグリゲータ上にデータを並べ替えて特定の OST にアクセスさせるようにする最適化を行った場合でも，Tofu の同じ z 軸上に並ぶアグリゲータ間では，対象の OST への I/O アクセスは**図 4**(a) に示すように，各アグリゲータ毎に独立に発行されてしまう．その結果，OSS での処理で I/O リクエ

スト間で競合が発生し，効率的に OST へアクセスすることが出来ないものと考えられる．これに対し，図 4(b) に示すように，アグリゲータ間で同期を取って逐次的に I/O アクセスを発行させることで，OSS での競合を軽減することができ，性能向上につながる可能性が期待できる．

京のローカルファイルシステム上で，MPI プロセス個々に別々のファイルへの書込みを行うケースにおいて，同じ OST にアクセスするクライアントプロセス群からの I/O 要求を逐次的に処理させることで，多くの I/O 要求を同時に独立に処理させるよりも高い性能が得られることが確認されている [9]．同時に OSS に到達する数多くの I/O 要求を協調させ，逐次処理させることで，OSS での I/O 要求間の競合を減らし，I/O 性能を向上させる点においては，この研究と同じである．文献 [9] との相違点は，この研究が MPI プロセス個々にファイルを別々にオープンしてアクセスし，各々の書込み性能を向上させる点に主眼を置いているのに対し，本研究では集団型 I/O による一つのファイルに対する全プロセスからの I/O 処理の性能を向上させる点で異なる．

4. 性能評価

FEFS 向け ADIO 改変実装に対し，京の 192 ノード（3 次元形状で 2×3×32）を用いて性能評価を行った．ここで 2×3×32 の形状を使った理由としては，割り当てられるローカルファイルシステムの OST を他のユーザのプログラムと共有させないためである．この形状によって全体で 6 個の I/O ノード（即ち 6 個の OSS）が割り当てられ，各 OSS は 2 つの OST を有するため，全体で 12 個の OST を利用した．そのため，I/O 処理で生成されるファイルのストライプカウントは 12 とした．

計算ノードへの MPI プロセスの配置方法で性能が変わる可能性があるため，本稿では全ての評価において Tofu の 3 次元形状に関して X・Y・Z の方向順で MPI プロセスを配置し，ストライプパターンに合わせた TP-IO を実装したものを基準データとして，以下の最適化の効果を評価した．

(1) Tofu の 6 次元座標を用いたアグリゲータ配置の最適化（以下，OPT1 と記す．）
(2) アグリゲータから OST への整列させたアクセス方式（以下，OPT2 と記す．）

派生データ型による集団型書き込み処理の評価のために本稿では HPIO ベンチマーク [10] を用いた．HPIO ベンチマークによるアクセスパターン例を**図 5** に示す．この図において，各々のプロセスがアクセスする最小データ領域の大きさが region size であり，region space で指定された空白部分を挟んで，プロセス数分だけ繋げたものを 1 ユニットとしている．さらにこのユニットをユーザが指定した個数分（region count）だけ繰り返したパターンで I/O 処理を行う．今回の試験では，region space を 256 B，region size を 3,744 B，region count を 100,000 と設定し，192 プロセス全体で約 137 GiB のデータの入出力をオリジナルの FJMPI とストライプパターンに合わせた TP-IO を取り入れたものに対して行った．計測では各々の設定毎に 12 回繰り返し，そのうちの最大と最小を除いた 10 回分で平均値を取ったものを計測データとした．なお，FEFS への I/O アクセスは，測定環境をできるだけ単純化させる目的から，ページキャッシュ効果を無効にするために O_DIRECT モードを設定した．

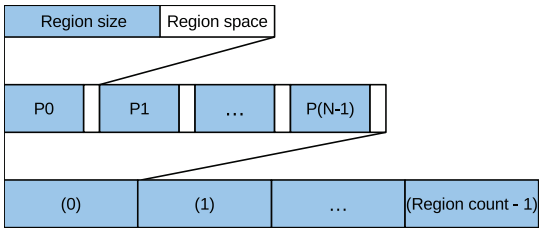


図 5 HPIO ベンチマークによる不連続アクセスパターンの生成

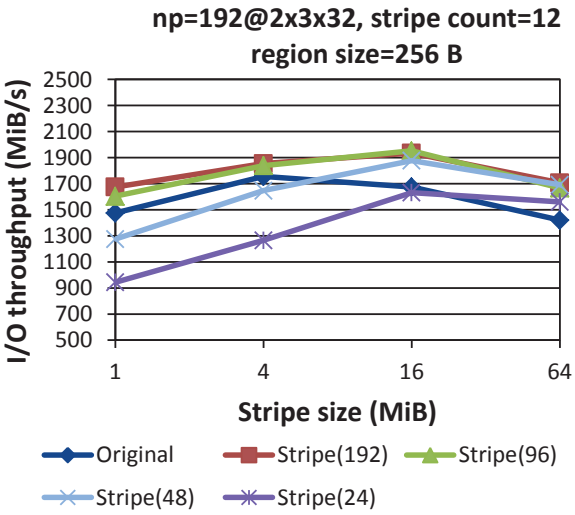


図 6 HPIO ベンチマークによる京でのオリジナルの FJMPI 並びに OPT1 の最適化を導入した実装によるスループット

はじめに参照データとして京の標準の FJMPI（GM-1.2.0-15) による性能を計測した。その結果を**図 6** に示す．この図において，”Original”はオリジナルの FJMPI，”Stripe”はストライピングパターンに合わせた TP-IO の計測データであり，後者の括弧内の数字はアグリゲータの数を表している．この図から分かるように，ストライピングパターンに合わせた TP-IO を導入したケースでは，アグリゲータ数を全プロセスの 192 あるいは半分の 96 としたときにはオリジナルの FJMPI よりもより高い性能が出ていることが分かる．逆にアグリゲータ数を減らしてゆくと，かえって性能を落とすため，基本的に全てのプロセスがアグリゲータとなることが望ましいと思われる．この計測データのうち，ストライピングパターンに合わせた TP-IO で最も高いスループットを達成したアグリゲータ数が 192 個

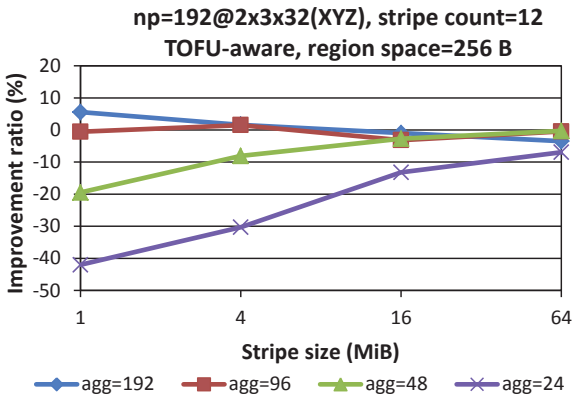


図 7 ストライピングパターンに合わせた TP-IO に対する OPT1 を導入した場合の性能向上率

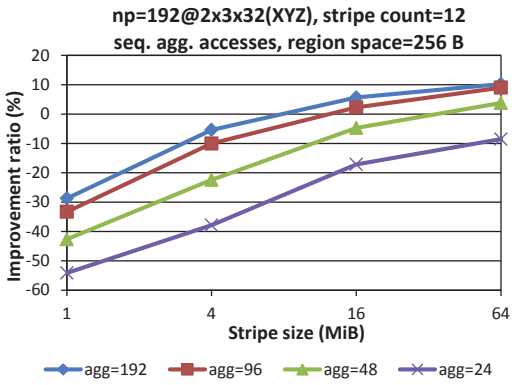


図 8 OPT1，OPT2 の両方を適用した実装の性能向上率

のケースを基準値として，以降の最適化実装によるスループットの向上率を検証した．

4.1 OPT1 の最適化の効果

ストライピングパターンに合わせた TP-IO に対し，上述の OPT1 の最適化を行った場合の評価を行った．その結果を**図 7** に示す．アグリゲータ数が 192 個の時に最も高い性能向上率を達成しているが，特にストライピングサイズが 1 MiB と短いところでの向上率が高く，最大で約 5.6% に達している．また，アグリゲータ数が少ないほどスループットは減少する傾向にあることも分かった．

4.2 OPT1 の最適化に OPT2 の最適化を加えた場合の効果

次に，前述の OPT1 の最適化を行った実装に対し，OPT2 の最適化を適用した実装に対しても性能評価を行った．その結果を**図 8** に示す．この評価結果から，最大で 10%の性能向上率を達成したが，このケースにおいては，特にストライピングサイズが 16 MiB 以上での性能向上率の増加が見られた一方で，ストライピングサイズが小さめの時の性能低下が一番大きかった。ストライピングサイズが大きい場合では TP-IO のサイクル数が少ないため，同期待ちの

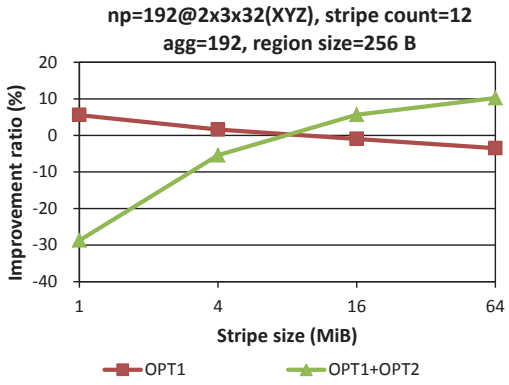


図 9 アグリゲータ数が 192 個の時の 2 つの最適化パターンでの性能向上率

コストが比較的小さく済み，サイクルごとに扱うデータ長も大きいために，アグリゲータから OST への逐次化されたアクセスによるメリットが得られやすく，高い性能向上が達成できたと考えられる．一方で，ストライピングサイズが小さいケースではサイクル数が多いために，かえってプロセス間での同期待ちのコストが無視できない程度に増大してしまい，さらにサイクルごとに扱うデータ長も短いため，大きな性能低下に繋がったものと考えている．

4.3 最適化実装による性能向上率に対する考察

以上の最適化実装の速度向上率から，ストライピングサイズ全般に渡って最適化をどのように図れば良いのかを検討した．以上の 2 ケースでの最適化に関して，全プロセスがアグリゲータとなるケースでの性能向上率を**図 9** に示す．図中の”OPT1”と”OPT1+OPT2”は，それぞれ OPT1 のみ適用した実装と OPT1 に OPT2 の最適化も組み合わせて実装したものである．

ストライピングサイズが小さいところでは OPT1 のケースが最も良く，ストライピングサイズが 16 MiB 以上と大きいところでは OPT1 と OPT2 の両方を実装したものが最も高い性能を示した．最適化実装をストライピングサイズの大きさに応じて変更する仕組みを取り入れれば，高速化において効果的な並列 I/O が実現できると考えられる．

5. 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 [11] 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 [12]．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．

6. 本稿のまとめ

我々は京の FEFS 向け MPI-IO 実装の高速化を目的に MPI-IO 内部のソフトウェアスタックである FEFS 向け ADIO に対し，ストライピングパターンに沿ったアグリゲータ間のデータ配置や Tofu および FEFS の構成に合わせたアグリゲータ配置を実装した．HPIO ベンチマークによる性能評価を行い，アグリゲータ間でストライピングパターンに沿ったレイアウトでデータを並べ替えるようにすることで，不連続な空白データを含む派生データ型を用いた集団型書き込みにおいて，Tofu 並びに FEFS の構成に合わせたアグリゲータの最適化された配置方法を適用することでストライピングサイズが小さいケースで，ストライピングパターンに合わせた TP-IO のみを実装したものに対し，最大で約 5%の性能向上を達成した．以上の最適化実装に対し，さらにアグリゲータから OST へのアクセスを同時に 1 個ずつの逐次処理になるように揃えることで，ストライピングサイズが 16 MiB 以上で，ストライピングパターンに合わせた TP-IO のみを実装したものに対し，最大で 10%の性能向上が得られ，提案する最適化手法の有効性が確認できた．

今回はベンチマークによる性能評価のみとなったが，TP-IO 内部の処理に要する時間を計測するなどにより，高速化を十分発揮できるための条件を特定することや，適切な最適化手法を選択できるような仕組みの検討と実装を今後の課題として進める．また，並列 I/O を用いるアプリケーション等による検証も含め，様々な異なるアクセスパターンによる性能向上の違いなどの評価も進める予定である．

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] Lustre: Lustre ADIO collective write driver, Technical report, Lustre (2008).
[9] 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).
[10] 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).
[11] Blas, J. G., Isaila, F., Singh, D. E. and Carretero, J.: View-Based Collective I/O for MPI-IO, *CCGRID*, pp. 409–416 (2008).
[12] 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).