Vol.2016-HPC-154 No.1
2016/4/25

プロセス配置を考慮した通信の最適化による集団型MPI-IOの高速化

辻田　祐一[1,a)]　堀　敦史[1,b)]　亀山　豊久[1,c)]　石川　裕[1,d)]

概要：並列計算機の規模の増大に伴い，そこで扱われるデータサイズも増加する傾向にある．大規模なデータを高速に扱うために，Lustre のような並列ファイルシステムを用いて MPI の並列 I/O インタフェースである MPI-IO を用いた高速並列 I/O が利用されている．MPI-IO の代表的実装として ROMIO が広く利用されており，集団型 I/O においては，Two-Phase I/O と呼ばれる高速化実装が用いられるが，各計算ノードに複数の MPI プロセスが配置される場合，MPI プロセスのランク配置によっては I/O 性能が低下する問題がある．そこで我々は Two-Phase I/O の実装内部の通信に着目し，プロセス配置を考慮した高速化実装を目指している．評価試験として東京工業大学の TSUBAME2.5 の 64 ノードを使い，HPIO ベンチマークを用いた 768 プロセスによる集団型 MPI-IO による書込み処理において，現行の ROMIO に対し最大で 67%の性能向上を確認した．

キーワード：MPI-IO, ROMIO, two-phase I/O, アグリゲータ, プロセス配置, データ通信

1. はじめに

近年の並列計算で扱われるデータサイズは計算規模の増加と共に益々大規模になり，ファイル入出力が性能ボトルネックの一つになってきている．そのために並列 I/O の性能向上が重要な課題となっており，並列ファイルシステムの役割が益々重要になってきている．並列処理で標準的に用いられる Message Passing Interface(MPI) [1] では，プロセス間の通信以外にも入出力インタフェースである MPI-IO も定められている．MPI-IO 実装の代表例として ROMIO [2] が挙げられ，開発母体の MPI 実装である MPICH [3] だけでなく，OpenMPI [4] 等でも MPI-IO 機能を担うライブラリとして利用されている．

MPI-IO が提供するインタフェースの中でも特に集団型 I/O は重要なものの一つであり，MPI-IO インタフェースを直接用いた場合だけでなく HDF5 [5] や Parallel netCDF [6] のようなアプリケーション向け I/O インタフェースにおいても並列 I/O 機能を提供するライブラリ実装として利用されており，MPI-IO 実装の高速化が益々重要な課題となっている．アプリケーションで良く使われるデータアクセスパターンとして，不連続なアクセスパターンに対する集団型 I/O があり，ROMIO においては，このようなアクセスパターンに対して Two-Phase I/O（以下，TP-IO）[7] と呼ばれる最適化実装を用いている．不連続なアクセスパターンに対してアクセス対象のみファイル I/O を行うと小さなファイル I/O 操作をシーク操作と共に数多く行う必要があり，著しく性能が低下してしまう問題があるが，TP-IO は，有限な大きさのバッファサイズ単位でデータ通信とファイル I/O を組み合わせた処理を繰り返すことで高速化を実現している．ここでデータ通信は各プロセスが持つデータをファイル I/O 向けのデータレイアウトに並べ替えて，連続なデータ領域を生成するために行われ，連続なデータのファイル I/O を実現することで性能向上に繋げている．データ通信に余分なコストがかかるが，ファイル I/O での性能向上のメリットが十分大きいために，TP-IO 全体での性能向上が可能になる．

ROMIO ではファイル I/O の処理を MPI プロセスの一部あるいは全部に割り当てており，ファイル I/O を行うプロセスをアグリゲータと呼んでいる．各ノードに複数のアグリゲータを配置する場合，割り当ての順番がノード毎に詰める形でファイル I/O 処理を割り当てるために，Lustre [8] のような並列ファイルシステムを用いる場合の入出力に不向きなレイアウトになる問題があった．これについては我々は既に性能向上の案としてアグリゲータのレ

[1] 理化学研究所　計算科学研究機構
[a)] yuichi.tsujita@riken.jp
[b)] ahori@riken.jp
[c)] kameyama@riken.jp
[d)] yutaka.ishikawa@riken.jp

イアウトをノード間でラウンドロビンにすることで性能向上が実現できることを示している [9], [10].

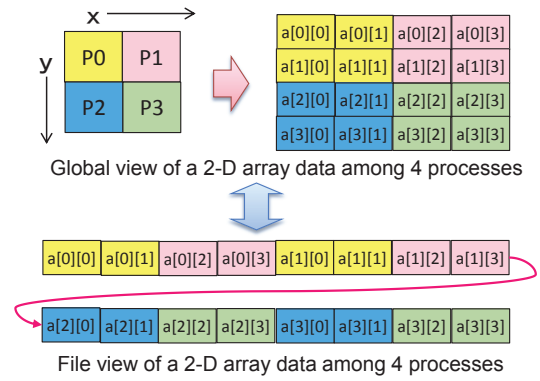
一方で TP-IO におけるデータ通信に関しては，全プロセスが非同期通信関数である MPI_Isend および MPI_Irecv を用い，ランク順にデータの送受信を発行しており，プロセス配置を配慮した設計になっていない．よって通信開始時にランクが前の方のプロセスに通信が集中する可能性があり，通信コストの増加に繋がる問題が考えられる．特に各ノード内に複数のプロセスがあり，ノード毎に詰めてランクが割り当てられる場合では，通信開始時に特定のノードに通信が集中してしまい，この場合にも通信コストの増加の可能性がある．よって今後のノード数・プロセス数の増加は TP-IO 内部の処理におけるデータ通信コストの増加に繋がるため，通信コストの低減が今後の重要な課題になっている．

そこで我々はノード内に複数のプロセスが配置されるケースに対し，プロセス配置に配慮した TP-IO のデータ通信の最適化実装を提案する．本実装では MPI プロセス群のノード間のランク配置をプログラムの実行開始時に確認し，ランク情報と配置情報からデータ通信の最適な順番を生成することで上述の特定のランクやノードに通信が集中するなどの問題を解消し，結果として TP-IO による集団型 MPI-IO の高速化を実現している．この機能は ROMIO 実装内部での通信やファイル I/O を最適化するローカルなランク配置情報等を生成するもので，ユーザが設定したランク配置を変更させるものではない．ユーザが設定したランク配置に依存せずに，可能な限り最適な通信順を生成しており，ユーザが容易に利用できる実装になっている．この実装を東京工業大学の TSUBAME2.5 の 64 ノードを用いて 768 プロセスを起動し，HPIO ベンチマーク [11] により不連続なアクセスパターンに対する集団型 MPI-IO の書込み性能を評価した結果，現行の ROMIO に対して得られた最大 I/O スループットとの比較で約 67%の性能向上を確認した．またアグリゲータの配置のみ最適化した実装と比較しても，約 16%の性能向上が確認できた．

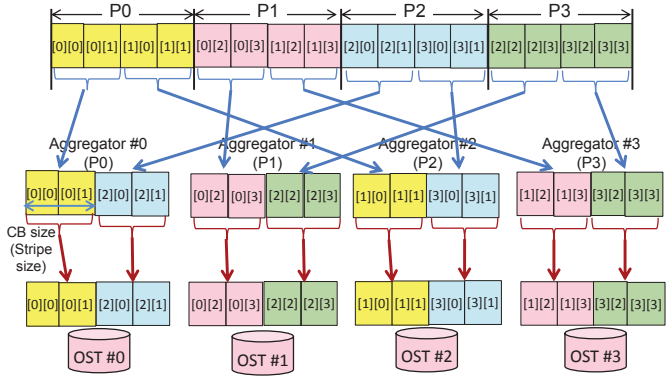
本稿では，まず第 2 章において，本実装の開発背景にある現行の ROMIO の TP-IO 実装と TP-IO 実装内のデータ通信の問題点について説明した後に，本提案手法について第 3 章で説明する．次に第 4 章にて今回行った性能評価試験の結果について報告する．第 5 章では関連研究について本提案手法との比較検討を行い，最後に第 6 章で本稿のまとめを行う．

2. Two-Phase I/O

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．この図では例として 4 プロセス間で格子状に分割された 2 次元配列に関する I/O を行う際のファイルビューの例（図 1(a)）と TP-IO による集団型 MPI-IO による書込



(a) 4 プロセス間でブロック分割された 2 次元配列とファイルビューの例



(b) 2 次元配列に対する TP-IO による集団型 MPI-IO による書き込み処理の例

図 1 TP-IO による集団型書き込みの処理の流れ（4 プロセス間での例）．図の上から下に向かって TP-IO の 1 ラウンドで行われている処理の流れを示しており，上から順にアグリゲータへのデータ送信とファイルへの書き込み処理が行われている．

み処理の流れ（図 1(b)）を示している．図 1(a) に示すように 2 次元配列を 4 プロセス間でブロック分割した場合，各々のプロセスのファイルビューは不連続なアクセスパターンになってしまい，データ領域のみアクセスする方式では I/O 性能が著しく低下してしまう．そこで ROMIO では TP-IO と呼ばれる最適化実装を用いて高速な I/O を実現している．その動作例を 図 1(b) に示している．TP-IO では，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処理を行う．書込み対象のデータはファイルビューに基づいて対象のアグリゲータにデータが通信により集められる．この際に集められる一時的なメモリ領域を 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．この図では Lustre を用いた例を示しており，CB の大きさはストライプサイズの大きさに設定される．CB に集められたデータは各々のアグリゲータが対象の OST に対し書込むことで TP-IO の 1 ラウンド分の処理が完了する．この図の例では 2 ラウンドの処理を示しているが，CB の大きさとファイルサイズによって決まるラウンド数（以下，TP-IO サイクル）だけ繰り返す．

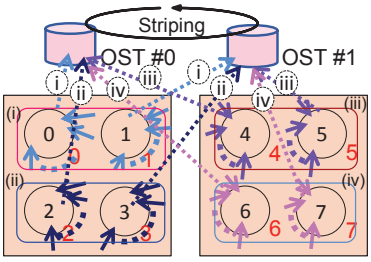


図 2 全プロセスがアグリゲータになる場合の現行の ROMIO による Lustre へのアクセスパターンとアグリゲータ間のデータ送受信パターンの例．四角は一つのノードを表し，丸の中の数字は MPI ランクを表している．また，角の丸い四角は同じストライピングアクセスのラウンドで動作するプロセス集団を明示している．

PC クラスタにおける ROMIO を用いた Lustre への集団型 I/O においては，デフォルトの設定では各ノードに 1 プロセスずつアグリゲータを配置する．さらに I/O 性能を高める場合などに，ユーザ側で `MPI_Info_set` によりノードあたりのアグリゲータの数を増やすこともできる．このアグリゲータの割り当ては，使用するノードのホスト名のリストを用いた ROMIO の実装内部の選出機能が担っており，ノード毎に詰めてゆく形でアグリゲータを配置している．

ROMIO は様々な並列ファイルシステムにも対応するために，下位レイヤに ADIO と呼ばれるファイルシステム向けドライバレイヤを持っている．この ADIO には MPI-IO インタフェースレイヤとの接続部分になる共通インタフェースレイヤがあり，その下に各ファイルシステムに特化したドライバレイヤが置かれている．Lustre も ADIO レイヤでサポートされており，Lustre 向けに最適化された高速化実装が実現されている [12]．この高速化実装は，Lustre のストライピング情報を事前に取得しておくことで，各アグリゲータ上で Lustre のストライピングパターンに合わせたデータ並べ替えを行っており，これにより各アグリゲータは特定の OST へのアクセスに限定されるため，アグリゲータ間での OST への書き込み処理の混雑が解消されている．

しかしながら複数の CPU を有する PC サーバを用いた近年のクラスタ環境では，CPU が持つコア数も増えるにつれて，ノードあたりに複数の MPI プロセスを起動することも容易になり，上述のようにノードあたりに複数のアグリゲータを配置した場合に，TP-IO を用いた Lustre への並列入出力処理が効率的に動作しないケースが出てくる．例えば**図 2** に示すように，各ノードに 4 プロセスずつ配置し，2 ノード全体で 8 プロセスを起動して 2 個の OST を有する Lustre に対し集団型書き込みを行う場合を考える．この図において，同じラウンドでストライピングアクセスをするプロセス群を角の丸い四角で囲んでおり，図中の (i) から (iv) はストライピングアクセスのラウンド番号と対応している．この場合，現行の ROMIO のアグリゲータ配置では，Lustre へのストライピングアクセスにおいて，同じノードのアグリゲータからそれぞれの対象となる OST へのアクセス（読出しおよび書き込み）が発生する（例えば，ランク 0 およびランク 1 から，それぞれ OST#0 および OST#1 へのアクセス：図中の **i**）．この場合，当該ノードから OST#0 および OST#1 へ繋がるネットワークの帯域を共有することになり，スループットが低下する可能性がある．また，アグリゲータは担当するファイルドメイン内で必要なデータを自分自身を含む全プロセスとの間で `MPI_Isend` と `MPI_Irecv` により送受信するが，このデータの送受信においても問題がある．例えば (i) のグループのランク 0 及び 1 のプロセスは，それぞれ自身を含む 8 個のプロセスからデータを受け取るが，両者とも同じノードにあるため，ノードへの通信が混雑する可能性がある．これらによって TP-IO の性能が低下する可能性がある．

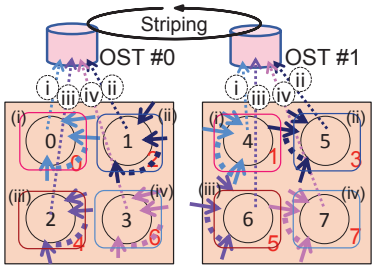


図 3 Lustre でのストライピングアクセスパターンを配慮したアグリゲータ配置最適化を適用した実装でのデータ送受信の例

そこで我々は**図 3** に示すような Lustre のストライピングアクセスパターンを配慮したアグリゲータ配置による高速化実装を行っており，これにより集団型 MPI-IO の性能向上を実現している [9], [10]．この実装では，MPI プロセスのランク配置に関係無く，ノード間でラウンドロビン配置になるようにアグリゲータを配置させている．この手法により，各々のストライピングラウンドにおいて，ノードあたり 1 個のアグリゲータから 1 個の OST に対しアクセスする形になるために通信経路や OST に対するアクセスの際の混雑を解消できる．その結果，I/O 性能の向上を可能にしている．しかしながらアグリゲータ群へのデータ収集に関しては現行の ROMIO の実装の通信方式のままで，各プロセスがランク順に通信関数を発行していた．この場合，通信開始時に先頭の方のランクに通信が集中するなどにより，アグリゲータ全体でのデータ収集に要する時間が長くなる可能性がある．

3. プロセス配置に配慮したデータ収集操作

ノードあたり複数のプロセスを配置させる場合に，前述の TP-IO でのアグリゲータへのデータ収集操作に要する時間が長くなる問題に対し，プロセスのノード間配置に配慮した通信方式を本稿では提案する．現行の ROMIO で

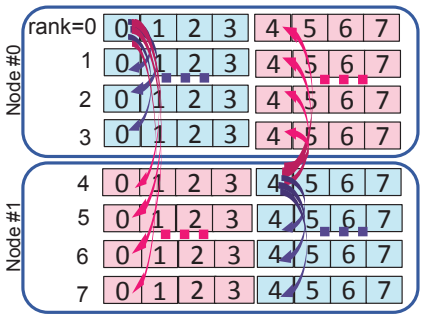


図 4　ノードあたり 4 プロセスずつ詰めて配置した場合の現行 ROMIO の TP-IO におけるプロセス間の送受信関数発行順．図中の四角内の数字は通信相手のランクを表しており，青色の四角はノード内通信，ピンク色の四角はノード間通信を表している．また矢印は MPI_Irecv の通信データの流れを示しており，濃い赤色のものはノード間通信を，濃い青色のものはノード内通信を表している．

は通信相手のランクを 0 から昇順に通信関数を発行している．また，ノード内に複数のプロセスがある場合，ノード間通信とノード内通信が混在しているが，これに対する配慮もなされていない．よって，この方式では通信開始時の通信処理の開始時間がプロセス間でずれが生じるなどにより，通信コストの増加に繋がる可能性がある．これに対し我々はノード間のランク配置を I/O 処理開始前に確認し，これを基に最適な通信発行順を生成させることで通信時間の短縮に繋げることにより集団型 MPI-IO 処理の高速化に繋げる手法を提案する．

ユーザによって様々な MPI プロセスのランク配置が想定されるが，現行の ROMIO においては，既に述べたように通信性能が MPI プロセス配置に依存する可能性がある．例えば図 3 に示すような 2 ノードで各ノードに 4 つずつプロセスを詰めて配置している場合には，通信相手のランクに関して図 4 に示すようにランクが 0 から 7 まで昇順に各プロセスが非同期の送受信関数を発行する．この場合，ノード#0 上のランク 0 にあるプロセスの通信が早く終了する一方で，ランクが大きいプロセスほど通信完了までの時間が長くなる可能性がある．また，通信開始準備に時間を要する可能性のあるノード間通信より先に通信コストの低いノード内通信を先に行うのは通信時間短縮の観点から効率が悪い．

そこで，我々はランク順に発行する現行 ROMIO に対し，以下に述べる 3 種類の最適化手法を ROMIO に実装し比較検討を行った．以下，新たに実装した 3 種類の通信発行順を生成する手法を説明する．

3.1　ランク順を配慮した通信発行順（pairwise）

まずノード間配置は考慮しない通信発行順の最適化として，各々のプロセスが自身のランクから順に送受信相手のランクを 1 個ずつずらしながら通信関数を発行す手法（以下，pairwise）を実装した．この手法を 2 ノードで各ノードに 4 プロセスずつ詰めて配置した場合の通信発行順を図 5 に示す．このケースでは，事前にノード配置等に関する情報収集を必要としない為に，前述の手法のような事前準備に要する処理が不要であると共に，プロセス毎に異なる通信発行順となるため，この図に示すように通信処理が特定のランクに集中することを回避できる．

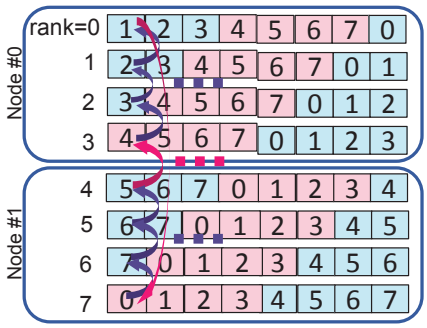


図 5　2 ノード構成で各ノードに 4 プロセスずつ詰めて配置した場合の pairwise 方式による通信発行順

Algorithm 1: ノード配置のみを配慮した通信発行順リスト生成（nd_shift）

```
Input: nprocs, nr_nodes, my_node_id, node_id[]
Output: comm_ranklist[]
1  list_idx ← 0
2  for i = 0 to nprocs − 1 do
3  |  used_rank[i] ← 0
4  end
5  for i = 0 to nr_nodes − 1 do
6  |  dest_node_id ← mod(my_node_id + i + 1, nr_nodes)
7  |  for j = 0 to nprocs − 1 do
8  |  |  if used_rank[j] == 0 and
   |  |  dest_node_id == node_id[j] then
9  |  |  |  comm_ranklist[list_idx] ← j
10 |  |  |  used_rank[j] ← 1
11 |  |  |  list_idx ← list_idx + 1
12 |  |  end
13 |  end
14 end
```

ただしノード毎に多くのプロセスを詰めて配置している場合には，ノード内通信がノード間通信よりも先行するプロセスが多くなるために，全体の通信時間が長くなる可能性がある．

3.2　ノード間のプロセス配置のみを配慮した通信発行順（nd_shift）

ノード間のプロセス配置のみを配慮した通信順決定の手法（以下，nd_shift）として Algorithm 1 に示すような実装を行った． MPI プログラム起動時に，各々のプロセスは配置されているノード ID のリスト (node_id[]) および自身が配置されているノード ID(my_node_id）やノード数（nr_nodes）を事前に取得しており，各々のプロセスが独立

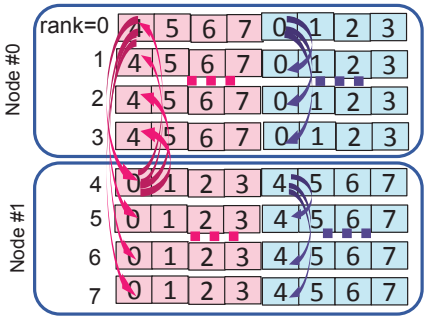


図 6　2 ノードで各ノードに 4 プロセスずつ詰めたランク配置における nd_shift 方式による通信発行順

に通信発行順リスト（comm_ranklist[]）を生成する．

2 ノードで各ノードに 4 プロセスずつ詰めて配置した場合にこの手法を適用した際の通信発行順を図 6 に示す．この手法では，自身が配置されているノードの次のノードをリストから割り出し，このノードにあるプロセスのランクを通信対象リストに追加してゆく．その結果，各々のプロセスはノード ID に関して昇順の方向にラウンドロビンでノード間通信を発行し，最後に自身のノード内の通信を発行する．これによりノード間通信が先行しかつ通信処理がノード間で均等に分散されるために通信時間の短縮が期待される．しかしながらノード内にある複数のプロセス間では同じ通信パターンとなってしまうため，ノード内のプロセス数が多い場合には通信時間が増加する可能性がある．

3.3　ノード間のプロセス配置とノード内ランク順を配慮した通信発行順（nd_rank_shift）

上で述べた nd_shift 手法において，さらにノード内の通信相手のプロセスが重複しないように配慮した手法として Algorithm 2 （以下，nd_rank_shift）を実装した． このケースでは nd_shift 手法と同様に通信相手のノード ID を自身のノード ID より大きい方にラウンドロビンでシフトするだけでなく，対象のノード内のプロセス群に対し，同じ通信フェーズで通信相手が重複しないようにノード内のランク順が同じ相手から順に通信を開始させるようにしている．そのため，nd_shift アルゴリズムにおける同一ノード内のプロセス間で通信順が同じになる問題を回避している．

2 ノードで各ノードに 4 プロセスずつ詰めて配置した際に，この手法を適用した場合の通信発行順を図 7 に示す．この図に示すように，前半のノード間通信では図 6 に示す nd_shift 方式の場合と同じパターンだが，後半のノード内通信では，上述の通り通信相手が重ならないように相手をずらしながら通信を行うようにしている．この結果，プロセス間での通信負荷の均等化が計られ，特にノード内のプロセス数が多い場合に通信時間短縮に繋がるものと期待される．

Algorithm 2: ノード配置とノード内のランク順を配慮した通信発行順リスト生成 (nd_rank_shift)

Input: hostlist[], my_hostname, nprocs, nr_nodes, my_node_id, my_node_rank, node_id[], node_nprocs[]
Output: comm_ranklist[]

```
1  list_idx ← 0
2  for i = 0 to nprocs − 1 do
3  |  used_rank[i] ← 0
4  end
5  for i = 0 to nr_nodes − 1 do
6  |  cand_idx ← 0
7  |  dest_node_id ← mod(my_node_id + i + 1, nr_nodes)
8  |  dest_node_nprocs ← node_nprocs[dest_node_id]
9  |  if dest_node_nprocs > 0 then
10 |  |  for j = 0 to nprocs − 1 do
11 |  |  |  if used_rank[j] == 0 and
   |  |  |  node_id[j] == dest_node_id then
12 |  |  |  |  cand_rank[cand_idx] ← j
13 |  |  |  |  used_rank[j] ← 1
14 |  |  |  |  cand_idx ← cand_idx + 1
15 |  |  |  end
16 |  |  end
17 |  |  for j = 0 to dest_node_size do
18 |  |  |  comm_ranklist[list_idx] ←
   |  |  |  cand_rank[mod(my_node_rank +
   |  |  |  j, dest_node_size)]
19 |  |  |  list_idx ← list_idx + 1
20 |  |  end
21 |  end
22 end
```

図 7　2 ノードで各ノードに 4 プロセスずつ詰めたランク配置に対し nd_rank_shift 方式を適用した場合の通信発行順

4.　性能評価

今回実装した ROMIO への拡張機能に対し，東京工業大学の TSUBAME2.5 の Thin ノードを用いて性能評価を行った．使用した TSUBAME2.5 の Thin ノードのネットワーク接続を含むハードウェア構成を表 1 に示す．TSUBAME2.5 では 2 つの InfiniBand 接続があり，計算ノードに加えてログインノードや Lustre 等が利用する 1st Rail と呼ばれるネットワークとは別に，計算ノード間のみを繋ぐ 2nd Rail と呼ばれるネットワークがある．本評価ではプロセス間の MPI 通信は 2nd Rail を用い，Lustre への

表 1　TSUBAME2.5 の Thin ノードのネットワーク接続を含むハードウェア構成

CPU	Intel Xeon X5670 2 個
メモリ	54 GB
インターコネクト	InfiniBand 4× QDR (40Gbps) 2 本

アクセスで用いる Rail-1 と通信経路を分けるようにした．なお，実装を行う MPI ライブラリとして TSUBAME2.5 でも動作実績のある OpenMPI ver.1.8.2 を選択した．この中の ROMIO に対し，提案する複数の通信方式が選択的に利用できるように実装し，性能評価を行った．I/O に関する試験で用いた TSUBAME2.5 が有する Lustre（Lustre ver. 2.5.27）は 1 つの OSS あたり 13 個の OST が配置されており，本評価では使用した計算ノードと同じ数の OST を使用する構成を用いた．以下，通信・I/O 等の基礎評価を含めた評価結果について報告する．

4.1　プロセス間通信の評価

改変した ROMIO の評価の前に，今回行った ROMIO 内部のデータ通信部分の有効性を検証するために `MPI_Isend`/`MPI_Irecv` 対によるプロセス間通信時間の評価を行った．評価対象としては，ROMIO の既存の通信方式であるランク順に非同期通信関数を発行する手法に加えて，今回の実装で行った 3 種類の通信方式を模擬したものも評価し，比較検討を行った．なお，MPI のランク配置はノード毎にラウンドロビンで割り当てる方法（以下，`RR`）とノード毎に詰めて割り当ててゆく方法（以下，`BK`）の 2 つを確認した．

図 8 に 64 ノードを用いて各ノードに 12 プロセスずつ配置し全体で 768 プロセスによる `MPI_Isend`/`MPI_Irecv` 対による全体全通信を 50 回通信を繰り返した際の平均時間を示す．この計測では，プロセス個々に `MPI_Waitall` による送受信完了までに要した時間を計測しており，送受信を 50 回繰り返した際の平均時間をランクごとに示している．評価した 2 つのランク配置を比べると，`RR` 配置が `BK` 配置よりも時間が短くなる傾向が見られる．いずれの配置においても我々が提案する 3 つの通信手法により時間短縮が実現できていることが確認できる．特に `BK` 配置ではランク順の手法では後に発行されるランクほど通信時間が増加する傾向があり，我々の提案手法が特に有効である可能性が確認できる．また実装した 3 種類の手法の中では，全体的に `nd_rank_shift` の手法が最も良い結果を示していた．

TP-IO ではこのようなプロセス間通信をファイル領域全体を網羅するまで TP-IO サイクルが複数回行われる．TP-IO サイクル数は各プロセスに割り当てられるファイル領域の大きさと TP-IO に用いる CB の大きさで決まってくるが，計算機環境や使用するファイルシステムによっては適切なサイズに設定することで性能向上に繋がる可能性もある．なお，Lustre を用いた TP-IO の場合には，Lustre

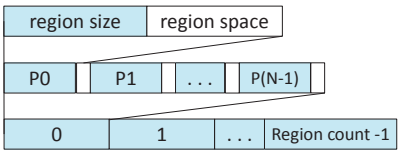


図 9　HPIO ベンチマークにおける派生データ型生成での 3 つのパラメタ（region size, region space, region count）．この例では N 個のプロセス間での派生データ型生成を示しており，例えば `P0` のデータ領域にはランク=0 のプロセスのデータの一部が書き込まれる．

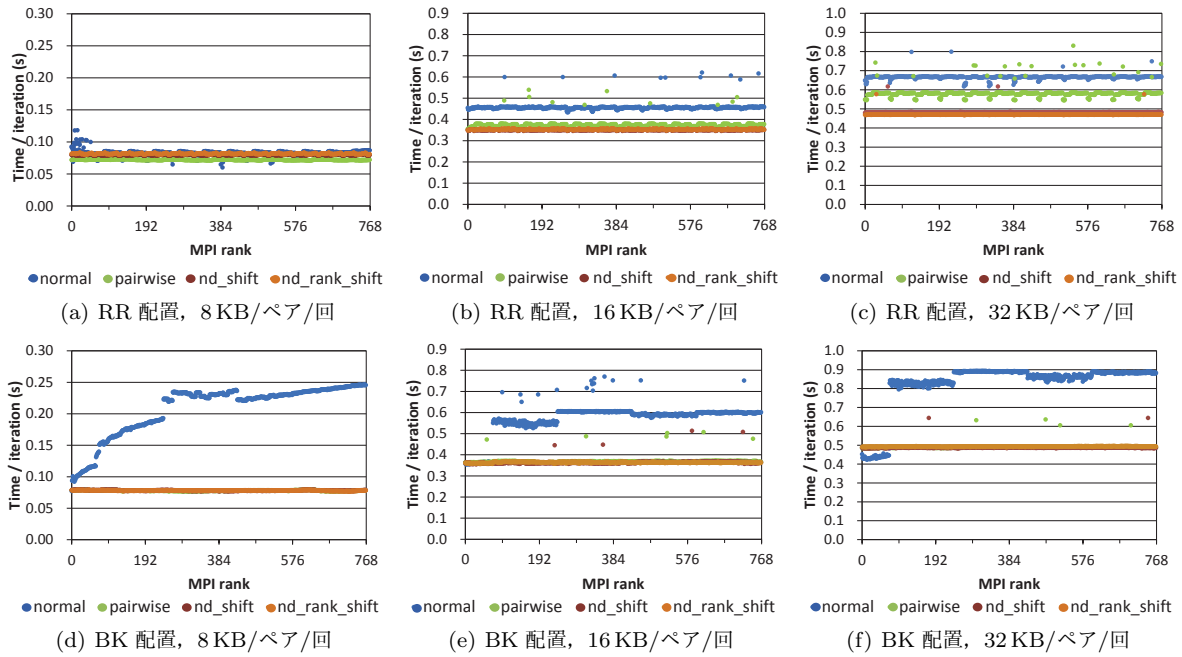
でのストライピングのサイズが CB の大きさに設定されるようになっている．TP-IO における通信時間の割合は，アクセスパターンや CB の大きさ等にもよるが，一般に通信部分が割と大きな割合を占めるので，以上の結果から，ROMIO において，我々が提案するアグリゲータの配置と非同期通信関数の発行順変更機能が性能向上に繋がることが期待できる．

4.2　HPIO ベンチマークによる ROMIO の評価

今回実装した 3 種類の通信方式を有する ROMIO に対し，空白部分を含んだ派生データ型によるアクセスパターンによる集団型書き込みの性能を評価するために，HPIO ベンチマークを用いて `MPI_File_write_all` の性能を評価した．なお，前述の通り MPI ランクのノード間配置ではノード毎に詰めて配置する `BK` 配置での通信削減効果が大きいため，この評価では `BK` 配置に関して評価を行った．

通信試験の場合と同様に，各計算ノードに 12 プロセスずつ起動し，64 ノードで合計 768 プロセスにより，Lustre の 64 個の OST を用いて入出力の性能を評価した．HPIO ベンチマークによる派生データ型生成は**図 9** に示すように region size，region space，region count と呼ばれる 3 つのパラメタを指定する必要があり，本評価では region size を 3,744B，region space を 256B，region count を 48,000 に設定した．その結果，768 プロセス全体で一つのファイルあたり約 140GB（$\sim (3744 + 256) \times 768 \times 48000$ Byte）の書込みを行った．ノードあたりのアグリゲータ数は最大の 12 まで変化させて計測を行い，一つのパラメタセットで 7 回の書込みを行い，その中の最大と最小の I/O スループット値を除く 5 個の計測値の平均値を求め，このような評価を日時を変えて数回計測した中から最大の I/O スループット値を最終的な計測値とした．

この計測で得られた I/O 性能を**図 10** に示す．比較のために，アグリゲータ配置の最適化を行っていない現行 ROMIO を用いた場合（`orig`）とアグリゲータ配置のみ最適化を行った ROMIO を用いた場合（`normal`）の評価も行っている．なお `orig` 及び `normal` 共に，ランク順に非同期関数を発行している．また，現行 ROMIO 以外はノードあたりアグリゲータが 2 個以上のケースでの評価結果を



(a) RR 配置，8 KB/ペア/回　(b) RR 配置，16 KB/ペア/回　(c) RR 配置，32 KB/ペア/回

(d) BK 配置，8 KB/ペア/回　(e) BK 配置，16 KB/ペア/回　(f) BK 配置，32 KB/ペア/回

図 8　MPI_Isend/MPI_Irecv 対による全体全通信時間

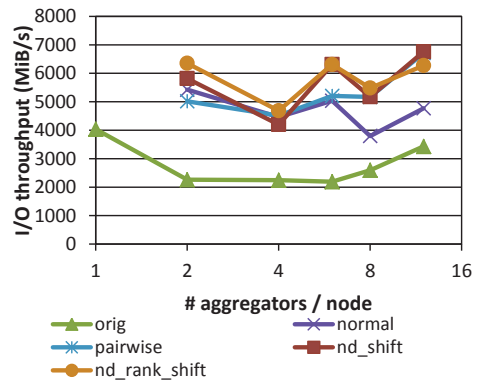


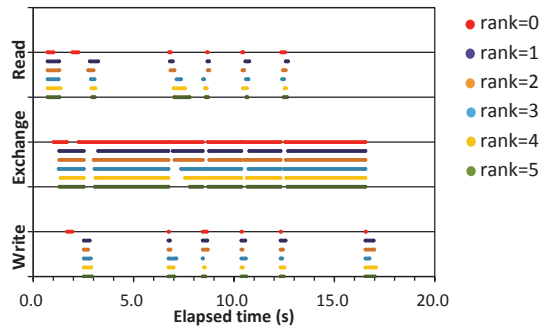
図 10　HPIO ベンチマークによる I/O スループット

示しているが，これはノードあたりアグリゲータを 1 個にした場合にはアグリゲーター配置最適化が無効になるためであり，また通信関数発行順についても現行 ROMIO との違いが無いため評価していない．

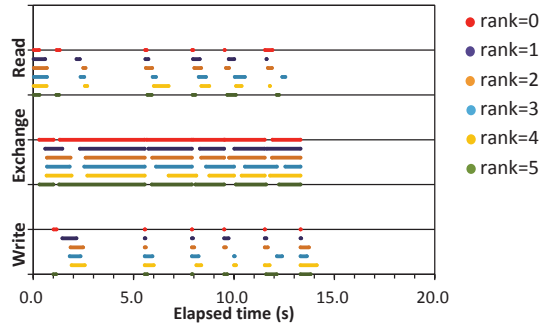
この図から現行 ROMIO（orig）やアグリゲータ配置を最適化したケース（normal）と比較して，今回提案した 3 つの通信方式で更なる性能向上が確認できた．現行 ROMIO での I/O 性能の最大値（約 4GB/s）に対して，通信方式を変更したケースでの最大の I/O 性能が約 7.4GB/s となり，約 67%の性能向上が確認できた．またアグリゲータ配置の最適化のみを行った実装に対しても最大性能に関して約 16%の性能向上が確認できた．3 つの提案手法の中では，特に nd_rank_shift 方式がノードあたりのアグリゲータ数を変えてゆく中で全般的に高い性能を出している傾向が見られるが，この評価結果からでは 3 つの方式のどれが最も適した手法であるかまでは結論付けるのは難しく，これについては今後の課題である．

次に，今回行った TP-IO 内部の通信発行順の最適化に関して，TP-IO 内部でどのような影響があるのかを調べるため，TP-IO の 3 つの主な処理であるファイル読出し（Read），データ交換（Exchange），並びにファイル書き込み（Write）に要する時間を性能評価と同様に HPIO ベンチマークで調査した．この評価のために ROMIO の内部に上記の 3 つの処理の開始時と終了時の時刻を gettimeofday により取得し，プログラム開始時からの経過時間に関して各処理の振舞いを調べたガントチャートを図 11 に示す．この評価では全プロセスをアグリゲータとする構成を用いており，768 プロセス中から，同じノード内にある最初の 6 プロセス（ランクが 0 から 5）の振舞いを，それぞれの ROMIO 実装に関して調査した．図 11 に示したものは TP-IO の最初の 6 回の TP-IO サイクルの処理の流れを示している．なお，ここで示すデータは各処理の実行タイミングを逐一取得して得られたもので，前述の HPIO ベンチマーク評価とは別にデータを取得している．ここからも主要な 3 つの処理の中でデータ通信（Exchange）の時間が他の 2 つの処理に比べて大きな割合を占めているのが分かる．

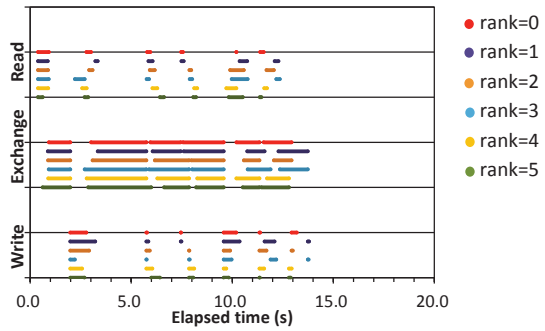
図 11(a) に示す現行 ROMIO の場合に比べ，図 11(b) に示すアグリゲータ配置の最適化のみ行ったケースで通信時間が短縮されているのが確認できる．これについては既に我々のこれまでの研究で報告しているように，各計算ノードにあるアグリゲータ群からの Lustre へのアクセスパターンが最適化されたことにより，通信の混雑が解消されたためである [9], [10]．このアグリゲータ配置最適化を行った実装の上で通信順の最適化を行った図 11(c) から図 11(d) の 3 つのケースでは特に nd_rank_shift 方式での時間短縮効果が大きいことが確認できる．使用するノード数やプロ



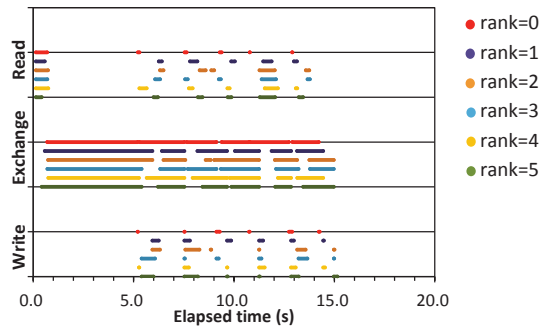
(a) 現行 ROMIO



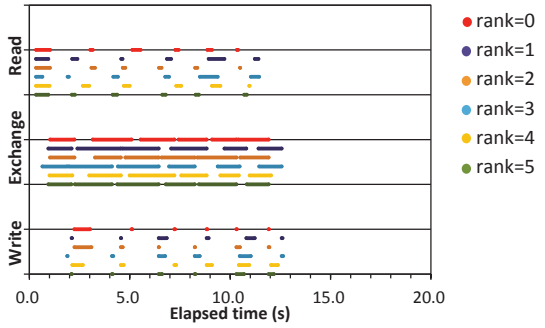
(b) アグリゲータ最適化のみ（`normal`）



(c) `nd_shift` 方式



(d) `pairwise` 方式



(e) `nd_rank_shift` 方式

図 11 集団型書き込みにおける TP-IO 内部の主要な 3 つの処理の振舞い

セス数が増えるにつれて，ノード間通信のコストも大きくなるが，ノード配置を考慮して通信関数の発行順序を変更すれば通信時間の短縮が可能で，その結果，TP-IO による集団型 MPI-IO の高速化に繋がる可能性が期待される．

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 [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，TP-IO 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．但し，プロセス数の増加に伴って，この前処理のコストが増加する問題がある．一方，我々の手法は現行 ROMIO と同様に TP-IO サイクル毎にオフセット・データ長の情報をプロセス間で通信しているため，前処理のコストの増加の可能性は無い．我々の実装では，アグリゲータのレイアウト最適化やアクセスするデータのアグリゲータ間での並び替えにおける通信順の最適化に着目しており，これらにより得られるメリットの方が大きいと考えている．

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．

アグリゲータとなるプロセス間で担当する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アクセス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 がある [15]．これはユーザから見えるファイルレイアウトではなく，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データレイアウトを各アグリゲータで取り入れることでファイルシステム側へのアクセスにおける通信経路の混雑を避けている．同様の最適化は Lustre 向けの ADIO 実装 [12] でも行われている．一方，我々の研究は，ノードあたりに複数のアグリゲータが起動された場合での最適なアグリゲータ配置方法を提案するものであり，これらの研究で提案されているようなファイルアクセスに向けたデータレイアウトの最適化実装の上に立ち，さらにアグリゲータの配置を並列ファイルシステムのデータレイアウトに最適な配置を実現するこ

とでさらなる性能向上を狙っている．

TP-IO による集団型 MPI-IO において，アクセスパターンによってはアグリゲータ間での通信処理とファイル I/O 処理の効率的な実行順を決めることも性能向上に繋がる可能性がある．例えば Liu ら [16] は，アグリゲータ間でのデータ通信のコストが大きいものほど先行して処理を進める実装により集団型読出しの高速化を実現している．アグリゲータのファイル I/O と通信処理がそれぞれ異なるアグリゲータで行われていれば，両者のオーバーラップ処理が可能なので，より通信コストの大きいアグリゲータを先行させて処理を進めることにより，全体の処理時間の短縮を実現している．一方，我々が提案する手法はアグリゲータ間の動作スケジューリングに関する最適化ではなく，個々のアグリゲータのデータ通信に要する時間を短縮させることを目的としている．特に，プロセスのノード間配置を配慮した `nd_shift` 手法や `nd_rank_shift` 手法では通信開始に要する時間が長くなる可能性のあるノード間通信を先行させ，最後にノード内通信を行う最適化を行っており，通信コストを基にアグリゲータの実行順序をスケジュールする彼らの方法とは異なる．また，我々の提案手法では，同じランクのプロセスに通信が集中してしまう問題を回避している．

6. 本稿のまとめ

我々は ROMIO 内部のデータ通信の発行順とプロセス配置に着目し，データ通信コストを低減する通信発行順の最適化実装を行い，TSUBAME2.5 を用いてその有効性を確認した．通信単体の基礎評価においてはノード間のプロセス配置を配慮した通信が有用であることを確認した．特にノード数が増えるにつれてその有用性が大きくなる傾向も確認した．次に HPIO ベンチマークを用いた MPI-IO 性能評価では，通信単体の評価結果と同様にノード間のプロセス配置を配慮した我々の手法の方が単純にランク順に通信を発行するよりも性能が向上することを確認した．ただし提案する 3 つの手法の中での優位性については今回の評価からは明確な結論を出すことは出来ていない．これについては今後の課題として検討してゆく予定である．

現在の実装は単純にランク配置を調べて，I/O 開始前に通信順を静的に決定しているが，関連研究で述べたように，より一般的な I/O アクセスパターンでは，TP-IO の各サイクルでアグリゲータ群の中にはデータ通信が発生しない，あるいは通信コストが小さいものが存在する可能性があるため，通信するデータ長なども配慮した通信発行順にすることで通信コストの低減に繋がる可能性もある．よってこのような配慮を含めた高速化実装の有効性を今後検討したい．

謝辞 本研究の一部は科学研究費（基盤研究（C））課題番号 25330148 の支援を受けております．また本研究を進める中で，東京工業大学学術国際情報センターからは TSUBAME2.5 を利用するにあたり，様々なご支援・ご助言を頂きました．さらに理化学研究所　計算科学研究機構　研究部門　システムソフトウェア研究チームの皆様からは数々のご支援やご助言等を頂きました．この場をお借りして感謝を申し上げます．

参考文献

[1] MPI Forum: `http://www.mpi-forum.org/`.
[2] 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).
[3] MPICH: `http://www.mpich.org/`.
[4] Open MPI: Open Source High Performance Computing, `http://www.open-mpi.org/`.
[5] The National Center for Supercomputing Applications: `http://hdf.ncsa.uiuc.edu/HDF5/`.
[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] Lustre: `http://lustre.org/`.
[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] 辻田祐一，堀敦史，石川裕: Lustre のストライピングアクセスパターンに配慮した集団型 MPI-IO の性能向上に向けた試み, 情報処理学会研究報告, 2015-HPC-149, 3 (2015).
[11] 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).
[12] Lustre: Lustre ADIO collective write driver, Technical report, Lustre (2008).
[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] 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).
[16] Liu, J., Chen, Y. and Zhuang, Y.: Hierarchical I/O Scheduling for Collective I/O, *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, IEEE, pp. 211–218 (2013).