Vol.2011-HPC-130 No.32
2011/7/28

情報処理学会研究報告
IPSJ SIG Technical Report

Two-Phase I/O の高速化に関する一検討

六 車 英 峰[†1,†2] 辻 田 祐 一[†3]
堀 敦 史[†4] 並木 美太郎[†5]

高性能な並列 I/O を提供している ROMIO の集団型 I/O では，Two-Phase I/O と呼ばれる最適化手法が用いられており，ファイルアクセスの回数を最小限にすることで高速化している．しかし，Two-Phase I/O では通信を行うため，プロセス数の増加等による通信コストの増加によって性能が低下する．また，近年，コア数の増加によってノード内の計算プロセスが増加しており，メモリ不足から Two-Phase I/O の性能を十分出せない可能性がある．本研究では，Two-Phase I/O の通信とファイルアクセスを並列に実行することで通信コストを隠蔽し高速化する検討を行う．マルチスレッドと非同期 I/O を用いた 2 つの方法で実装を行い，その性能を評価した．その結果、マルチスレッドによる実装が最も良い性能を示し，また，オリジナル Two-Phase I/O の最高性能を上回った上でメモリ使用量の低減が可能であることが確認できた．一方，非同期処理による実装ではメモリ使用量が厳しく制限される状況において性能を向上できる可能性があることが確認できた．

Optimization of Two-Phase I/O for High Throughput Parallel I/O

Hidetaka Muguruma,[†1,†2] Yuichi Tsujita,[†3]
Atsushi Hori[†4] and Mitaro Namiki [†5]

MPI-IO provides fruitful APIs for parallel I/O in MPI programming. One of the well-known MPI-IO libraries named ROMIO adopted several optimization techniques such as Two-Phase I/O. Two-Phase I/O consists of I/O and communication phases to improve performance in accessing non-contigous data patterns in a collective manner. However, increase in the number of MPI process in turn leads to increase in the communication times, and as a result its performance is degraded. Furthermore, we may have a huge scale of data with an increase in the number of CPU cores. Such situation required more memory resources for MPI applications, thus we can not utilize enough memory for Two-Phase I/O to improve performance. Since two-phase I/O has a room to overlap of those operations, we have implemented two schemes; multithreaded and asynchronous operations. We have evaluated the two schemes on PC cluster systems. Through this evaluation, the former scheme outperformed the latter one and reduced the amount of used memory size. On the other hand, the latter one has a little performance advantage compared with the original two-phase scheme when a buffer size for Two-Phase I/O was quite small.

1. はじめに

計算機の演算性能の高速化によって，計算アプリケーションはより精緻で大規模な計算が実現可能となっている．一方で, アプリケーションの大規模化は扱うデータのサイズを増加させるため，I/O 性能の向上は重要な課題となっている．

MPI フォーラム[8] では高性能な並列 I/O のためのインタフェースである MPI-IO を策定している．この MPI-IO の最もよく知られた実装のひとつである ROMIO[9] において，集団型 I/O を効率良く実行するために，Two-Phase I/O[10]（以下，TP I/O）と呼ばれる最適化が用いられている．TP I/O は，通信によるデータ並べ替えとファイルアクセスの 2 つの操作によって実現され，複数のプロセスによる多数の小さなファイルアクセス要求をまとまった大きなファイルアクセスにすることでファイルアクセスの回数を減らし，高速化を実現している．しかしながら，TP I/O では通信を行うため，プロセス数やデータ量の増加などにより通信コストが増すと性能が低下する．また，TP I/O による最適化では，データ並べ替えのために Collective Buffer（以下，CB）と呼ばれる一時的なバッファを必要とし，CB が十分なサイズでなければ，TP I/O は複数回に分けて実行され性能が低下する．そのため，TP I/O の性能を最大化するためには，TP I/O が一回で実行を完了するために十分なメモリ量を CB として確保する必要がある．しかし，近年のマルチコア・メニーコア化によりノード内のプロセスやデータ量は増加しており，メモリ消費量が増加する傾向がある．

†1 近畿大学システム工学研究科
Kinki University
†2 独立行政法人科学技術振興機構 CREST
JST CREST
†3 近畿大学工学部
Faculty of Engineering, Kinki University
†4 理化学研究所 計算科学研究機構
AICS, RIKEN
†5 東京農工大学
Tokyo University of Agriculture and Technology

© 2011 Information Processing Society of Japan

また，プロセスやデータ量の増加は性能を最大化するために必要な CB サイズも増加させるため，TP I/O の最大性能を出すために必要なサイズのメモリ領域が確保できない可能性がある．

本研究では，TP I/O の通信とファイルアクセスを並列に実行することで通信のコストを隠蔽し，集団型 I/O を高速化する手法を提案する．本提案手法において，TP I/O を並列に実行するためには，TP I/O を複数回に分けて実行させる必要がある．従って，メモリ使用量を節約した上で性能を向上させられる可能性がある．本提案手法の並列化の方法として，これまでの研究で実装した[11] 1)Pthread[6] を用いたマルチスレッド処理と 2) ファイルシステムの非同期 I/O を用いた方法の 2 つの方法で実装し，その性能を実験した．

今回は Linux が動作する PC クラスタ環境で並列ファイルシステムの一つである PVFS2[3] を構築し、上記の 2 つの実装の性能評価を行った。その結果、マルチスレッドによる実装はオリジナルの TP I/O よりも少ないメモリ使用量でオリジナルの最大性能を超えることを確認できた．一方で，非同期処理による実装はメモリ使用量が厳しく制限される環境において性能を向上させられる可能性があることが確認できた．

以下，第 2 章で Two-Phae I/O について述べ，第 3 章では Two-Phase I/O 高速化の提案手法について述べる．第 4 章において今回行った性能評価について述べる．次に第 5 章において，関連研究について述べた後，第 6 章において本論文のまとめと今後の課題について述べる．

2. Two-Phase I/O

計算アプリケーションにおいて，ファイル上の多次元のデータ構造のデータを部分行列へ分割して各プロセスへ割り当てる場合，一般的に各プロセスのファイルアクセスは不連続なものとなる．不連続なアクセスパターンによる I/O は，多量の小さなファイルアクセスとなるため I/O の性能は低下する．TP I/O では，I/O の際にプロセス間でデータ交換を行うことで，各プロセスのファイルアクセス要求をまとめて，できる限り連続なファイルアクセス要求へ変えており，それによりファイルアクセスの回数を減らして高速な I/O 性能を実現している．

TP I/O は MPI-IO の実装の一つである ROMIO において実装されており，集団型 I/O を呼び出した際に実行することが可能である．TP I/O を呼び出した MPI プロセスのうち，実際にファイルアクセスを行うプロセスをアグリゲータと呼ぶ．標準では MPI プロセスを呼び出した全てのプロセスがアグリゲータとなるが，アグリゲータの数はファイルオープン時にヒントを与えることで変更可能である．

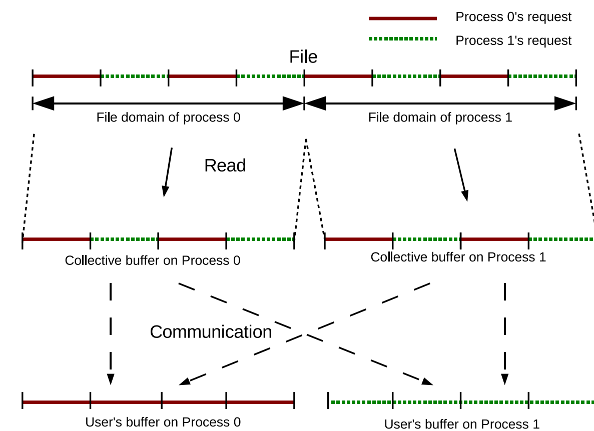

図 1　TP I/O による集団型読み込み操作の様子（2 プロセス）

図 1 に 2 プロセスによる TP I/O による集団型読み込み処理の動作を示す。 図では全てのプロセスが I/O アグリゲータである．アグリゲータにはアクセスするファイル領域が分割して割り当てられる．まず始めに，アグリゲータが割り当てられたファイル領域から Collective Buffer（CB）へデータを読み込む．その後に，通信によってデータが各プロセスへ配布される．

集団型書き込みの場合では，この逆の手順で実行される．ただし，書き込むデータに空白がある場合は，ファイル上のその部分のデータを壊さないために，データを事前に読み込む操作を行う．

ROMIO の実装においては CB サイズを任意に決定することが出来るが，アグリゲータに割り当てられた領域が CB サイズより大きく，一回の TP I/O の実行では全てのデータに対する処理が完了できない場合は，全てのデータに対する処理を完了するまで TP I/O は繰り返し実行される．TP I/O が複数回実行される場合には，ファイルアクセスとデータ並べ替えの回数が増加するため，ファイルアクセスと通信のオーバーヘッドの増加により性能は低下する．従って，TP I/O の性能を最大化するためには，TP I/O が一回で完了するのに十分な大きさの CB サイズを設定しなければならない．

© 2011 Information Processing Society of Japan

3. Two-Phase I/O 高速化の提案手法

本章では、TP I/O の高速化の提案手法について概説する．その後に，我々が実装したマルチスレッドによる並列化 TP I/O と非同期 I/O による並列化 TP I/O の説明を行う．

TP I/O が複数回実行される場合，ある回の TP I/O は，その前の TP I/O の完了を待って実行されるが，それらはそれぞれ別のデータ領域に対する処理であるので，独立に実行が可能である．そこで我々は，並列処理の技法であるパイプライン処理を用いることで，TP I/O のファイルアクセスとデータ並べ替え処理を並列実行し高速化する検討を行った．

図 2 は，集団型ファイル読み出しを実行した際に，(a)TP I/O と (b) 並列化した TP I/O において，ファイル読み込みとデータ並べ替えが処理されるタイミングを示している．TP request（TP 要求）とは，TP I/O を一回実行する要求である．図では，TP 要求は，ファイル読み込み処理とデータ並べ替え処理をしている場合とで区別して表現している．

オリジナルの TP I/O(a) では 2 つの処理が交互になされるのに対して並列化 TP I/O(b) では 2 つの処理が同時に実行されていることがわかる．図 2(b) より，並列化した TP I/O の全体の実行時間は,（最初の読み出しの時間）＋（オーバーラップの時間）＋（最後の並べ替えの時間）である．オーバーラップする時間は，TP 要求の数が増すことで増加し，TP 要求の数は CB サイズを小さくすることで増加する．すなわち，並列化 TP I/O では CB サイズを小さくすることで隠蔽する処理の割合が増加し，性能を向上することが出来るため，メモリを節約した上で性能を向上させることが期待できる．

この図では，データの並べ替えとファイルアクセスに要する時間が同じであるが，実際には通信やディスクアクセスの性能，あるいはアクセスパターンやプロセス数等の影響によって，ファイルアクセスとデータ並べ替えの時間のバランスは変わる．データ並べ替えの時間に対してファイル読み出しの時間が大きい場合では，オーバーラップする時間において，データ並べ替えの処理はファイル読み出し処理に隠蔽されていると考えられる．従って，この場合，並べ替えに要する時間のうちの（TP 要求の数 − 1）／（TP 要求の数）が隠蔽されると考えられる．

また，並列化による実行時間の短縮は，ファイル読み出しとデータ並べ替えの時間の比が 1:1 の場合が最も効果的に短縮されると考えられる．処理時間が同じであれば，一方の処理を待つ無駄な時間が無いためである．

しかしながら，TP 要求が増加することによってファイルアクセスおよびデータ並べ替えのための通信の回数が増すため，オーバーヘッドが増加する．従って，回数の増加により，

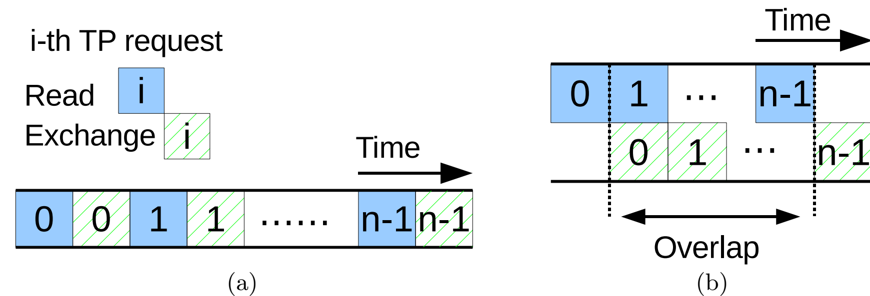

図 2　TP I/O の処理タイムライン (a) オリジナル TP(b) 並列化 TP

処理の隠蔽が進むことによる性能向上をオーバーヘッドの増加による性能低下が打ち消してしまう可能性が考えられる．

3.1 マルチスレッド版 TP I/O の実現手法

マルチスレッドによる TP I/O の並列化実装では，TP I/O におけるファイルアクセスとデータ並べ替えをそれぞれ異なるスレッド上で実行することで並列化している．今回実装したマルチスレッドによる並列化 TP I/O の動作を**図 3** に示す。

本実装では TP I/O のファイル読み出しを専用のスレッド（Read スレッド）で実行し，データ並べ替えの処理はメインスレッド上で実行する．また，スレッド間の TP 要求の受渡しは，それぞれのスレッドに対してキューを持たせることで実現している．本実装の動作は，まず始めに，メインスレッドが 2 つの TP 要求を作成し，それを Read スレッドのキューへ入れる．その後はメインスレッドのキュー (Exchange キュー) を見て要求があればそれを処理し，1 つを処理し終える度に新たに 1 つ TP 要求を作成して Read スレッドへ入れる．これを全てのデータを処理し終えるまで繰り返す．Read スレッドではキューに TP 要求があればそれを取り出してファイル読み出し処理を行い，Exchange キューへ渡す．本実装では，2 つの CB を確保し，ダブルバッファリングと同じ仕組みで動作する．ただし，TP I/O の回数が 1 回の場合では，並列に実行する TP 要求が無いため，その場合は，CB は 1 つしか確保しない．

3.2 非同期版 TP I/O の実現手法

ファイルシステムによっては、非同期型 API が整備されている．今回我々は，これを利用した並列化 TP I/O の実装も行った．非同期 I/O 版でもやはり 2 つの CB を用意し，ダブルバッファリングによってデータ並べ替えとファイル読み出しをオーバーラップさせる．

© 2011 Information Processing Society of Japan

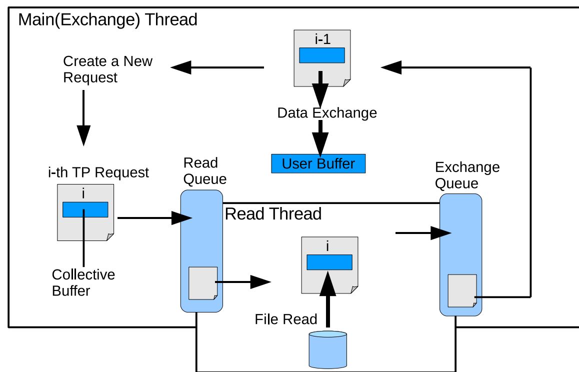

図 3　マルチスレッド版 Two-Phase I/O の実装

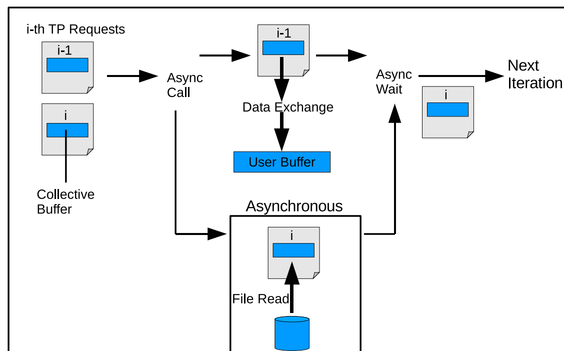

図 4　非同期版 Two-Phase I/O の実装機構

また，マルチスレッド版と同じく，TP I/O が 1 回しか実行されない場合には，CB は 1 つしか確保しない．

図 4 は，非同期 I/O を用いた TP I/O の動作を示した図である．非同期版の実行は，まず始めに TP 要求を作成し，次に非同期のファイル読み出しを実行する．その間にデータ並べ替えの処理を行う．データ並べ替え処理を行う TP 要求は，前回の実行でファイル読み出しをされた TP 要求である．データ並べ替えが終わると，非同期ファイル読み出しの完了を待ち，その後に次の TP I/O の実行に移る．これを全てのデータに対する処理を終えるまで繰り返し実行する．

表 1　使用した各 PC クラスタのノードの仕様

	構成 1	構成 2
CPU	1 ×Intel Xeon E5530 2.4 GHz	1 × Intel Pentium-D 3.2 GHz
メモリ	8 GB	1.5 GB
NIC	Broadcom BCM 5716 × 2	Broadcom BCM 5721
ネットワークスイッチ	hp 2824al（MPI 通信用） 3Com SuperStack3 3824（PVFS2 データ通信用）	
OS	Fedora12 (kernel 2.6.34)	Fedora12 (kernel 2.6.32)
MPI library	MPICH ver. 1.2.1p1 ベースの改造実装	
並列ファイルシステム		PVFS ver. 2.8.2

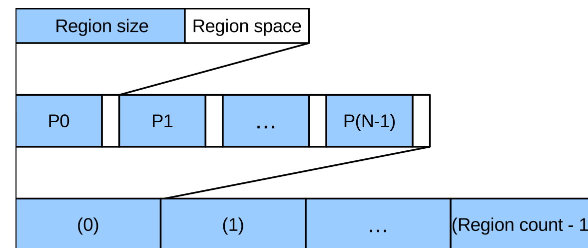

図 5　HPIO ベンチマークにおける各プロセスのアクセス領域配置

4. 性 能 評 価

今回実装した TP I/O の性能評価を行った．性能評価は 2 種類のノード（各々9 ノード）で構成された 18 ノードの PC クラスタで行った。**表 1** に，使用した各 PC クラスタのノードの仕様を示す。 各ノードは 2 つのギガビットイーサネットスイッチにより接続され，それぞれ MPI 通信と PVFS2[3] のデータ通信に利用している．ファイルシステムに利用した PVFS2 は，8 I/O ノード，1 メタデータノードで構成した．また，性能評価には HPIO ベンチマーク[4] を用いた．

HPIO ベンチマークにおいて，各プロセスがアクセスするファイル上のデータの配置を**図 5** に示す．ファイル上のデータ配置は，プロセスがアクセスを要求する領域である Region size と要求しない領域である Region space が隣接し，それがプロセス数並ぶ．そして，そのまとまりが Region count 数配置されている．

© 2011 Information Processing Society of Japan

各プロセスがアクセスするデータのサイズは (Region size)×(Region count) となるので，全てのプロセスが要求するデータの総量は，それにプロセス数を掛けたものである．しかし，TP I/O では空白領域も含めてまとめてファイルからデータを読み込むので，アクセスするデータの総データ量は，(Region size + Region space)×(Region count)-(Region space) である．(最後の空白は読み込まない．)

4.1 並列化実装の動作検証

並列化 TP I/O が実際に処理を並列に実行できていることの検証を行った．今回，マルチスレッドを用いた方法と非同期 I/O を用いた方法の二つの方法で実装を行ったが，非同期 I/O を用いた実装では，ファイル読み込みの時間をうまく計測することが出来なかった．そのため，マルチスレッドを用いた実装についてのみ検証を行う．

実験では，HPIO ベンチマークを 8 プロセスで実行した．アクセスパターンのパラメタは，Region size=(64KiB-128B)，Region space=128B，Region count=1024 である．すなわち，空白 128B を含む 64KiB のデータを各プロセス 1024 個ならべた．合計 512MiB のファイルである．全てのノードをアグリゲータとしたので，各プロセス 64MiB のファイルへアクセスする．従って，CB サイズが 64MiB の際に，TP I/O は 1 回で完了する．実験では，CB サイズを 1MiB から 64MiB まで変化させることで TP I/O 回数を 64 回から 1 回まで変化させた．

図 6 は，マルチスレッド版 TP I/O が実際にどれだけの時間を短縮しているかを示している．マルチスレッド版 TP I/O の実行時間 (T-TP) とその内部のファイル読み込み (Read) とデータ並べ替え (exch) の時間，及びファイル読み込みとデータ並べ替えの合計 (Read+Exch) の時間（以下，合計時間）を表示している．合計時間は，並列に実行しなかった場合に要したであろう実行時間を示している．Ratio は，合計時間に対するマルチスレッド化 TP I/O の実行時間の割合であり，図中の右側の縦軸は Ratio を示すための軸である．図を見ると，TP I/O が複数回実行されている場合に，合計時間よりも実行時間が短縮されていることがわかる．一方で TP I/O が一回の場合では全く性能が向上していない．これは，1 回の場合では並列に実行できる TP 要求がないためである．TP I/O の回数 1 回から 4 回までの実行時間を見ると，ファイル読み込みとデータ並べ替え，及びその合計時間はそれぞれ増加しており，性能は低下している．これは，TP I/O の回数が増すことでオーバーヘッドが増加するためと考えられる．一方で実行時間は 2 回の時に最短となり，4 回の時でも 1 回の時より短くなっており，それぞれ，10 %と 6 %時間が短縮していた．これは，TP I/O の回数が増すことで，オーバーラップする二つの処理の時間が増すためであると考えられる．

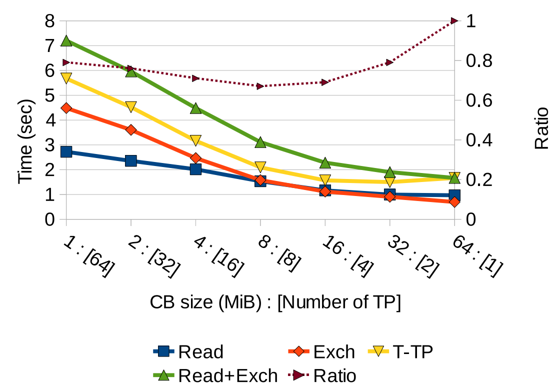

図 6 マルチスレッド版 TP I/O の実行時間の内訳と各処理の合計時間との比

実行時間の各処理の合計時間に対する比 (Ratio) を見ると，8 回の時が最も小さい．これは，回数の増加によってオーバーラップする時間が増したことと，読み出しと並べ替えに要する時間であるために，並列処理の効率が高かったためである．しかしながら，回数が 8 回以上では 1 回の時よりも実行時間は増している．これは，オーバーヘッドの増加による性能低下の影響により性能向上の効果が打ち消されてしまうためである．したがって，TP I/O の回数の増加によって単純に性能が向上しないことが分かる．しかしながら，TP I/O が 1 回の場合より性能が向上することがあることから，並列化実装においては，条件によって性能を向上させられる可能性があり，その際，性能を最大化する最適な TP I/O の回数があることが考えられる．

4.2 性能比較

本提案手法である並列化 TP の 2 つの実装とオリジナルの TP I/O の性能比較を行った．HPIO ベンチマークへ与えたパラメタは図 6 の場合と同じであり，ベンチマークを実行した 8 プロセス全てがアグリゲータとなり，それぞれ 64MiB のファイルアクセスを行う．また，図 6 の場合と同じく，CB サイズを 1MiB から 64MiB へと変化させることで，TP I/O を 64 回から 1 回まで変化させて性能を測定した．

図 7 は，TP I/O の回数を変えた場合のオリジナルの TP I/O(Orig) と非同期版 (Async)，およびマルチスレッド版 (Thread) の性能を比較したものである．

(a) は TP I/O に指定した CB サイズおよび TP I/O の回数を横軸にとって比較してい

© 2011 Information Processing Society of Japan

る．図を見ると，TP I/O が 1 回の場合では，並列実装版はオリジナルとほとんど性能が変わらない．これは，並列に実行できる TP 要求が無いためであり，TP I/O が逐次に実行されているためであると考えられる．その他の TP I/O の回数についてみると，マルチスレッド版は全ての場合でオリジナルより勝っている．一方，非同期版では，TP I/O 回数 2 回から 8 回までにおいて，わずかに性能が向上しているように思われる．また，TP I/O 回数が 16 回，32 回の場合ではマルチスレッド版と同程度の性能を記録している．TP I/O が 64 回の場合には他の実装と比べて突出して性能が向上している．

並列化実装では，I/O と並べ替えの 2 つの処理を並列実行するためにダブルバッファリングを行っている．そのため，並列化実装では CB を 2 つ持っている．(b) はメモリ使用量に着目し，横軸を合計の CB サイズで比較した図である．この図は，(a) における 2 つの並列化実装の計測結果を右へ 1 目盛ずらしたものである．ただし，並列化実装において，TP I/O が 1 回の場合は CB は 1 つしか確保していない．そこで，(b) においては，CB=32MiB×2 のデータとの混同を避けるため，並列化版の TP 回数が 1 回の場合のデータはプロットしていない．また，同じ合計サイズで比較する場合，TP I/O 回数はオリジナルと並列化実装とで異なる．ここでは，メモリ使用量に着目するため省略している．

(b) を見ると，スレッド版では 32MiB の場合と 64MiB の場合では，オリジナルの最高性能（CB サイズ 64MiB，TP 回数が 1 回）に対して，それぞれ 18 %，23 %性能が向上している．32MB の場合では，メモリ使用量が半分であるので，マルチスレッド版ではメモリ節約と性能の向上が同時に実現されていることがわかる．

一方で，非同期版では，合計 CB サイズが 2MiB 以外の場合ではオリジナルとほとんど性能が変わらない．しかしながら，合計の CB サイズが 2MiB の場合 ((a) において TP I/O の回数が 64 回の場合) では，他の実装と比較して性能が大きく向上している．合計 CB サイズが 2MiB の際に他の実装よりも性能が向上した理由については，現在のところ判明していないが，この結果は CB サイズが小さい場合においては，非同期 I/O を用いた並列化によって性能を向上させられる可能性があることを示している．

5. 関 連 研 究

入出力をスレッドによってバックグラウンドで行わせる実装はこれまでにも数多くなされている[5)–7)]．しかしながら，それらは計算と I/O をオーバーラップさせるものであり，集団型 I/O 内部で実行される TP I/O のデータ並べ替えのための通信とファイルアクセスをオーバーラップさせる我々の方法とは異なる．View-based I/O[1)] は，不連続なファイルアクセスを行う集団型並列入出力の最適化の手法である．View-base I/O は，TP I/O を改良したもので，TP I/O がアクセス時にファイル領域の割り当てを動的に行い，データのオフセットとサイズを I/O アグリゲータへ送信するのに対して，ファイル領域の割り当てを静的に行い，また，アクセス時のオフセットと長さの通信を不要にしたことで，通信量や計算量を減らし，I/O 性能を向上させている．また，実装内部にキャッシュバッファを用意することでも性能を向上している．2) では View-Based I/O に GPFS のライブラリを用いてバックグラウンドによる書き込みとデータを先読みする機能を加えた実装が提案されている．この実装ではスレッドが用いられており，I/O がデータ並べ替え処理とは別スレッドで実行されている．I/O とデータ通信のオーバーラップという点では，本研究はこれに近い．しかしながら，GPFS の機能に依存した実装であることと，CB サイズによる性能の変化に着目していない点で本研究とは異なる点である．本研究では，マルチスレッドを用いた場合ではファイルシステムに依存せず，非同期 I/O を用いる場合でも，非同期 I/O を提供しているファイルシステム上で実行可能であり，様々なファイルシステム上で実行可能である．

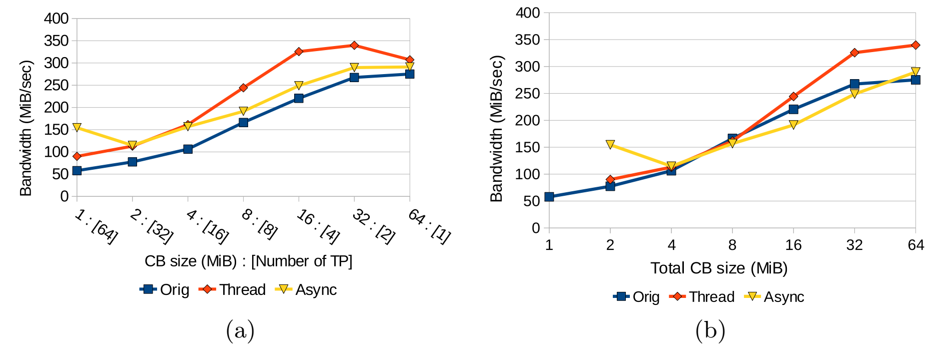

図 7 (a)CB サイズで比較した場合と (b) 合計の CB サイズで比較した場合

6. まとめと今後の課題

本稿では，TP I/O の並列化を提案し，ファイル読み込みの TP I/O に対してマルチスレッドによる実装と非同期 I/O による実装を行い，その動作検証と実装の違いによる性能を比較した．動作検証では，マルチスレッド版の実装を用いて内部の各スレッドの実行時間を確認することで，実際に並列化されていることを確認した．また，並列化実装においては，オリジナルの TP よりも性能を向上させられる可能性があり，その場合には性能を最大化す

© 2011 Information Processing Society of Japan

るために適切な TP I/O の回数決定する必要があることがわかった．また，マルチスレッド版においては，メモリ使用量を減らした上で性能を向上可能であることが確認でき，本提案手法が有用であることを示した．一方で，非同期版では，CB サイズが小さい場合において性能を出すことが確認できた．この理由については現在調査中であるが，この結果は，メモリ使用量が厳しく制限されている環境において性能を向上させられる可能性を示している．

今後，更に並列化 TP I/O に関する調査を進めると共に，現在取り組んでいる TP I/O の書き込みに対する並列化実装を完成させるつもりである．

謝辞　本研究を進めるにあたり，東京大学情報基盤センター長の石川 裕　教授から様々な有用なご意見を頂きました．なお，本研究の一部は科学研究費 若手研究（B）課題番号 21700063 ならびに 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」の支援を受けております．

参 考 文 献

1) Francisco JavierGarcía Blas, Florin Isaila, DavidE. Singh, and Jesús Carretero. View-based collective i/o for mpi-io. In *CCGRID*, pages 409–416, 2008.
2) JavierGarcia Blas, Florin Isaila, Jesus Carretero, David Singh, and Felix Garcia-Carballeira. Implementation and evaluation of file write-back and prefetching for MPI-IO over GPFS. *International Journal of High Performance Computing Applications*, 24:78–92, 2010.
3) PhilipH. Carns, WalterB. Ligon III, RobertB. Ross, and Rajeev Thakur. PVFS: A parallel file system for Linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327. USENIX Association, October 2000.
4) Avery Ching, Aloc Choudhary, Wei keng Liao, Lee Ward, and Neil Pundit. Evaluating I/O characteristics and methods for storing structured scientific data. In *Proceedings 20th IEEE International Parallel and Distributed Processing Symposium*, page49. IEEE Computer Society, April 2006.
5) Phillip Dickens and Rajeev Thakur. Improving collective I/O performance using threads. In *Proceedings of the Joint International Parallel Processing Symposium and IEEE Symposium on Parallel and Distributed Processing*, pages 38–45, 1999.
6) Instituteof Electrical and Electronic Engineers. Information Technology – Portable Operating Systems Interface – Part 1: System Application Program Interface (API) – Amendment 2: Threads Extensions [C Languages]. 1995.
7) Xiaosong Ma, Marianne Winslett, Jonghyun Lee, and Shengke Yu. Improving MPI-IO output performance with active buffering plus threads. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS'03)*, page 68b. IEEE Computer Society, April 2003.
8) MPI Forum. `http://www.mpi-forum.org/`.
9) Rajeev Thakur, William Gropp, and Ewing Lusk. On implementing MPI-IO portably and with high performance. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, pages 23–32, 1999.
10) Rajeev Thakur, William Gropp, and Ewing Lusk. Optimizing noncontiguous accesses in MPI-IO. *Parallel Computing*, 28(1):83–105, 2002.
11) 六車 英峰, 辻田 祐一. マルチスレッドによる Two-Phase I/O の高速化の検討. 研究報告「ハイパフォーマンスコンピューティング (HPC)」, No.2010-HPC-127, pp.1-6, 情報処理学会 (2010).

© 2011 Information Processing Society of Japan