Vol.2011-ARC-197 No.35
Vol.2011-HPC-132 No.35
2011/11/29

パイプライン型 Two-Phase I/O の Lustre における性能評価

六車英峰[†1,†5] 吉永一美[†1,†5] 辻田祐一[†1,†5]
堀敦史[†2,†5] 並木美太郎[†3,†5]
石川裕[†2,†4]

近年，並列計算が扱うデータの大規模化により，I/O は主要なボトルネックとなっており，さらなる高速化が望まれている．また近年のコア数の増加によりメモリ消費が大きくなり安易に大規模なメモリ領域を確保できない．Two-Phase I/O は MPI-IO 実装の 1 つである ROMIO における高性能な集団型 I/O を実現する最適化であるが，性能向上のために確保するメモリ量を大きくする必要がある．また，I/O コストのノード間でのばらつきが性能に影響する問題もある．我々がこれまでの研究において提案した Two-Phase I/O の改良案は，PVFS2 が利用できる小規模な PC クラスタにおいては，条件によってはメモリ利用量を低減した上で性能を向上することが可能であることを示した．しかしながら，より大規模な環境での本実装の評価を行う必要があると考え，本稿では，東京大学情報基盤センター所有の T2K オープンスパコンを用い，Lustre ファイルシステムに対する性能評価を行った．その結果，今回の環境においても条件によってはオリジナルよりもメモリ利用量を低減した上で性能向上を実現できることが確認できた．

Performance Evaluaton of Pipelined Two-Phase I/O on a Lustre File System

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

Since recent parallel computing applications generate huge scale of data, I/O operation is one of the bottlenecks in performance improvement. Furthermore, available memory size per CPU core is shrinking with an increase in the number of CPU cores. MPI-IO is available as a powerful parallel I/O interface in parallel computations using MPI. ROMIO is one of the well-known MPI-IO implementations, and it adopts an effective implementation scheme named Two-Phase I/O. However, I/O performance improvement requires a large scale of memory resources. Besides unbalanced I/O operation times among MPI processes might bring further performance degradation. We have already proposed a pipelined scheme in the Two-Phase I/O protocol and proven its effectiveness on a small PC cluster with a PVFS2. In order to evaluate its effectiveness on a large scale system, we have evaluated it on a big machine, T2K Open Supercomputer at the University of Tokyo by using a Lustre file system. Here we have evaluated I/O throughput and found its performance advantages relative to the original Two-Phase I/O in parallel with reducing memory resource utilization in some conditions.

1. はじめに

近年，計算機の演算能力の向上に伴い，アプリケーションが大規模化しており，それに伴ってデータサイズも大規模化が進んでいる．一方で，I/O の性能向上は計算機の演算能力の向上に追い付いていない現状があり，I/O がアプリケーションの性能の主要なボトルネックとなっている．I/O の性能向上についてはこれまでも様々な研究が行われている．並列ファイルシステムは複数のノードにファイルを分散することで高性能でスケーラブルなファイルシステムを実現するものであり，代表的なファイルシステムに PVFS2[1)]，Lustre[2)] や GPFS[3)] などが挙げられる．一方クライアント側の高速化には，並列プログラムの複数のプロセスの I/O をまとめて行う集団型 I/O と呼ばれる手法が用いられる．MPI Forum[4)] が策定した MPI-2 において MPI-IO と呼ばれる高性能な並列 I/O インタフェースが定義されており，その MPI-IO において集団型 I/O はサポートされている．MPI-IO の実装の一つである ROMIO[5)] において，集団型 I/O の最適化として，Two-Phase I/O[6)](以下，TP

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

Vol.2011-ARC-197 No.35
Vol.2011-HPC-132 No.35
2011/11/29

I/O と呼ぶ) が用いられている．TP I/O は，複数ノードによる不連続なアクセスパターンの I/O を高速に実行することが出来るが，性能向上のためには最適化のために確保するメモリのサイズを大きくする必要がある．しかし，近年，ノード内のコア数の増加によって，メモリ消費が増加し，大きなメモリサイズを確保することが難しくなっている．また，TP I/O は I/O と通信を複数回実行することがあり，その際にノード間の I/O コストのばらつきが性能に影響することが考えられる．

我々はこれまでの研究で，この TP I/O 内部の通信と I/O をオーバーラップさせことで TP I/O の性能を向上させる方法を提案している[7]．本稿ではこの提案手法をパイプライン型 TP I/O と呼ぶ．本提案手法はメモリを節約した上で性能向上が期待でき，また，ノード間の I/O コストのばらつきによるの影響を緩和させられる可能性がある．これまでの研究では小規模な 9 ノード構成クラスタを利用して PVFS2 ファイルシステムに対する性能評価を行い，条件によってはメモリ消費量を節約した上で性能向上することが可能であることを示した．しかしながら, これまでの評価は小規模なクラスタを利用しているため，より大規模な環境での評価が出来なかった．また，これまでは PVFS2 ファイルシステムを利用したが，大型の並列計算機では，よりスケーラブルな Lustre や GPFS が利用されており，これらのファイルシステム上での性能評価が必要であった．

そこで今回は，東京大学情報基盤センター所有の T2K オープンスパコン[8] 上で最大 64 ノード上で性能評価を行った．また, ファイルシステムに Lustre ファイルシステムを利用して性能評価を行った．また，本提案手法によるノード間の I/O コストのばらつきによるの影響の緩和についても調査を行った．

以下，第 1 章で Two-Phase I/O について述べ，次に第 2 章でパイプライン型 TP I/O について述べ，そして第 3 章で Lustre ファイルシステムについて説明を行う．第 4 章において性能評価について報告し，次に第 5 章で関連研究について述べた後に，第 6 章でまとめと今後の課題について述べる．

2. Two-Phase I/O

並列アプリケーションでは，一つのファイルに対して複数のプロセスがそれぞれ不連続なファイルアクセスを行う場合がある．例えば，ファイル上の多次元データ構造のデータを部分行列へ分割して複数の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．このような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が発生するため I/O の性能は低下する．TP I/O はこのような不連続なアクセスパターンによる集団型 I/O を高速化する最適化手法である．

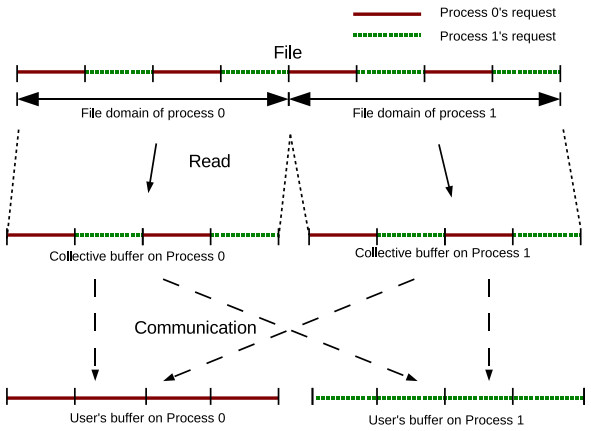


図 1　TP I/O による集団型読み込み操作の様子（2 プロセス）

2.1 Two-Phase I/O の動作

図 1 に 2 プロセスによる集団型読み込みの TP I/O の動作を示す．TP I/O において，集団型 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 I/O アグリゲータと呼ぶ．標準では MPI_File_open を呼び出したプロセスが動作するノード上の 1 プロセスが I/O アグリゲータとなる．I/O 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．I/O アグリゲータには全体のアクセス領域が分割して割り当てられる．図では 2 つのプロセスのいづれも I/O アグリゲータとなっているため，ファイル領域は 2 つの I/O アグリゲータへ分割して割り当てられている． 集団型読み込みの場合，TP I/O はまず始めに，I/O 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）と呼ばれる一時的なバッファへデータを読み込む．その後に，通信によって各プロセスが要求するデータが渡される．TP I/O のファイル読み込み操作ではこのように，ファイル上のデータをまとめて読み込む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．

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 read-modify-write と呼ばれる操作を行う．

Vol.2011-ARC-197 No.35
Vol.2011-HPC-132 No.35
2011/11/29

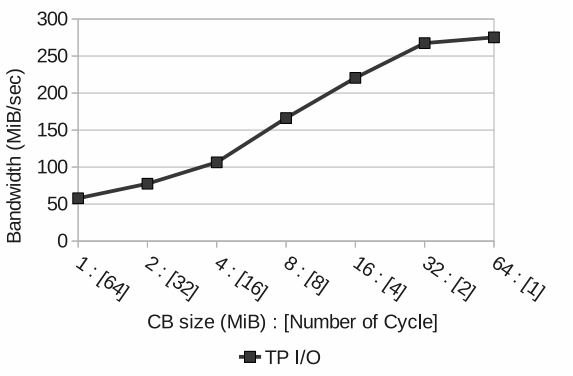


図 2　CB サイズを変えた場合の TP I/O の性能

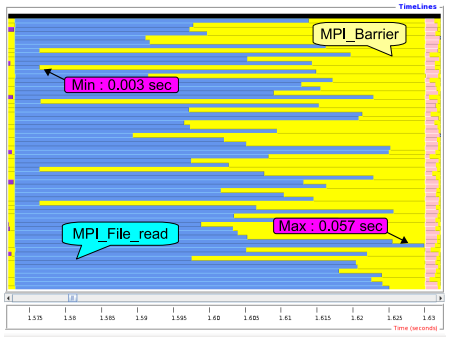


図 3　64 ノードによるファイル上に連続に並んだ 4MiB のデータに対する非集団ファイル読み出し

ROMIO の実装において CB サイズはファイルオープン時に cb_buffer_size を設定したヒントを与えることで任意に決定することが出来る．T2K オープンスパコンに標準にインストールされている MPI ライブラリにおいては，CB サイズのデフォルトサイズは 4MiB である．CB サイズがアグリゲータに割り当てられた領域より小さい場合，CB サイズ分の I/O とデータ並べ替え（以下，この一連の処理をサイクルと呼ぶ）を複数サイクル繰り返す．

2.2　Two-Phase I/O の問題点

TP I/O は複数サイクル実行される場合に，I/O とデータ並べ替えのための通信の回数の増加によりオーバーヘッドが増加し性能が低下する．

図 2 は，T2K オープンスパコンとは異なる，9 ノード構成の PC クラスタ環境で Two-Phase I/O の性能を測定したものである．性能評価には HPIO ベンチマーク[9] を利用し，別の 9 ノード構成クラスタ上の PVFS2 ファイルシステム (1 メタデータサーバ，8 データサーバ) に対して集団型読み込みを行った．HPIO ベンチマークは 8 プロセスで実行した．ここで 1 ノードあたり 1 プロセスが起動し，全てのプロセスをアグリゲータとしている．ファイルサイズは 512MiB であり，各アグリゲータには 64MiB のデータ領域が割り当てられる．図から，CB サイズが 64MiB から減少するにつれて，サイクル数が増加することで性能が低下していることがわかる．このように TP I/O 性能は CB サイズの大きさに依存しており，性能向上のためにはより大きなメモリを確保する必要がある．性能が最大となるのは CB サイズがアグリゲータに割り当てられたデータサイズと同じ場合である．そのため，データサイズが増加すると最大性能を得るために必要なメモリサイズは増加する．

また，TP I/O では I/O とデータ並べ替えを交互に行うという特性上，ノード間での I/O コストのバラつきによって生じる待ち時間が蓄積する問題がある．

TP I/O ではデータ並べ替えの際に，通信するデータに関する情報を取得するために，集団型 I/O を呼び出した全プロセスと全対全の通信を行っている．そのため，データ並べ替えの度に全プロセスと同期することになる．TP I/O では I/O とデータ並べ替えは交互に実行されるため，データ並べ替えは同期によってその前に実行された最も I/O の完了の遅いプロセスを待つことになる．

図 3 は T2K 上で 64 プロセスが 256MiB のファイルに対して，それぞれファイル上の異なる 4MiB のデータを読み込んだ時の振る舞いである．計測はベンチマークプログラムに HPIO ベンチマーク，ファイルシステムに Lusre(ストライプサイズ 1MiB，ストライプカウント 30) を用いている．HPIO ベンチマークを非集団型で連続なアクセスパターンを用いて実行した．この場合，HPIO ベンチマーク内部では各プロセスが MPI_File_read 関数をそれぞれ異なるデータ領域に対して実行する．また，HPIO ベンチマークでは全てのプロセスの I/O が完了する時間を計測するため，MPI_File_read 関数の前後に MPI_Barrier 関数が呼ばれている．

結果の可視化は，HPIO ベンチマークを MPE ライブラリを用いてコンパイルすることで，実行時に MPI 関数をトレースしたログファイルを出力させ，そのログファイルを jumpshot

Vol.2011-ARC-197 No.35
Vol.2011-HPC-132 No.35
2011/11/29

を用いて可視化している.[10]．図 3 から，ファイル読み込みに要する時間は最小で 3 ミリ秒，最大で 57 ミリ秒であり，10 倍以上の開きがあることが分かる．また，I/O 後の同期では，最も早く I/O が完了したプロセスにおいて，自身の I/O に要した時間の約 18 倍にあたる 54 ミリ秒の待ち時間が発生していたことが分かる．

TP I/O では 1 サイクルの度に全プロセス間で同期をとるため，サイクル数が増加すると，このような待ち時間が蓄積すると考えられる．

3. パイプライン型 Two-Phase I/O

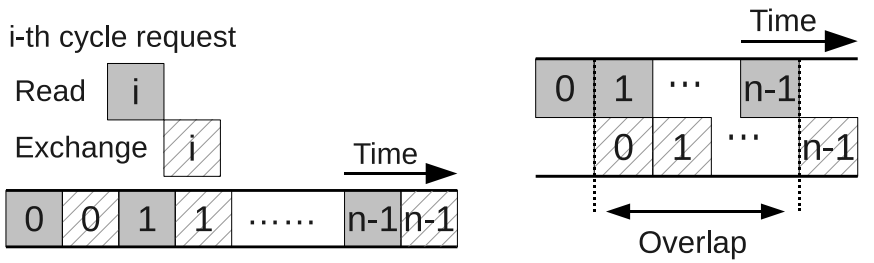
本提案手法であるパイプライン型 TP I/O について以下，説明する．

図 4 は，(a)TP I/O と (b) パイプライン型 TP I/O において，集団型ファイル読み込みのデータ並べ替えとファイルアクセスが処理されるタイミングを表した図である．図では，ファイル読み込みとデータ並べ替えの処理を区別して表現している．一つの処理は CB サイズのファイル領域に対する処理である．オリジナルの TP I/O ではファイル読み込みとデータ並べ替えが交互に行われるのに対し，パイプライン型 TP I/O では，CB サイズ単位で分割された領域を順に処理していく際に，前回のファイル読み込みが現在のデータ並べ替えとオーバーラップして実行されている．

パイプライン型 TP I/O では CB サイズを小さくすることでオーバーラップする割合が増加する．図 4(b) より，パイプライン型 TP I/O は，最初の読み込みと最後の並べ替え以外のサイクル数-1 回のファイル読み込みとデータ並べ替えはオーバーラップして実行される．サイクル数はアグリゲータに割り当てられる領域を CB サイズで割った数であるので，CB サイズを小さくすることで，サイクル数は増加し，オーバーラップされる処理も増加する．

このようにパイプライン型 TP I/O の性能を最大化するためには，より小さな CB サイズを設定しサイクル数を増やす必要がある．そのため，オリジナルの TP I/O よりもメモリ使用量を減らした上で性能向上が期待できる．

また，TP I/O においてデータ並べ替えは，最も I/O の完了が遅いプロセスを待って実行されるが，一方で次に実行される I/O はデータ並べ替えを待って実行されるため，間接的に次に実行される I/O は前回の I/O の完了が遅いプロセスの影響を受ける．パイプライン型 TP I/O ではこの影響を緩和するために，複数の CB を用いている．これにより，I/O 処理は前回のデータ交換の完了を待たずに別の CB を利用して実行出来る．また，I/O のコストがデータ並べ替えのコストよりも小さい場合，データ並べ替えが完了する前に I/O は完了する．この場合，次のデータ並べ替え時にはすでに I/O 処理が完了した CB が供給



(a) オリジナル TP I/O　(b) パイプライン型 TP I/O

図 4　TP I/O の処理タイムライン

されると考えられる．従ってこの場合，データ並べ替えは I/O の完了の最も遅れたプロセスに待たされることがなく，待ち時間の蓄積を回避できると考えられる．

この図では，データの並べ替えとファイル読み込みに要する時間が同じであるが，実際には通信やファイルアクセスの性能，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，ファイル読み込みとデータ並べ替えのコストのバランスは変わる．このコストの大きさの差が顕著になると，オーバーラップによって隠蔽されるコストの全体に占める割合は小さくなる．そのため，オーバーラップによって得られる性能向上は小さくなると考えられる．

今回はマルチスレッド処理による実装 (以下，マルチスレッド版) と非同期 I/O による実装 (以下，非同期版) の 2 つの実装を用いて Lustre における性能評価を行った．なお，確保する CB の数はサイクル数を超えないようにした．これは，サイクル数以上の CB を確保しても使われないためである．また，非同期 I/O による実装では，2 つの CB を用いてダブルバッファリングを行っているが，サイクル数が 1 の場合は CB は一つしか確保しない．また，今回はファイル読み込みの TP I/O の実装を用いて性能評価を行った．

以下、マルチスレッド版と非同期版のそれぞれの実装手法について説明する。

3.1 マルチスレッド版 Two-Phase I/O

マルチスレッドによるパイプライン型 TP I/O のファイル読み込みの実装では，Pthreads[11] を用いて 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．

マルチスレッドによるパイプライン型 TP I/O のファイル読み込みの動作を**図 5** に示す。

マルチスレッドによる実装では TP I/O のファイル読み込みを専用スレッド（Read スレッド）上で実行し，データ並べ替えの処理はメインスレッド上で実行する．メインスレッ

Vol.2011-ARC-197 No.35
Vol.2011-HPC-132 No.35
2011/11/29

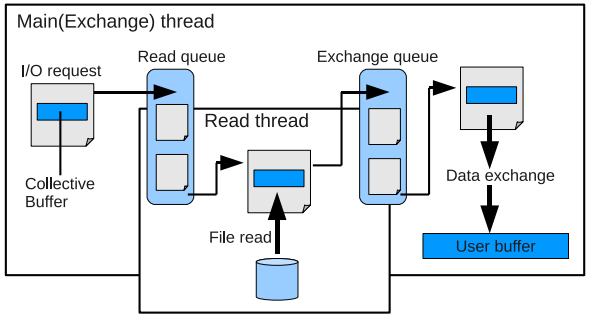


図 5　マルチスレッド版パイプライン型 TP I/O の実装

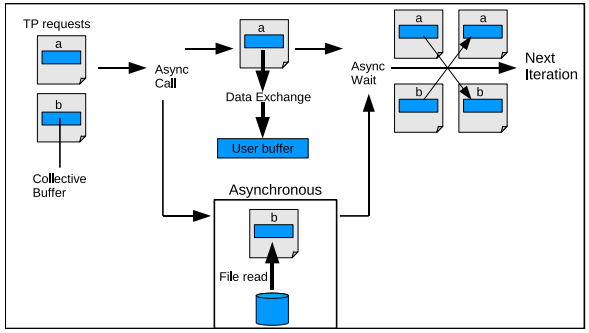


図 6　非同期版パイプライン型 TP I/O の実装機構

ド上では I/O の要求 (以下，I/O 要求) の作成も行う．スレッド間での I/O 要求の受け渡しは，スレッド間で共有して設けてあるキューを用いて行われる．マルチスレッド版では，任意の数の CB を確保することが出来る．複数の CB を確保すれば，ファイル読み込みが終わった際に，メインスレッドがデータ並べ替えを終えて CB を開放するのを待たずに次のファイル読み込みを行うことができる．

本実装の動作は，まずメインスレッドが I/O 要求を CB の数だけ作成し，それを Read スレッドのキュー (Read キュー) 入れる．Read スレッドは Read キューに I/O 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，メインスレッドのキュー (Exchange キュー) へ渡す．メインスレッドは，初めの I/O 要求作成の後は Exchange キューを見て，I/O 要求があればそれに対応する CB を用いてデータ並べ替え処理を行う．そして，メインスレッドは 1 つ処理を終える毎に新たに 1 つ I/O 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．

3.2　非同期版 Two-Phase I/O

非同期 I/O による実装では，2 つより多くの CB を持たせる実装が出来なかった．そのため，非同期関数を用いた 2 つの CB による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て実行する． 非同期 I/O によるパイプライン型 TP I/O の動作を**図 6** に示す．

非同期版は，まず初めに，最初のファイル読み込みを同期型関数で実行する．その後は，図にあるようにデータ並べ替えを行う前に次のサイクルのデータ読み込みを非同期関数で行う．これによって，データ並べ替えと次のサイクルのデータ読み込みがオーバーラップして実行される．データ並べ替え処理が終わった後にデータ読み込みの完了を待ち，その後に次のサイクルを実行する．これを割り当てられた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．

4.　Lustre ファイルシステム

Lustre[2] はオープンソースの並列ファイルシステムである．Lustre を構成する要素は，メタデータを管理する MDS（metadata server）とそのデータを保管する MDT（metadata target），ファイルのデータを管理する OSS（object storage server）とそのデータを保管する OST（object storage target），そして Lustre ファイルシステムへアクセスするための Lustre クライアントである．Lustre ファイルシステムではファイルを任意の数の OST へ分割して保存することが出来る．この分割数をストライプカウントと呼ぶ．また，分割するデータのサイズも任意に決定出来る．このサイズをストライプサイズと呼ぶ．これらは Lustre クライアント上で設定可能である．Lustre ファイルシステムでは，ver 1.6.5.1 からファイル読み込み性能の向上のために Read-ahead と呼ばれる機構を備えている[12] Read-ahead の起動条件は，同じファイル記述子を用いて連続なデータに対するファイル読み込みを 2 回以上行った場合であり，かつ，データがクライアント側キャッシュ上に無い場合である．また，Read-ahead は不連続な領域の読み込みが実行されるとアルゴリズムがリセットされ，再び起動条件を満たすまで Read-ahead は実行されない．Read-ahead のサイズは，1MiB か，ファイル読み込みのサイズがそれより大きい場合にはファイル読み込みのサイズ分を先読みする．なお，Read-ahead がキャッシュに蓄えるデータの上限値は max_read_ahead_mb で

Vol.2011-ARC-197 No.35
Vol.2011-HPC-132 No.35
2011/11/29

表 1　T2K オープンスパコンの仕様

CPU	AMD Quad Core Opteron 2.3 GHz, 4 Cores, 4 Sockets
memory	32 GB / node
network	Myrinet-10G × 4 / node
OS	RedHat Enterprise Linux 5.1
MPI library	MPICH2 ver. 1.4 ベースの改造実装
並列ファイルシステム	Lustre ver. 1.8.11

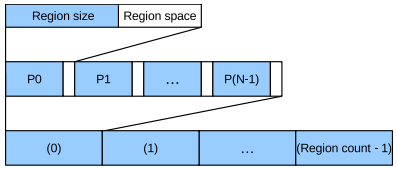


図 7　HPIO ベンチマークにおけるファイル上のデータ配置

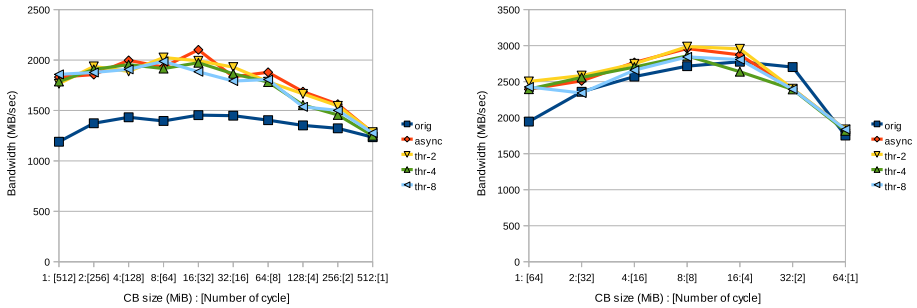
ある．Read-ahead が起動している間は max_read_ahead_mb で設定されたサイズまでデータを読み出す．

今回利用する T2K オープンスパコン上で利用可能な Lustre ファイルシステムは，MDT が 1 ノード，OST が 30 ノードである．したがってストライプカウントの最大数は 30 である．また，max_read_ahead_mb は 40MiB である．

5. 性能評価

T2K オープンスパコンの 64 ノードを利用して性能評価を行った。計算機の概要を**表 1** に示す。MPI ライブラリにはパイプライン型 TP I/O を実装した mpich2-1.4 を用いた．プロセス間の通信には IP over Myrinet を利用している．また，今回の評価では Lustre のストライプサイズを 1MiB，ストライプカウントを 30 に設定した．

また，性能評価には HPIO ベンチマーク[9] を用いた．HPIO ベンチマークでは，メモリ上とファイル上のそれぞれのデータ配置を連続か不連続かに任意に決定できる．今回はメモリ上を連続，ファイル上を不連続に設定した．今回のファイル上の各プロセスがアクセスする領域の配置を**図 7** に示す．Region size は各プロセスがアクセスするデータが配置されており，Region space を挟んでプロセスのランク番号順に並んでいる．このひとまとまりが Region count 回繰り返し配置されている．一方メモリ上のデータは連続であり，各プロセスが (Region size)×(Region count) のサイズを持ち，ファイル読み込みの場合には，このメモリへファイルを読み込む．TP I/O を用いない場合は各プロセスが直接ファイル上の Region size へアクセスするため，Region space の領域へはアクセスしない．しかし，TP I/O では要求をまとめてアクセスする際，アクセス回数を最小にするために領域の端にある空白を除いた全領域にアクセスする．従って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．



(a)8 ノード　　(b)64 ノード

図 8　HPIO を用いた Lustre に対する集団型ファイルを読み込みの性能

今回の性能評価ではオリジナルの TP I/O(以下，オリジナル) と非同期関数版 TP I/O(以下，非同期版) およびマルチスレッド版 TP I/O(以下，マルチスレッド版) の性能を比較する．マルチスレッド版では CB の数を 2，4，8 の 3 つの場合で測定し，I/O コストのノード間ばらつきによる影響の緩和の有効性を調べた．

5.1 Lustre における性能評価

HPIO ベンチマークを用いて，ノード数 8 ノードと 64 ノードの 2 つの場合について，CB サイズを 1MiB から 2 倍ずつ増やしアグリゲータへ割り当てられたデータサイズまで変化させた場合の集団型ファイル読み込みの性能をそれぞれ測定した．ファイルサイズは全プロセスで 4GiB に設定し，1 ノードにつき 1 プロセスで起動し，全てのプロセスをアグリゲータとした．HPIO のパラメタは Region size=1024B, Region space=0 に設定した．Region count は，ファイルサイズが 4GiB となるように，8 ノードの場合は 512×1024 に，64 ノードの場合は Region count を 64×1024 に設定した．この場合に各アグリゲータへ割り当てられるデータサイズは，それぞれ 512MiB と 64MiB である．

性能評価の結果を**図 8** に示す．図 8(a) が 8 ノードの結果で (b) が 64 ノードの結果であ

Vol.2011-ARC-197 No.35
Vol.2011-HPC-132 No.35
2011/11/29

る．横軸は CB サイズとサイクル数であり，括弧の中がサイクル数である．図中の凡例にある「thr-N」は N 個のバッファによるマルチスレッド版の実装を表している．また，マルチスレッド版において，バッファ数はサイクル数より多くは確保されない．これはサイクル数より多いバッファ数を確保しても利用されないからである．バッファ数がサイクル数より大きくなる場合，バッファ数はサイクル数と同じである．サイクル数が 1 回の場合には実際にはマルチスレッド版と非同期関数版のどちらもバッファは 1 個しか確保されず，従って I/O とデータ並べ替えの処理はオーバーラップはされない．図 8 の (a)(b) のどちらの場合においても，サイクル数が 1 回のときはオリジナルと同等の性能となっている．これはマルチスレッド版と非同期版がともに，サイクル数 1 の場合ではオーバーラップされないために，オリジナルと同じ振る舞いをしているためである．

図 8 から 8 ノードの場合では，サイクル数が 1 の場合を除く全 CB サイズで本提案手法の実装の方が性能向上している．各実装の性能の最大値を比較すると，非同期版の性能 (CB=16MiB 時) と，マルチスレッド版の性能 (CB 数 2，CB=8MiB 時) は，オリジナルの性能の最大値 (CB=16MiB 時) と比較して，それぞれ約 1.45 倍と約 1.39 倍性能が向上している．CB が 1MiB の場合では，オリジナルと比較して，非同期の性能は約 1.54 倍，マルチスレッド版の CB 数 8 の場合の性能が約 1.56 倍である．また，8 ノードの場合では CB が 1MiB の場合でも本提案手法はオリジナルの性能の最大値を上回っている．CB が 1MiB の場合の本提案手法と，オリジナルの最大の性能 (CB=16MiB) を比較すると，非同期版とマルチスレッド版の CB 数 2 の場合では，CB として確保したメモリ量は 8 分の 1 でありながら，それぞれ 1.26 倍と 1.21 倍性能が向上している．またマルチスレッド版の CB 数 8 の場合でも CB として確保したメモリ量が 2 分の 1 でありながら性能を約 1.28 倍向上させている．

一方 64 ノードでは，各実装の最大の性能を比較すると，オリジナルの性能 (CB=16MiB) の最大値と比較して非同期版 (CB=8MiB 時) とマルチスレッド版 (CB 数 2，CB=8MiB 時) ともに約 1.07 倍となっている．CB が 1MiB の場合では本提案手法はオリジナルの性能と比較して大きく向上している．この場合の非同期版とマルチスレッド版の最大性能 (CB 数 2) はオリジナルと比較してそれぞれ，約 1.23 倍と約 1.29 倍である．

本提案手法は多くの場合でオリジナルよりも性能が向上しているが，CB が 32MiB の場合には明らかにオリジナルの方が提案手法の性能を上回っている．この場合サイクル数が 2 であるため，CB は 2 以上確保されない．従ってこの場合のマルチスレッド版の CB 数は全て 2 である．この時のオリジナルの性能はマルチスレッド版の約 1.3 倍である．なぜこのようになるかは現在のところよくわかっていない．

ノード数が 8 の場合と 64 の場合のどちらの場合においても，オリジナルの TP I/O の性能が，CB サイズの減少に伴って性能が低下するという想定された結果になっていない．これは，Lustre の Read-ahead の機能によるものと考えられる．オリジナルの TP I/O において，データ並べ替えが実行されている間に Read-ahead によってバックグラウンドで次の領域のデータがクライアントのキャッシュに読み出されており，それによって，I/O コストの一部がデータ並べ替えに隠蔽されていると考えられる．しかしながら，結果を見ると，多くの場合で本提案手法が勝っていることから，本提案手法のほうが Read-ahead と比較してより多くの I/O コストの隠蔽が可能であると考えられる．本提案手法が Read-ahead による I/O コストの隠蔽よりも性能が勝る原因として，Read-ahead では隠蔽できないコストの存在が考えられる．オリジナル TP I/O はデータ並べ替えと I/O を交互に行っているが，データ並べ替え処理の間に Read-ahead によってデータ読み込みが実行されても，次の I/O 処理時に行うクライアントキャッシュからのデータコピーのコストは隠蔽することが出来ないと考えられる．また，Read-ahead による読み込みサイズは max_read_ahead_mb で設定されたサイズが上限である．そのため，CB サイズがその上限より大きい場合には，Read-ahead は次の I/O 処理で要求するデータを全て先読みすることが出来ない．その場合，I/O 処理において不足するデータを追加で読み出す必要があり，そのコストは隠蔽できないと考えられる．

また，今回は I/O コストのノード間のばらつきの緩和を目的として，スレッド版実装において複数の CB を確保する実装を行っているが，今回の結果からはその効果は観測されなかった．

6. 関 連 研 究

集団型 I/O を高速化する研究はこれまでにも行われている．View-based I/O[13] 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．TP I/O と異なる点は，TP I/O が動的にファイル領域の割り当てを行い，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，View-based I/O ではファイル領域の割り当てを静的に行い，アクセス時のオフセットと長さの情報の通信を不要とすることで，性能を向上させている．また，キャッシュバッファを持たせることでも性能向上を実現している．

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も提案されている[14]．この実装では I/O がデータ並べ

Vol.2011-ARC-197 No.35
Vol.2011-HPC-132 No.35
2011/11/29

替え処理とは別スレッドで実行されており，I/O 処理をオーバーラップさせている．また，GPFS に対する MPI-IO の最適化実装を行った研究[15] においても，ダブルバッファリングによって I/O とデータ並べ替えをオーバーラップして実行する提案がなされている．I/O とデータ並べ替え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，これらは GPFS の機能に依存した実装であるため，他のファイルシステム上では利用できない．本提案手法は，Lustre や PVFS2 の他，ROMIO がサポートする多くのファイルシステム上で利用することが可能である．

7. まとめと今後の課題

本稿では，T2K オープンスパコンを用いて Lustre ファイルシステムに対する本提案手法の性能評価を行った．その結果，ノード数が 8 の場合に，オリジナルの TP I/O に対して最大で 1.56 倍の性能が確認でき，条件によっては CB として確保するメモリ量を 8 分の 1 に減らした上で 1.26 倍の性能向上を確認することができた．ノード数が 64 の場合では最大で 1.29 倍の性能向上が確認できた．また，今回オリジナルの TP I/O においても，本提案手法と同様に TP I/O 内部の I/O と通信がオーバーラップすることによると考えられる性能向上が観測された．これは Lustre ファイルシステムの Read-ahead の効果によるものと考えられる．しかしながら，今回の性能評価において，多くの場合でオリジナルの TP I/O の性能を提案手法が上回っていることから，本提案手法のほうが，Read-ahead による I/O コストの隠蔽よりも，より多くのコストを隠蔽できるものと考えられる．しかし，条件によっては Read-ahead による性能向上が本提案手法の性能を上回る場合が観測された．この原因については今後調査する予定である．また，マルチスレッド版 TP I/O においてバッファ数を増やすことによって I/O コストのノード間のばらつきの影響の緩和する手法に関しては，今回はその効果は観測できなかった．これについても今後詳しく調査検討する予定である．

謝辞 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計算に資するシステムソフトウェア技術の創出」及び東京大学情報基盤センター スーパーコンピュータ若手利用者推薦制度の支援を受けております．

参 考 文 献

1) PVFS2: . http://www.pvfs.org/pvfs2/.
2) Lustre: . http://www.lustre.org/.
3) General Parallel File System: . http://www-03.ibm.com/systems/software/gpfs/.
4) MPI Forum: . http://www.mpi-forum.org/.
5) 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).
6) Thakur, R., Gropp, W. and Lusk, E.: Optimizing noncontiguous accesses in MPI-IO, *Parallel Computing*, Vol.28, No.1, pp.83–105 (2002).
7) 六車 英峰，辻田 祐一，堀 敦史，並木 美太郎：Two-Phase I/O の高速化に関する一検討，情報処理学会研究報告 2011-HPC-130, No.132 (2011)．オンラインドキュメント．
8) T2K Open Supercomputer Alliance: . http://www.open-supercomputer.org/.
9) 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).
10) MPICH2: . http://www.mcs.anl.gov/research/projects/mpich2/.
11) 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).
12) Sun Microsystems,Inc.: *Lustre 1.8 Operations Manual* (2010). http://www.lustre.org/.
13) Blas, F. J.G., Isaila, F., Singh, D.E. and Carretero, J.: View-Based Collective I/O for MPI-IO, *CCGRID*, pp.409–416 (2008).
14) Blas, J. G., Isaila, F., Carretero, J., Singh, D. and Garcia-Carballeira, F.: Implementation and Evaluation of File Write-Back and Prefetching for MPI-IO Over GPFS, *International Journal of High Performance Computing Applications*, Vol.24, pp.78–92 (online), DOI:10.1177/1094342009359015 (2010).
15) pierre Prost, J., Treumann, R., Hedges, R., Jia, B. and Koniges, A.: MPI-IO GPFS, an optimized implementation of MPI-IO on top of GPFS, *In Proceedings of Supercomputing 2001* (2001).