Vol.2014-OS-130 No.3
2014/7/28

並列分散ファイルシステムに対するオンライン重複除外機構の導入に向けて

松宮 遼[1,a)]　佐々木 慎[1,b)]　高橋 一志[1,2,c)]　大山 恵弘[1,2,d)]

概要：重複除外機構とは，ファイル内や異なるファイル間で，重複したデータを記録装置上には 1 度しか記録しないというものである．重複除外機構で導入されている方法の 1 つとして，Content-Defined Chunking (CDC) がある．CDC とは，ファイルをその内容に基づいて複数のチャンクに分割するものである．CDC による重複除外機構は，バックアップシステムなどに実装されている．そして近年は，高性能計算におけるストレージシステムへの CDC による重複除外機構の導入が検討されている．そこで我々は，高性能計算で利用されている並列分散ファイルシステムの Gfarm の I/O サーバに対して，CDC による重複除外機構を実装し，マイクロベンチマークによる評価を行った．その上で明らかとなった課題について本稿で述べる．

Towards Implementation of Online Deduplication Systems in Parallel Distributed File Systems

Abstract: Deduplication systems enable to write deduplicated data once in recording devices. Content-Defined Chunking (CDC) is one of mechanisms implemented in deduplication systems. CDC divides a file into multiple chunks. Deduplication systems which adopt CDC are implemented in such backup systems. Recently, deduplication systems which implement CDC are going to be adopted into the storage systems for high performance computing. Hence, we adopted a deduplication system which implements CDC into Gfarm I/O server as a parallel distributed file system used in the field of high performance computing. We also evaluated our system by microbenchmarks. In this paper, we discuss the issue of implementing our system.

1. はじめに

高性能計算におけるストレージシステムは，図 1 のように階層性を意識した設計となっていることがある．Gfarm [1] や Lustre [2] に代表される並列分散ファイルシステムは，テープストレージをはじめとする大容量ストレージより必要なデータをロードする．計算ノードは，並列分散ファイルシステム上のデータを使い，計算を行っていく．計算結果は，まず複数の I/O サーバからなる並列分散ファイルシステムに格納され，必要に応じてテープストレージなどといった大容量ストレージにストアされる．

[1] 電気通信大学
The University of Electro-Communications
[2] 科学技術振興機構 CREST
JST CREST
[a)] r.matsumiya@ol.inf.uec.ac.jp
[b)] sasashin@ol.inf.uec.ac.jp
[c)] kazushi@inf.uec.ac.jp
[d)] oyama@inf.uec.ac.jp

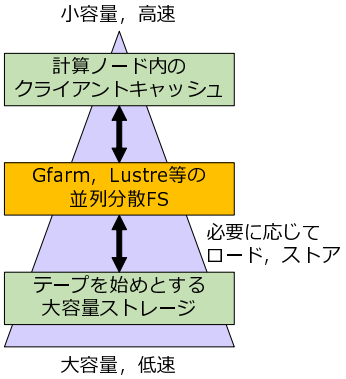

図 1　階層性を意識したストレージシステム
Fig. 1　Storage systems in consideration of hierarchy

このようなストレージシステムの下で，計算ノードが，並列分散ファイルシステムの容量を超えるデータを，読み書きをするような計算を行おうとする場合を考える．この場合，並列分散ファイルシステムは，計算ノードが計算し

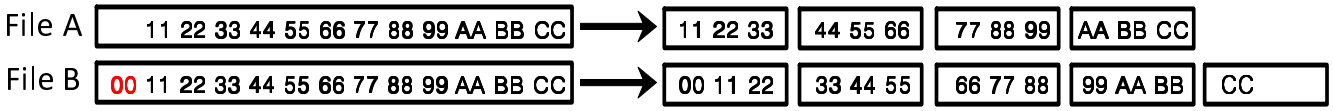

図 2　FSC の例
Fig. 2　Example of FSC

ている間でも，大容量ストレージとの間でロードとストアを繰り返し行う必要が生じる．この繰り返しが，高性能計算におけるボトルネックとなる可能性がある．この繰り返しの回数を減らす為に，並列分散ファイルシステムに重複除外機構を導入することが検討されている [3]．

重複除外機構とは，ファイル内や異なるファイル間で，重複したデータを記録媒体上には 1 度しか記録しないというものである．重複除外機構は，多くのストレージシステム [4], [5], [6], [7], [8] に導入されている．

重複除外機構で導入されている方法の 1 つとして，Content-Defined Chunking (CDC) がある．CDC とは，ファイルをその内容に基づいて可変長のチャンクに分割する方法である．

ストレージに対する重複除外には，オフラインによるものとオンラインによるものの 2 つがある．オフラインによるものとは，重複除外を行っている間はストレージに対するアクセスがないということが保証されているものである．オンラインによるものとは，そのような保証がないものである．本研究では，計算ノードが，並列分散ファイルシステムの容量限界を超えるようなデータを必要とすることを想定する．この場合，大容量ストレージから必要なデータをロードしている間に，計算ノードからファイルを要求されることがあり，オフラインでの重複除外を行うことは困難である．

CDC はストレージの空間効率を高めるには有用な方法である．しかし，CDC は分割するファイルが大きなサイズのとき，ファイル分割の為の計算に時間がかかることが知られている．オンラインで重複除外処理を行うためには，この部分を高速にすることが必要であると考えられている．事実，CDC によるファイル分割の並列化の研究は行われている [9], [10], [11], [12], [13], [14], [15]．しかし，これらの研究による方法を用いても，結局はファイル分割によるオーバーヘッドが発生してしまう．このオーバーヘッドは，書き込むデータのサイズに対して増加する．したがって，非常に巨大なサイズのファイルを扱う時や，短時間で膨大の数のファイルを扱う時，このオーバーヘッドは無視できなくなる．

CDC による重複除外処理をオンラインに行う先行研究が存在する [7], [8] が，これらは 1 ノードで重複除外処理を行うことを前提としている．高性能計算では，ストレージに対し，クライアントからファイルの書き込みが，短時間で大量に要求されることがある．1 ノードで重複除外処理を行っていた場合，計算資源が重複除外処理によって枯渇してしまい，レイテンシが発生してしまう．

高性能計算での利用を前提として，CDC による重複除外処理を導入した先行研究も存在する [16] が，この研究は計算ノードが重複除外処理を行う．つまり，計算ノードが計算している間での重複除外処理は，計算ノードの性能低下を招く．

したがって，本研究では，高性能計算での利用を想定し，CDC による重複除外処理をオンラインに行う手法を提案する．提案手法では，I/O サーバで重複除外処理を行うことにより，重複除外処理をクライアント以外の複数ノードで行う．そして，提案手法ではクライアントの処理と非同期に重複除外処理行う．これにより，ファイル分割によるオーバーヘッドを隠蔽することができるので，従来手法よりも高速に，並列分散ファイルシステムの読み書き処理が行えると考えられる．また提案手法を，Gfarm の I/O サーバデーモン (gfsd) に実装し，評価を我々は行った．

本稿の構成を以下に示す．まず，重複除外や CDC について述べる．その後，CDC による重複除外処理におけるファイル分割処理の計算オーバーヘッドや，CDC による重複除外をオンラインで行う上での課題について述べる．そして，その課題を解決する手法の提案，gfsd に対する提案手法の実装，及びその評価それぞれ述べる．最後に，関連研究について述べた後，まとめと今後の課題について述べる．

2. 重複除外の方法

本章では，重複除外の方法について述べる．重複除外には，ファイル単位のものと，チャンク単位のものがある．ストレージにおいては，チャンク単位での重複除外が使われることが多い．本章では，チャンク単位での重複除外を行うときに，ファイルをチャンクに分割する手法について述べる．まず，ファイルを固定長のチャンクに分割する，Fixed-Sized Chunking (FSC) とその欠点について説明する．その後，CDC について説明する．

2.1 FSC

FSC は，ファイルを固定長のチャンクに分割する方法で，ZFS [4] や Venti [5] などによって利用されている．しかし FSC には，重複したデータを持つ 2 つのファイルの

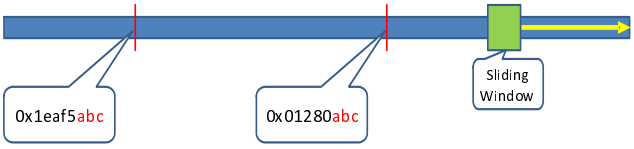

図 3　CDC の例
Fig. 3　Example of CDC

間に重複データに位置のズレが有った時，重複データのズレが例え 1 バイトであっても，それ以降のファイルの重複を検出できないことがある．

FSC の例を図 2 に表す．この例ではファイルを 3 バイト単位で分割した．左上の四角はファイル A の内容を表し，左下の四角はファイル A の先頭に 1 バイト追加したファイル B の内容を表す．このような状態で，ファイル A とファイル B の間で重複除外を行った場合，ファイル B の先頭 1 バイト以外の部分が除外されるのが理想である．しかし，FSC によるファイル分割を行った場合，ファイル A は右上の四角，ファイル B は右下の四角のように分割される．チャンク単位で考えた時，どの 2 つのチャンクを取っても，等しくならない．ゆえに，ファイル B のどの部分の重複も除外されない．

2.2　CDC

CDC は，ファイルをその内容に基づいて，可変長のチャンクに分割する手法であり，重複データの位置にズレがあっても，重複を検出できることがある．低帯域幅のための分散ファイルシステムである LBFS [6] によってはじめて実装され，Data Domain [7] や HydraFS [8] などでも利用されている．CDC はチャンクの境界を定めるために，固定長の sliding window を用いる．Sliding window とは，ファイルの先頭から 1 バイトずつスライドさせるものである．通常，sliding window の大きさは 48 バイトである．Sliding window 内におけるファイルデータのフィンガープリントを計算する．CDC では通常，この時のフィンガープリントとして Rabin fingerprint [17] を用いる．

Sliding window 内におけるフィンガープリントの下位ビットが，ユーザが事前に指定した値と一致したとする．この時，sliding window の末端をチャンクの境界線とし，重複除外機構はファイルを分割する．ここで利用するフィンガープリントの下位ビット数も，ユーザが指定する．フィンガープリントの下位 n ビットを利用する時，重複除外機構が生成するチャンクの平均サイズは 2^n バイトとなる．

CDC によりファイルをチャンクに分割を行う例を図 3 に示す．青いバーはファイル全体を表し，緑の四角は sliding window を表す．なお，簡単のため，この例におけるフィンガープリントのサイズは 4 バイトとなっている．Sliding window の末端が赤い線の位置にあるとき，sliding window 内におけるフィンガープリントの下位 12 ビットが特定値 (図では 0xabc) である．この時，赤い線がチャンクの境界線となる．

CDC には，ユーザが指定したフィンガープリントの下位ビットや，ファイルの内容によっては，チャンクの境界線を 1 つも生成しないということがある．このため，本研究で行った実装では，最大チャンクサイズを設定した．

3.　課題

先述の通り，CDC は重複除外におけるファイル分割において有用である．しかし，ファイルをチャンクへ分割する過程で，$s-w+1$ 回フィンガープリントの計算を行う必要がある．このため，FSC に比べ，CDC の処理速度は低速となってしまう．ここで，s はファイルのバイト数，w は sliding window のバイト数である．

そこで，このフィンガープリントの計算を含め，CDC による重複除外の処理を，クライアントの処理と非同期的に実行させる手法を我々は考える．これにより，フィンガープリントの計算オーバーヘッドを隠すことができる．

CDC による重複除外の処理を，クライアントの処理と非同期で実行させるには，まず，重複除外処理をどのタイミングで実行させるべきかを考える必要がある．`write` システムコールなどにより，書き込みが発生したタイミングで適宜，重複除外を行わせる方法は不適切である．ファイル内の特定の箇所に対して，複数回の書き込みがあったとする．この場合，書き込みが発生する度にスレッドを生成させる必要がある．しかし，その箇所に対する最後のスレッドが出力したファイル分割の結果のみがディスク上に反映されればよい．つまり，最後のスレッド以外のスレッドの処理は，ファイルの内容のコンシステンシが取れてさえいれば良いという場合，必要のない処理であり，計算資源を浪費してしまう．特定の箇所に対する最後の書き込みが行われたタイミングで，デーモンは非同期に CDC を行うようにするという方法が考えられる．しかし，プログラムを解析して，ある箇所に対する最後の書き込みを検知するのは困難である．

非同期的に重複除外を実行させようとするとき，我々はコンシステンシを考慮しなければならない．あるファイルで重複除外の処理を非同期的に行っていたとする．その処理が完了する前に，重複除外を行っているファイルに対する `close` システムコールが発行されるなどして，クライアント側のファイルディスクリプタが閉じられる．その時，ファイルの内容のコンシステンシは保証されない．このコンシステンシが保証されない場合，高性能計算における，ファイルを用いた計算ノード間のパイプライン処理が正常に行えない可能性がある．ファイルの内容を変更する前の状態を保存し，重複除外処理が完了する前にファイルを参照した場合は，そちらの内容を読みこむという方法も考えられる．しかし，この方法では，ファイルの内容が変更

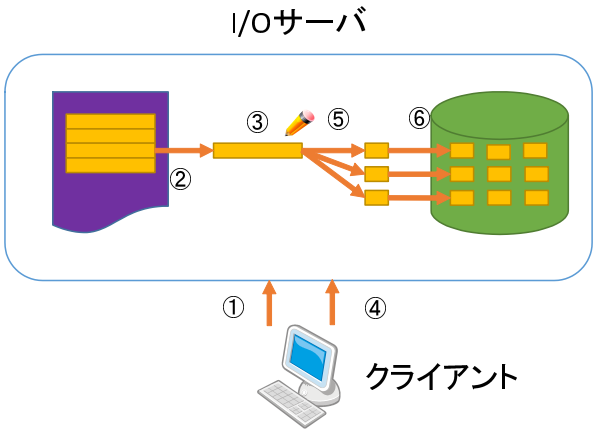

図 4　I/O サーバとクライアントの処理
Fig. 4　Processing of I/O server and client

される前のものなのか，された後のものなのかをクライアント側からは直接確認できない．その場合も，結局，計算ノード間のパイプライン処理が正常に行えない可能性がある．そのため，処理を非同期的に実行させるには，我々はコンシステンシを考慮する必要がある．

4. 提案手法

3 章で述べた問題を解決させる手法を本章では述べる．提案手法は 2 つからなる．1 つは，クライアントがファイルディスクリプタを閉じた時に，I/O サーバが，クライアントの処理と非同期的に重複除外処理を行うものである．そして，その非同期的な重複除外処理において，close-to-open コンシステンシを保証するというのが，もう 1 つである．

4.1 用語定義

提案手法を述べる上で必要となる用語を本節で説明する．それはチャンクファイルと，マッピング情報である．この 2 つは，I/O サーバによって保持される．

チャンクファイルは，各チャンクの内容が記載されているファイルである．1 つのチャンクに対して，1 ファイルが割り当てられる．

マッピング情報は，並列分散ファイルシステム上のファイルと，どのチャンクが，そのファイルのどこに配置されているかが記録された情報である．ファイルを読み書きする時に I/O サーバにより参照され，書き込みが行われる時に，I/O サーバにより更新される．

4.2 非同期な重複除外

クライアントでは `close` システムコールなどによって，ファイルディスクリプタが閉じられる．提案手法では，その閉じられる処理と同時に，I/O サーバが，ファイルで変更が有った箇所に対して CDC によるファイル分割処理を，クライアントの処理と非同期的に行う．

クライアントによって，ファイルの内容に変更が加わわる時の動作について図 4 を用いて説明する．左の紫色の図形は並列分散ファイルシステム上のファイル，黄色の四角はチャンク，右の緑色の円筒は I/O サーバ上のディスクを表す．①まず，クライアントによってファイルの内容に変更を加えるリクエストが，I/O サーバに与えられる．②I/O サーバは，命令を受信すると，変更箇所に対応するチャンクの内容を別のファイルにコピーする．この時のコピー先を仮チャンクファイルとする．③その後，変更を仮チャンクファイルに書き込む．以降，既に仮チャンクファイルが生成されているチャンクに対して，変更が加わった場合は，仮チャンクファイルの方に書き込む．仮チャンクファイルは，内容が変わっても，CDC によるファイル分割は行われない．また，ファイルの内容を読み込むときも，読み込む箇所のチャンクで仮チャンクファイルが生成されていれば，仮チャンクファイルの内容を読み込む．

クライアントが，ファイルディスクリプタを閉じた時の動作について同じ図を用いて説明する．④クライアントは，ファイルディスクリプタを閉じる時，非同期的に I/O サーバに対して通知を行う．I/O サーバは，通知を受信すると，閉じようとしているファイルにおける仮チャンクファイルが生成されているかを調べる．閉じようとしているファイルにおける仮チャンクファイルが，1 つ以上生成されていたとする．⑤ この時，I/O サーバは，閉じようとしているファイルにおける全ての仮チャンクファイルに対し，CDC によるファイル分割を行う．⑥ ファイル分割処理が終わった後は，仮チャンクファイルから分割されたチャンクが，I/O サーバのディスク上に保存される．

4.3 コンシステンシの保証

提案手法では，close-to-open コンシステンシを取る．これは，あるファイルに対して `close` の後に `open` が行われた場合に，その `close` までにそのファイルに対して行われた変更は，`open` 以後の操作で必ず観測されることを保証するものである．このコンシステンシでは，あるアプリケーションが `write` などによってファイルの変更を行っても，そのアプリケーションによって `close` されるまで，その変更を即座に他のアプリケーションやクライアントが観測するとは限らない．close-to-open コンシステンシは，Gfarm や NFS [18] などでも取られている．

ファイルとチャンクのマッピング情報は，ファイルへの書き込みがあった時と，CDC によるファイル分割処理の時に更新される．提案手法では，マッピング情報が更新された時，古いマッピング情報は無効化される．この時，他のクライアントから古いマッピング情報に対してアクセスされた時，どのようなアクセスかによって I/O サーバの挙動が異なる．古い情報に対するアクセスが，ファイル分割処理であった時，I/O サーバはそのアクセスを無効化させ

る．古い情報に対するアクセスが，ファイルへの書き込みであった時，I/O サーバは新しいマッピング情報を元に再度ファイルへの書き込み処理を行う．

5. 実装

提案手法による効果を測定するために，我々は gfsd に提案手法を実装した．また，本研究では，Gfarm の FUSE [19] 用クライアントである gfarm2fs によって，計算ノードは Gfarm のメタデータサーバ及び I/O サーバに接続することを前提とする．gfarm2fs は，ファイルを閉じる時，FUSE 側のファイルディスクリプタが正常に閉じられれば，メタデータサーバや I/O サーバ側の処理とは非同期的に，アプリケーションに処理を返す．これは FUSE の仕様によるものであるが，本研究ではこの仕様を利用することで，非同期的な重複除外処理を実現させている．

5.1 4 種類のファイル

提案手法を実装した gfsd は，マッピング情報ファイル，チャンクファイル，仮チャンクファイル，ディスクリプタ情報ファイルの 4 種類のファイルの読み書きを行う．マッピング情報ファイル，チャンクファイルはディスク上のローカルファイルシステムに保存される．仮チャンクファイル，ディスクリプタ情報ファイルは，tmpfs 上に保存される．

マッピング情報ファイルには，ファイルとチャンクの対応関係が記載されている．基本的には，Gfarm 上の 1 ファイルノードにつき，マッピング情報ファイルが 1 つ，gfsd により割り当てられる．I/O サーバ上でのファイル名は，割り当てられたファイルにおける Gfarm 上での i ノード番号，割り当てられたファイルの Gfarm 上でのジェネレーション番号，割り当てられたファイルの Gfarm 上でのパスによって決まる．

チャンクファイルには，4.1 節で述べたように，各チャンクの内容が記載されている．1 つのチャンクに対して，1 ファイルが割り当てられる．I/O サーバ上でのファイル名は，チャンクの内容のハッシュ値 (SHA-1) によって決まる．

仮チャンクファイルには，4.2 節で述べたように，書き込み等によって，内容に変更があったチャンクの内容が記載されている．I/O サーバ上でのファイル名は，元のチャンクの内容のハッシュ値 (SHA-1)，変更が行われたファイルの Gfarm 上の i ノード番号，変更が行われたファイルの Gfarm 上でのジェネレーション番号，変更が行われたファイルにおける元のチャンクのオフセットによって決まる．

ディスクリプタ情報ファイルには，gfsd 内部の処理で必要となるファイルの情報が記載されている．また，ファイルとチャンクのマッピング情報も記載されている．このマッピング情報については後述する．マッピング情報ファイルと同様に，基本的には Gfarm 上の 1 ファイルにつき，ディスクリプタ情報ファイルが 1 つ，gfsd により割り当てられる．ディスクリプタ情報ファイルのファイル名は，割り当てられたファイルにおける Gfarm 上での i ノード番号，割り当てられたファイルの Gfarm 上でのジェネレーション番号，割り当てられたファイルの Gfarm 上でのパス，一番最初に起動された gfsd のプロセス ID によって決まる．

5.2 共有メモリ

gfsd は，異なるクライアントからのファイルアクセスの要求は，異なるプロセスによって処理されている．そのため，クライアントからの接続があるたびに `fork` し，サーバ用プロセスを生成している．よって，あるクライアントから閉じられ，CDC による分割処理が行われているファイルの情報をクライアント間で共有するために，我々は共有メモリを利用する方法を取った．我々は 3 種類の共有メモリを利用する．1 種類は，`mmap` システムコールにより確保され，他の 2 種類は `shmget` システムコールによって確保されるものである．

`mmap` システムコールによりマッピングされるファイルは，ディスクリプタ情報ファイルである．クライアントは Gfarm の API 関数である，`gfs_pio_open` 関数によって，ファイルを開くリクエストを送る．gfsd はそれを受信すると，マッピング情報ファイルとディスクリプタ情報ファイルを開く．もしマッピング情報ファイルやディスクリプタ情報ファイルが存在しなかった場合は新規作成する．ディスクリプタ情報ファイルは，開かれると直ちに gfsd が `mmap` システムコールを使うことで内容がメモリ上に展開される．この時，他の部分の実装を簡潔にさせるために，ファイルにおける各チャンクのオフセットが計算され，`mmap` された領域に書き込まれる．

ファイルへの書き込みがあった時や，CDC によるファイル分割処理が終わった時にマッピング情報の更新が行われる．その後，`MS_INVALIDATE` フラグをつけて，`msync` システムコールを実行する．`MS_INVALIDATE` フラグとは，この `msync` システムコール以降に他のプロセスが行った，同じファイルに対するマッピングメモリへの書き込みをファイルに反映させないというものである．これにより，4.3 節で述べたマッピング情報の無効化を実現させている．

マッピング情報の更新が開始してから，`msync` システムコールにより，マッピングが反映されるまでの間でクライアント間の競合が発生することがある．この競合により，マッピング情報が破壊されてしまう可能性がある．

この競合を解決するために，`shmget` システムコールを使うことで確保できる，共有メモリ領域を，我々は利用することにした．ディスクリプタ情報ファイルと結びついており，ディスクリプタ情報ファイルが作成されると，共有

表 1　ノード 1 台あたりの環境
Table 1　Environment of one node

CPU	Intel Xeon 6 cores 12 threads 2.40 GHz × 2
Memory	48 GB
HDD	SAS 15000 rpm 600 GB
OS	CentOS 6.3 64 bit
Gfarm	2.5.8.7
gfarm2fs	1.2.9.6
IOR	2.10.3

メモリ領域が確保される．確保されたメモリ領域は，ディスクリプタ情報ファイルとの紐付けが行われ，ロックとして利用される．マッピング情報の更新を行う直前に，更新を行うディスクリプタ情報ファイルと紐付けられたロックを獲得する．他のクライアントの操作により，既にロックされていた場合は，ロックが解除されるまでブロックされる．マッピング情報の更新後，更新の成功の有無に関わらず，更新を行っていたディスクリプタ情報ファイルと紐付けられたロックを解放する．

また，複数のクライアントが，同じ Gfarm 上のファイルを同時に開くことがある．このとき，ディスクリプタ情報ファイルの処理や，そのファイルに結びついているロックの初期化などが競合してしまうことがある．したがって，この競合は，ディスクリプタ情報ファイルの破壊を起こす可能性がある．

この競合を解決するために，我々は，最初に立ち上がった gfsd プロセスが `shmget` システムコールにより，ただ 1 つの共有メモリ領域を確保するようにした．gfsd は，ディスクリプタ情報ファイルを開く前に，その共有メモリ領域を利用してロックを獲得する．もしロックが獲得出来なかった場合は，ロックが解除されるまでブロックされる．そして，ディスクリプタ情報ファイルの処理や，それに結びついているロックの初期化などが完了したときに，ロックを解除する．

6. 実験

我々が実装した重複除外機構の性能を検証するために，Gfarm に CDC による重複除外機構を導入したものと，導入していないものに対し，IOR [20] によるマイクロベンチマークを実行した．メタデータサーバノード，I/O サーバノード，クライアントノードは共に 1 台ずつであり，QDR 4X の InfiniBand により相互に接続されている．ノード 1 台あたりの環境を表 1 に表す．本実験での平均チャンクサイズは 8 KB，最大チャンクサイズは 64 KB とした．

ベンチマークの方法を以下に示す．まず，IOR でファイルを作成し，書き込みを行わせた．そして，書き込みが行われたファイルを，IOR に読み込ませた．これを，シーケンシャルアクセスとランダムアクセスの双方について行わせた．IOR にオプションを与えることで，ファイルサイズを指定することができる．本実験では，ベンチマークに使うファイルサイズを 4 MB から 1.2 GB までとしている．

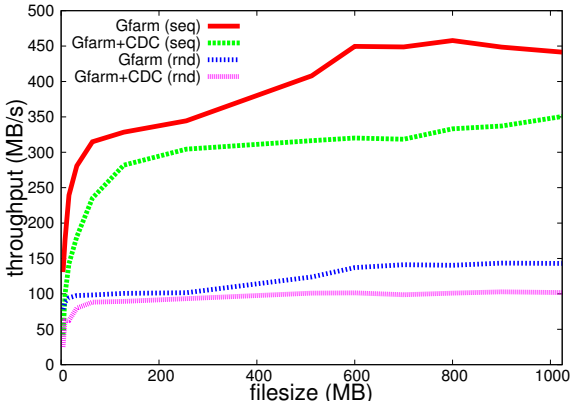

図 5　IOR によるファイル読み込みのベンチマーク結果
Fig. 5　The result of reading file benchmark with IOR

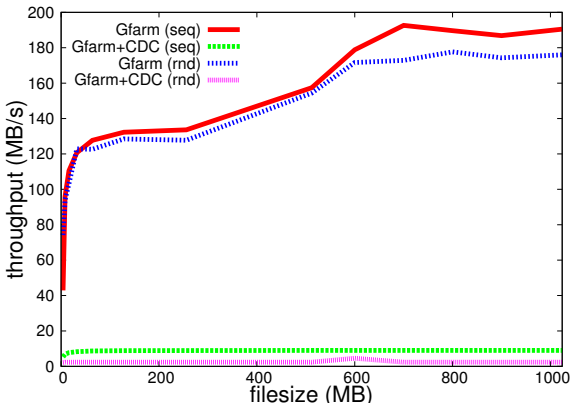

図 6　IOR によるファイル書き込みのベンチマーク結果
Fig. 6　The result of writing file benchmark with IOR

ベンチマークの結果を図 5 と図 6 に示す．まず，読み込みでは，ファイルサイズが十分に小さい時，シーケンシャルアクセス，ランダムアクセス共に，提案手法を導入したことによって，3 倍以上の時間がかかってしまった．これは，クライアントからファイルを開く要求がされた時に行われる，ディスクリプタ情報ファイルや，ロックの初期化によるものだと考えられる．しかし，ファイルサイズが十分に大きくなると，最悪でも 1.5 倍程度の差となった．これは，ファイルを開く要求がされた時の初期化処理にかかる時間が十分に短くなったからである．一方，ファイルへの書き込みについては，ファイルサイズの大小問わず，重複除外機構導入後のスループットは，重複除外機構を導入しなかった場合に比べて非常に低速となってしまった．特に，ランダムな書き込みの場合は，最大 70 倍以上の差となっている．この原因は現在調査中である．

7. 関連研究

SAR [21] は，SSD を利用することで重複除外機構の高速化を行ったものである．この研究で使っている重複除外機

構は，FSC によるファイル分割を行ったものであり，CDC を用いるファイル分割を行った本研究とは異なる．

マルチコア CPU [9], [16] や GPU [10], [11] を利用した並列化によって，CDC によるチャンク分割の速度を向上させる研究が行われている．また，マルチコア CPU と GPU の双方を利用した，CDC によるチャンク分割の並列高速化手法も提案されている [12], [13], [14], [15]．提案手法は，CDC によるファイル分割処理を非同期的に行わせる．これによって，チャンク分割処理を他のファイル操作とオーバラップさせることを可能にした．

村上らの研究 [16] は，本研究と同様に，Gfarm に対して CDC を用いた重複除外機構を導入している．この研究は CDC による重複除外処理をクライアントで行っている．そのため，計算ノードの計算資源を利用することになり，計算ノードの性能低下を招くことがある．提案手法は，クライアントではなく I/O サーバで重複除外処理を行っている．

P-Dedupe [9] は，重複除外機構の一部を非同期化させている．しかし，CDC によるチャンク分割処理は同期的に行っているため，結局，ファイル分割処理によるオーバーヘッドが発生する．提案手法では，その処理も非同期的に行わせることで，ファイル分割によるオーバーヘッドを削減した．

Meister ら [3] は，高性能計算での利用を前提としたストレージに対して，CDC 等による重複除外機構の導入を検討した．具体的には，実在の高性能計算用ストレージやファイルに対し，重複除外機構を導入した場合の重複除外率を測定した．しかし，実際のストレージに対する重複除外機構の導入及び評価を行っていない．本研究では，実際のストレージに対して重複除外機構を導入し，評価を行った．

本研究と同様に，Data Domain [7] や，i-Dedup [22] は，CDC によるオンライン重複除外機構を実現させている．しかし，これらは I/O サーバが 1 ノードのみである．本研究は，複数の I/O サーバの存在にも対応している．

HydraFS [8] は，HYDRAstor [23] という複数の CAS システムからなる分散ストレージでの利用を前提に，CDC によるオンライン重複除外機構を実装している．HydraFS では，I/O サーバが複数でも，CDC によるファイル分割処理を行うノードはただ 1 つである．本研究は，I/O サーバの数だけファイル分割処理を行う．

8. まとめと今後の課題

本研究では，高性能計算での利用を前提とし，Gfarm の I/O サーバに CDC による重複除外機構の導入を行った．大容量のファイルにおいて，CDC では，フィンガープリントの計算でオーバーヘッドが発生する．そこで我々はフィンガープリントの計算をクライアントの処理と非同期的に行う手法を提案した．

以下，今後の課題について述べる．提案手法では，ファイルの変更や削除によって，どのファイルからも参照されなくなったチャンクの対処を行っていない．FSC や CDC による重複除外機構で発生した，不要なチャンクの回収に関する研究がなされている [24], [25]．これらの研究を参考にした上で，Gfarm 上での実装を行う予定である．

提案手法では，I/O サーバ間での重複除外を行っていない．例えば，チャンク A，チャンク B，チャンク C からなるファイル X が，ある I/O サーバ S に記録されているとする．今，チャンク B，チャンク D，チャンク E からなるファイル Y が，あるクライアントから書き込まれるとする．ファイル Y が，I/O サーバ S に書き込まれる時は，重複しているチャンク B の内容は書き込まれずに，ファイル Y のデータがディスク上に書き込まれる．しかし，チャンク B が記録されていない I/O サーバ T にファイル Y が書き込まれようとしている．このとき，チャンク B は，I/O サーバ S にデータがあるにもかかわらず，I/O サーバ T にも書き込まれてしまう．このような場合でも，チャンク B の内容が I/O サーバ T に書き込まれないようにすることで，並列分散ファイルシステムの空き容量を増やすことができる．

また，Gfarm では I/O サーバ間でのファイルのレプリケーション機能がサポートされている．しかし本研究では，まだこの機能への対応を完了させていない．今後は，レプリケーション機能にも対応させる必要がある．

近年は，CDC をベースとした，重複除外機構用の新たなファイル分割手法が提案されている [26], [27], [28]．CDC による重複除外処理を，これらを使った重複除外処理に置き換えることで，並列分散ファイルシステムの空き容量を増やすことができる可能性がある．

そして，提案手法を導入した Gfarm において，書き込みが非常に低速であることが，実験より明らかとなった．書き込み処理の手法や実装について見直しを行う必要がある．

これらの改善を行った後，WRF [29]，Montage [30]，BLAST [31] といったようなデータインテンシブアプリケーションを対象とした性能評価を従来手法と比較する形で行いたい．

謝辞

本研究を行うにあたって，有益な助言をしていただいた筑波大学の建部修見准教授及び建部・川島研究室の方々に深く感謝する．本研究は，科学技術振興機構戦略的創造研究推進事業 (JST CREST) の研究課題「ポストペタスケールデータインテンシブサイエンスのためのシステムソフトウェア」の支援を受けている．

参考文献

[1] Tatebe, O., Hiraga, K. and Soda, N.: Gfarm Grid File System, *New Generation Computing*, Vol. 28, No. 3 (2010).
[2] Schwan, P.: Lustre: Building a File System for 1,000-node Clusters, *Linux Symposium '03* (2003).
[3] Meister, D., Kaiser, J., Brinkmann, A., Cortes, T., Kuhn, M. and Kunkel, J.: A study on data deduplication in HPC storage systems, *IEEE/ACM SC '12* (2012).
[4] Bonwick, J.: ZFS Deduplication (Jeff Bonwick's Blog), `https://blogs.oracle.com/bonwick/entry/zfs_dedup` (2009).
[5] Quinlan, S. and Dorward, S.: Venti: a new approach to archival storage, *USENIX FAST '02* (2002).
[6] Muthitacharoen, A., Chen, B. and Mazières, D.: A low-bandwidth network file system, *ACM SOSP '01* (2001).
[7] Zhu, B., Li, K. and Patterson, H.: Avoiding the Disk Bottleneck in the Data Domain Deduplication File System, *USENIX FAST '08* (2008).
[8] Ungureanu, C., Atkin, B., Aranya, A., Gokhale, S., Rago, S., Calkowski, G., Dubnicki, C. and Bohra, A.: HydraFS: a high-throughput file system for the HYDRAstor content-addressable storage system, *USENIX FAST '10* (2010).
[9] Xia, W., Jiang, H., Feng, D., Tian, L., Fu, M. and Wang, Z.: P-Dedupe: Exploiting Parallelism in Data Deduplication System, *IEEE NAS '12* (2012).
[10] Bhatotia, P., Rodrigues, R. and Verma, A.: Shredder: GPU-accelerated incremental storage and computation, *USENIX FAST '12* (2012).
[11] Kim, C., Park, K.-W. and Park, K. H.: GHOST: GPGPU-offloaded high performance storage I/O deduplication for primary storage system, *ACM PMAM '12* (2012).
[12] Ma, J., Zhao, B., Wang, G. and Liu, X.: Adaptive pipeline for deduplication, *IEEE MSST '12* (2012).
[13] Tang, Z. and Won, Y.: Multithread Content Based File Chunking System in CPU-GPGPU Heterogeneous Architecture, *IEEE CCP '11* (2011).
[14] 松宮　遼, 平井成海, 佐々木慎, 高橋一志, 大山恵弘：CPU-GPU アーキテクチャを用いた Content-Defined Chunking の実装と評価，情報処理学会 ComSys '13 (2013).
[15] Matsumiya, R., Takahashi, K., Oyama, Y. and Tatebe, O.: Content-Defined Chunking for CPU-GPU Heterogeneous Environments, *USENIX FAST '14 poster session* (2014).
[16] 村上じゅん, 石黒　駿, 大山恵弘：Content-Defined Chunking を用いた重複除外キャッシュ機構の実装と評価，情報処理学会研究報告ハイパフォーマンスコンピューティング，Vol. 2012-HPC-137, No. 4 (2012).
[17] Rabin, M. O.: *FINGERPRINTING BY RANDOM POLYNOMIALS*, Center for Research in Computing Techn., Harvard Univ. (1981).
[18] : Linux NFS faq, `http://nfs.sourceforge.net/`.
[19] : FUSE: Filesystem in Userspace, `http://fuse.sourceforge.net/`.
[20] : IOR HPC Benchmark — Free System Administration software downloads at SourceForge.net, `http://sourceforge.net/projects/ior-sio/`.
[21] Mao, B., Jiang, H., Wu, S., Fo, Y. and Tian, L.: SAR: SSD Assisted Restore Optimization for Deduplication-Based Storage Systems in the Cloud, *IEEE NAS '12* (2012).
[22] Srinivasan, K., Bisson, T., Goodson, G. and Voruganti, K.: iDedup: Latency-aware, Inline Data Deduplication for Primary Storage, *USENIX FAST '12* (2012).
[23] Dubnicki, C., Gryz, L., Heldt, L., Kaczmarczyk, M., Kilian, W., Strzelczak, P., Szczepkowski, J., Ungureanu, C. and Welnicki, M.: HYDRAstor: a Scalable Secondary Storage, *USENIX FAST '09* (2009).
[24] Strzelczak, P., Adamczyk, E., Herman-Izycka, U., Sakowicz, J., Slusarczyk, L., Wrona, J. and Dubnicki, C.: Concurrent Deletion in a Distributed Content-Addressable Storage System with Global Deduplication, *USENIX FAST '13* (2013).
[25] Fu, M., Feng, D., Hua, Y., He, X., Chen, Z., Xia, W., Huang, F. and Liu, Q.: Accelerating Restore and Garbage Collection in Deduplication-based Backup Systems via Exploiting Historical Information, *USENIX ATC '14* (2014).
[26] Kruus, E., Ungureanu, C. and Dubnicki, C.: Bimodal Content Defined Chunking for Backup Streams, *USENIX FAST '10* (2010).
[27] Romański, B., Heldt, L., Kilian, W., Lichota, K. and Dubnicki, C.: Anchor-driven subchunk deduplication, *ACM SYSTOR '11* (2011).
[28] Wei, J., Zhu, J. and Li, Y.: Multi-modal Content Defined Chunking for Data Deduplication, *USENIX FAST '14 poster session* (2014).
[29] : The Weather Research&Forecasting Model Website., `http://www.wrf-model.org/`.
[30] : Montage - Image Mosaic Software for Astronomers, `http://montage.ipac.caltech.edu/`.
[31] : BLAST: Basic Local Alignment Search Tool, `http://www.ncbi.nlm.nih.gov/BLAST/`.