Vol.2012-HPC-135 No.28
2012/8/2

ノンブロッキングトランザクションに基づく分散ファイルシステムのための分散メタデータサーバの設計と実装

平賀 弘平[1,3]　建部 修見[2,3]

概要：現在ハイパフォーマンスコンピューティングの分野で広く利用されている分散ファイルシステムは，メタデータ管理機構がボトルネックとなっており，多数の小さなファイルをスケーラブルに扱うのは困難な問題である．本稿では，分散ファイルシステムのメタデータ管理サーバを複数ノードへ分散化を行い，スケーラビリティを向上させる手法について提案する．ファイルシステムのツリー状のネームスペースを，分散ノードで効率的に管理するため，親 inode 番号と，ファイル名のペアをキーとした inode の分散格納形式を提案する．また，複数の inode エントリをトランザクショナルに更新するために，ノンブロッキングなソフトウェアトランザクショナルメモリのアルゴリズムをベースとした分散トランザクションを利用する．ノンブロッキングトランザクションは，マルチリーダ化と，共有ロックモードの追加を行い，サーバサイドでのトランザクションの実行によるリモート通信回数の削減を行った．性能評価では，88 クライアントから単一ディレクトリへ並列にファイル作成を行うワークロードにおいて，サーバ 8 台の利用により 62,000 ops/sec，サーバ 1 台の場合と比べて約 2.58 倍の性能向上が得られた．

1. はじめに

現在メモリに収まらない規模のデータ処理のため，ハイパフォーマンスコンピューティングの分野では，分散ファイルシステムが広く利用されている．分散ファイルシステムは多数の I/O ノードを用いて I/O 性能や信頼性を向上させている．大規模データを複数の I/O ノードに分散し，競合が発生しないように I/O 要求のコントロールを行い，局所性を考慮した実データ・タスクの配置により，I/O スループットを向上している．一方で，ファイルシステムはデータの格納場所の管理のため，inode と呼ばれるメタデータを保持している．ファイルシステムはツリー状のディレクトリネームスペースの提供や，ネームスペースと実データ格納場所のマッピング管理，ファイルの排他制御管理などを行う必要がある．これらの機能提供のため，現在 HPC の分野で広く利用されている分散ファイルシステムである PVFS [1] や Lustre [2] では，メタデータサーバを用いてメタデータを集中管理している．

現在の分散ファイルシステムは，I/O ノードを多数並べることにより，大容量のファイルを高スループットで処理することには長けているが，多数の小さなファイルをスケーラブルに扱うのは苦手としている．これは，メタデータ管理機構がスケーラブルでない為であると考えられる．メタデータは複雑なツリー状のネームスペースを扱っている為，単純には分散化できない上に，コンシステンシ維持のための同期や直列化のコストが大きくなるという問題がある．しかし，エクサフロップス級のスーパーコンピュータを支援するファイルシステム実現に向けて，今後ますます増加するファイルシステムクライアント数，I/O ノード数，IOPS をスケーラブルに処理するため，メタデータ管理機構の分散化によるスケールアウトは必要である．

我々は，ポストペタスケールをターゲットとした分散ファイルシステムのための，分散メタデータ管理システム，*PPMDS* を提案する．PPMDS は，ノンブロッキングなトランザクションを用いることで，グローバルなロックやオペレーションの直列化無しに，細粒度な同期によるメタデータ操作を行える．メタデータは，スケーラブルな分散 Key-Value store （以下 KVS と表す）上に保持される．KVS 上に，ツリー状のディレクトリネームスペースを管理するための inode データ構造を実装する．inode データ構造の工夫により，従来のファイルシステムが苦手としていた，同一ディレクトリへのファイルの並列作成，参照，

[1] 筑波大学大学院システム情報工学研究科
Graduate School of Systems and Information Engineering, University of Tsukuba
[2] 筑波大学システム情報系
Faculty of Engineering, Information and Systems, University of Tsukuba
[3] 独立行政法人科学技術振興機構 戦略的創造研究推進事業
JST CREST

削除に高い並列性を実現する．

本稿では，PPMDS のプロトタイプを実現した．PPMDS はファイルシステムのメタデータのみを管理するシステムで，ファイル，ディレクトリのメタデータの作成，参照，削除のみを行うことができる．

以下，第 2 章では，ファイルシステムの inode 構造を分散化する上での課題を述べた後，メタデータの分散化に取り組んでいる既存研究について述べる．第 3 章では，我々が提案する PPMDS の概要と，inode 分散格納形式，ノンブロッキングトランザクションについて述べた後，実装について述べる．第 4 章では，PPMDS の性能評価として，単一ディレクトリへの並列メタデータ操作のスケーラビリティと，PPMDS のサーバ台数を増やした場合のスケーラビリティ評価について述べる．第 5 章では，まとめと今後の課題を述べる．

2. 背景と関連研究

2.1 ローカルファイルシステムのディレクトリネームスペース管理

本節では，一般的なローカルファイルシステムがどのようにディレクトリネームスペースを管理しているかについて述べる．ext2，ext3 等の従来のローカルファイルシステムは，ファイルやディレクトリを，inode と呼ばれるメタデータと実データを保存するデータブロックで構成する．inode には，所有者，所有グループの情報や，最終更新時刻等のタイムスタンプ情報，パーミッション等のメタデータが保存されている．さらに実データが保管されているデータブロックのブロック番号が保管されている．データブロックが多数あり，ブロック番号の配列の数が足りない場合は，ブロック番号の配列を格納するデータブロックを用意して，2 段，3 段の間接参照によりそれを拡張する．

データブロックの中身は，その inode がファイルの場合は実データが保管される．ディレクトリの場合は，そのディレクトリに含まれる子エントリの名前と inode 番号（以下 ino と表す）のペアのリストが格納される．その他に，ディレクトリの lookup の高速化のため，ディレクトリのデータブロックには B-tree や hash を用いたインデックスを含めることもある．ext3 では，インデックスはディレクトリごとに作成される．

2.2 従来の inode データ構造の問題点

従来のローカルファイルシステムのメタデータ管理手法は，マシンローカルなブロックデバイスに最適化されたものであり，複数マシンによる分散メタデータ管理には適していない．ファイルシステムが肥大化してくるにつれて，inode ブロックやデータブロックの間接参照によるオーバヘッドが増加する．メタデータ分散化の際はできるだけ単純化するのが望ましいと考える．また，inode はファイルやディレクトリと 1 対 1 に対応するため，単純な構造で管理できるが，一方でネームスペースはディレクトリのデータブロック内のマッピングテーブルによってツリー構造が保持されるため，単純ではない．たとえば，あるディレクトリに多数のファイルを作成する場合，そのディレクトリのデータブロックにアクセスが集中し，並列性が阻害される．また，エントリ名と ino のマッピングテーブルが肥大化していく問題もある．分散メタデータサーバの高い並列性の実現には，このマッピングテーブルをうまく複数のサーバに分散化させた上で，同期や直列化等の競合が発生しないように設計する必要がある．

2.3 関連研究: 分散ファイルシステムのメタデータ管理

GIGA+ [3] は，単一ディレクトリ内のファイルを複数のサーバに分散するインデックスを提供するシステムで，既存のファイルシステムの補完を目的としている．GIGA+ はインクリメンタルなディレクトリ分割を行う．ディレクトリ内のファイル数がしきい値を超えると，それらをハッシュ関数で 2 つのグループに分割し，半分を別のサーバに移動させる．グループの分割方法は，ハッシュ値の空間を中央で前半，後半に分割して決定する．その後ファイル数が増えて再び分割しきい値を超えた場合，ハッシュ空間をさらに半分に分割し，後半を別のサーバに移動する．このように，ハッシュ空間を再帰的に分割していき，インクリメンタルにディレクトリエントリを分散する．ハッシュ空間の分割は，各サーバが個々に判断できるため同期や直列化を必要としない．また，ハッシュ空間をどのように分割したかという情報と，現在のサーバ一覧があれば，どのハッシュ空間がどのサーバに割り当てられているかを決定できる．クライアントは，ハッシュ空間がどのように分割されたのかという情報をサーバから取得し，更新があった場合はその情報をキャッシュするので，基本的には 1 ホップで目的のファイルを格納しているサーバに問い合わせられる．GIGA+は，少なくとも数百万のファイルを POSIX-compliant な単一ディレクトリに格納し，高いスケーラビリティと並列性を実現する．しかし，GIGA+ の提供する分散インデックススキームは，ディレクトリ分割中のファイルの移動は単純なコピーであるため，移動中の一貫性の問題や，失敗した際のアトミック性の保証は GIGA+を使うシステム側で保証しなければならない問題がある．

3. PPMDS

本章では，我々が提案する PPMDS の説明を行う．現在の並列ファイルシステムは，単一ディレクトリへの大量ファイルの同時作成等，ファイルシステム操作のスケーラビリティが十分でないという問題がある．我々はファイルシステムのメタデータ管理を行うサーバ（以下 MDS と表

す）を，複数ノードへ分散し，スケーラビリティを向上させる．

3.1 概要

PPMDS のメタデータ分散化アプローチの概要を述べる．まず，同一ディレクトリに大量のファイルを作成すると，ディレクトリエントリが肥大化し，性能が低下する問題に対処するため，親 ino とファイル名のペアをプライマリキーとしたデータ構造を用いる．また，メタデータの複数のエントリをトランザクショナルに更新する必要があるため，ノンブロッキングな Software Transactional Memory のアルゴリズムをベースとした，分散トランザクションを用いる．

3.2 メタデータ格納形式

従来のローカルファイルシステムで用いられている inode 管理のデータ構造は，マシンローカルなブロックデバイスに最適化されたものであり，分散ノード環境には適していない．そこで，まずは inode 構造を分散化させやすい形に変換する．

基本的に，MDS が管理すべきメタデータは以下の 2 種類に分類できる．

- 個々の inode エントリと 1 対 1 に対応付けられる情報. e.g. 実データ格納先情報，パーミッション，タイムスタンプ
- ツリー状のディレクトリネームスペース．自エントリの名前と，親ディレクトリ，子エントリへのマッピングを管理.

メタデータを分散化するにあたって，特に後者のツリー状のディレクトリネームスペースを管理するため，従来のローカルファイルシステムが行っているような，ディレクトリのデータブロックを介した間接参照が存在すると，参照先の管理が多段になり効率が悪くなると考えられる．そこで，提案手法では inode データ構造を単純な Key-Value 構造に変換した．Key-Value の各ペアは，個々の inode エントリを表す．inode エントリと 1 対 1 に対応付けられる情報は，すべて Value の中に格納される．Key には，（親 ino，エントリ名）という 2 つの情報のペアを格納する．inode エントリは，この 2 つの情報のペアによってシステムワイドにユニークに識別される．

ディレクトリネームスペースは，Key の情報と，範囲検索可能な Key-Value のインデックスによって表現する．

図 1，**図 2** に例を挙げる．図 1 は，ファイルルックアップの例を示す．/dir/file1 というファイルをルックアップする手順は，

(1) ルート inode エントリを読み込む．ルートの親は存在しないので，親 ino は 0 とする．
(2) ルートディレクトリのパーミッションをチェックし，ルックアップが許可されていることを確認する．また，ルートディレクトリの ino （= 1）を確認する．
(3) (1, “dir”) を key とした，/dir の inode エントリを読み込む. パーミッションのチェックと，/dir の ino （= 2）を確認する．
(4) (2, “file1”) を key とした，/dir/file1 の inode エントリを読み込む.

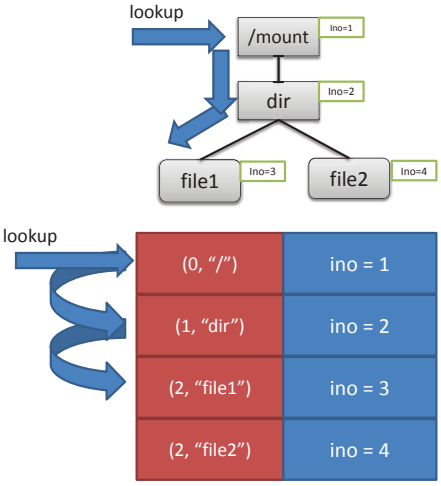

図 1 /dir/file1 のルックアップの例

図 2 /dir の readdir の例

図 2 は，readdir によるディレクトリエントリの獲得の例を示す．/dir の子 inode エントリ一覧を獲得する手順は，(1) /dir をルックアップし，/dir の ino（= 2）を獲得する．(2) Key を “2” で前方一致検索（範囲検索）する．

3.3 メタデータ分散化

本節では，3.2 節で述べたメタデータ格納形式を，複数のサーバに分散させる方法を説明する．**図 3** に，3 台の MDS に分散した場合の例を示す．PPMDS の各サーバは，それぞれ独立した Key-Value store を持つ．inode エントリを分散格納するために，PPMDS は，3.2 節で挙げたデータ以外に，inode エントリの分散先サーバを示す情報を格納する．分散先サーバ情報はディレクトリごとに作成し，inode エントリと同じ Key-Value store 上に格納する．Key はディレクトリの ino で，Value にはそのディレクトリの子エントリが分散格納されるサーバのサーバ識別子（sid）のリストを格納する．各 inode エントリは，親ディレクト

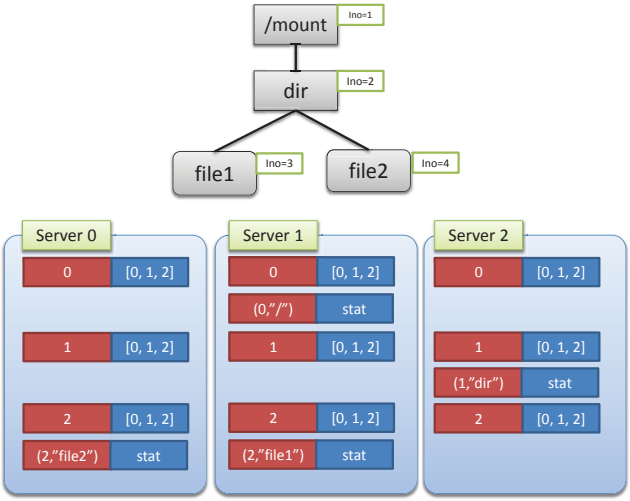

図 3 分散メタデータ格納形式

リの分散先サーバリストに従い，ハッシュ関数を利用した分散によって，格納先サーバの sid を決定する．擬似コードを以下に示す．

```
target_sid = sid_list[hash("name")%sid_list.size()]
```

"name" は inode エントリの名前を表す．名前から適当なハッシュ関数を用いてハッシュ値を求め，modulo 演算を用いて格納先 sid を決定する．分散先サーバ情報が格納される Key-Value ペアは，そのディレクトリを担当するすべてのサーバに複製して格納する．

inode エントリ格納先の決定のため，直接 inode エントリの Key とサーバ一覧を用いて決定するといった，より簡易な方法も考えられるが，提案手法でディレクトリごとに分散先リストを保持するのには，以下に挙げる理由がある．

- ディレクトリごとに分散先を柔軟に変化させたい場合が考えられる．多くのディレクトリは分割の必要のない小さなディレクトリなので，分割数を増やすとむしろオーバヘッドが増える場合がある．
- あるディレクトリの削除操作とディレクトリ内への新しいファイルの作成操作が衝突した場合に，オペレーションの衝突判定を行うために，この分散先サーバリストを利用する．詳しくは 3.4.4 節で述べる．
- 分散先を key のハッシュ値で決定する場合，ローカリティのコントロールができない．例えば，sid が 1 のサーバに保存されるユニークな key を生成したい場合があるが，一方向ハッシュを用いる場合，ハッシュ値から逆算して key を求めるのは困難である．詳しくは 3.4.2 節で述べる．

3.3.1 ファイル作成手順

具体的なファイル作成の手順を以下に示す．

(1) 事前準備として，クライアントは "/dir" をルックアップし，/dir の ino （= 2）を獲得する．
(2) クライアントは，どれかのサーバに対して，ファイル作成リクエストを送信する．
(3) サーバはファイル作成トランザクションを開始する．
(4) サーバは自身が持つ KVS に対して，Key = 2 で問い合わせを行い，分散先サーバリストを取得する．
(5) 獲得した分散先サーバリストと，作成するファイル名から，inode エントリの格納先を計算する．
(6) inode エントリ格納先サーバに対して，inode エントリの作成を行う．Key は，(2, "name")
(7) トランザクションをコミットする．

トランザクションについての詳細は 3.4 節で述べるが，この仕組みによって，複数のサーバにまたがる複数の Key-Value ペアをトランザクショナルに更新できる．(2) では，どのサーバに対してファイル作成指示を送っても良いが，実際に inode エントリを保存することになるサーバにリクエストを送信すれば，トランザクション中の通信回数を削減できる．本稿の評価で用いる実装では，クライアントサイドでも作成するファイル名からハッシュ値を計算し，ディレクトリの分散先サーバリストは現在存在する MDS のすべてを利用すると仮定して，事前に inode エントリが格納されるサーバを予測している．ファイル作成のトランザクションは，(4) でディレクトリの分散先サーバリストを読み込み，(6) で inode エントリの書き込みを行う．これらの読み込みと書きこみは，複数のクライアントが同一のディレクトリに対して並行にファイル作成トランザクションを発行しても衝突しないため，お互いの処理のために同期待ちは発生しない．

3.3.2 ディレクトリ作成手順

ディレクトリの作成を以下に示す．

(1) 事前準備として，クライアントは "/" をルックアップし，/の ino （= 1）を獲得する．
(2) クライアントは，どれかのサーバに対して，ディレクトリ作成リクエストを送信する．
(3) サーバはディレクトリ作成トランザクションを開始する．
(4) サーバは自身が持つ KVS に対して，Key = 1 で問い合わせを行い，分散先サーバリストを取得する．
(5) 獲得した分散先サーバリストと，作成するディレクトリ名から，inode エントリの格納先を計算する．
(6) inode エントリ格納先サーバに対して，inode エントリの作成を行う．Key は，(1, "name")
(7) 新たに作成するディレクトリの分散先サーバリストを決める．
(8) 分散先サーバリストを，関係するすべてのサーバに対して作成する．Key は，新たに作成したディレクトリ inode エントリの ino となる．
(9) トランザクションをコミットする．

ディレクトリの作成とファイルの作成の違いは，(8) で自身の分散先サーバリストを，関係するすべての MDS に書きこむ部分である．書き込みの回数がファイル作成と比

べて増えるが，ディレクトリの作成よりも，ファイルの作成の優先を選択した．(7) は，現在はその時点でサーバが知り得る限り最新の全 MDS のリストを格納している．最新の MDS のリストは，予め MDS の設定ファイルにスタティックに定義している．別の手段として，Chubby [4] や ZooKeeper [5] 等のシステムに，MDS のメンバー表を動的に保管する方法が挙げられる．(8) で作成する ino は，システム全体でユニークでなければならない．現在は，MDS 起動時に割り当てる sid と，サーバローカルカウンタの組み合わせで，ユニークな ino を作成している．

3.4 ノンブロッキングトランザクション

3.3.1 節，3.3.2 節で挙げた例のように，メタデータ操作の際，複数のサーバにまたがる Key-Value store の複数の Key-Value ペアを読み込み又は書き込み操作をする必要がある．これらのメタデータ操作は，それぞれ独立に行われ，他の操作の影響をうけないように排他的に行われる必要がある．また，操作はクライアントの故障などで予期せず中断する場合があるが，その場合もシステム全体を止まらないようにする．途中で失敗した操作はシステムに反映されないよう，一連の操作にはアトミック性が要求される．そこで提案手法では，Dynamic Software Transactional Memory [6] をベースとした，ノンブロッキングなソフトウェアトランザクションを，Key-Value store 上に実現する熊崎らの手法 [7] を PPMDS に実装し，一部拡張を行った．拡張部分を以下に示す．

- トランザクションを，Key-Value store のクライアント側だけでなく，サーバ上でも実行できるように変更し，トランザクション実行中に発生するリモート Key-Value ストアへのアクセス回数を削減した．
- トランザクション中の Key-Value ペアの読み込みを，他のトランザクションの書き込みと衝突しない限り，並列で行えるようにした．(マルチリーダ対応)
- トランザクション中に読み取った Key-Value ペアが，他のトランザクションによって書き換えられることを防ぐために，読み取り時に共有ロックモードを設定可能にした．

ノンブロッキングトランザクションの詳細な動作の仕組みは，文献 [7] を参照されたい．本節では，基本的な動作の説明と，我々が拡張した点について述べる．

3.4.1 概要

KVS 上の複数の Key-Value ペアに対する一連の読み込み，書き込みを 1 つの処理単位としたものがトランザクションである．トランザクションは，複数のクライアントから並行に発行され，各サーバはそれぞれのトランザクションを並行に処理するが，システム全体ではトランザクションがある逐次的な順番で処理されたかのように振る舞う．また，トランザクションは Serializable 分離レベルを満たす．この特性により，コミットに到達するトランザクションは，他の処理中のトランザクションから分離される．

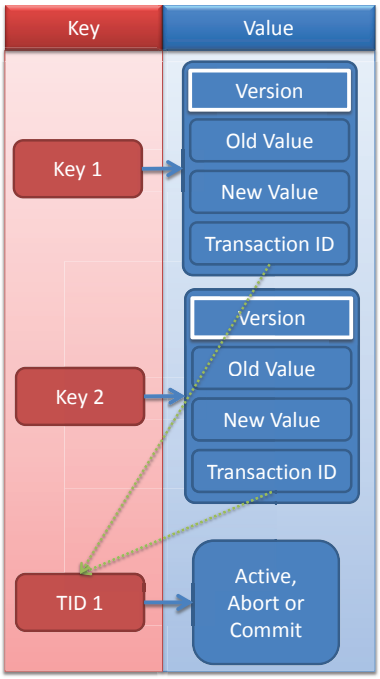

図 4 トランザクショナルキーバリューデータ構造

重要な特性として，本トランザクション実装は，obstruction-freedom である．これは，トランザクションが十分に長い時間単独で走ったら進行するという進行保証を表す．より強い進行保証である lock-freedom や wait-freedom と同様に，obstruction-freedom はノンブロッキングである．ノンブロッキングトランザクションは，障害や遅延が発生したトランザクションが，他のトランザクションの進行を邪魔しないという特性を持つ．

ノンブロッキングトランザクションをサポートするため，KVS は，Key-Value ペアを**図 4** に示す構造で保持する．各 Key-Value ペアは，Key，バージョン番号，Old Value，New Value，*Transaction ID* から構成される．Transaction ID は，その Key-Value ペアの *Owner* を表す．Transaction ID を Key として KVS に問い合わせれば，Owner の現在の状態を取得できる．Owner の状態は，*Committed*, *Aborted*, *Active* の三種類がある．Owner が現在 Committed であれば，既にその Key-Value ペアを最後に処理したトランザクションは処理を終えているため，New Value が最新の値となる．Owner が現在 Aborted であれば，最後に処理したトランザクションは処理を終えているが，処理は途中で中断している．トランザクションは，Key-Value ペアの Owner になる際に，その時点での最新の値を Old Value に退避するため，Owner の状態が Aborted である場合は Old Value がコミット済みの最新の値となる．Owner が Active であれば，現在トランザクションがその Key-Value ペアを処理中であることが分かる．

トランザクションは，Key-Value ペアに対して読み込み，書き込みを行う前に，Key-Value ペアを *Open* して，Owner になる必要がある．もし，Key-Value ペアの Owner の状態が Active であれば，現在他のトランザクションがその Key-Value ペアの Owner であり処理中なので，処理が終わるのを待つか，Owner 権を奪うかを選択しなければならない．Owner 権を奪うには，他のトランザクションの状態を Aborted へ変更する．トランザクションはこの判断をコンテンションマネージャと呼ばれるコンポーネントに依頼する．この競合解決方法は，obstruction-freedom を満たす範囲内ならば，どのようなアルゴリズムでも良い．現在は文献 [7] と同様にバックオフ待機アルゴリズムを用いている．これは，ある待ち時間上限の範囲内では，一定時間待機後に Owner の状態を再チェックし，Active なら前回の 2 倍の時間待機を繰り返し，待ち時間上限を超過したら Owner 権を奪うというアルゴリズムである．

トランザクションのコミット処理は，自身の Transaction ID に関連付けれている状態を，Active から Committed に変更することで成される．この変更は，Compare And Swap（CAS）によって，アトミックに行う．もし，この変更に失敗したら，トランザクションは他のトランザクションによって処理が中断されたと判断する．

3.4.2 トランザクションのサーバサイド実行

文献 [7] の提案は，標準的な memcached [8] 互換の KVS の利用を前提としている．KVS サーバ側は手を加えずにクライアント側のみでトランザクションを実現し，幅広い KVS サーバに対応している．我々はより高性能な MDS を目的としており，memcached との互換性は必要としていない．従って KVS サーバ側も変更を行い，トランザクションをサーバサイドで実行し，トランザクション中に発生するリモート KVS サーバへのアクセス回数を削減する．

クライアント側でトランザクションを実現する場合，トランザクション中で発生する Key-Value ペアのアクセスの都度，サーバと通信する必要がある．その上，トランザクションの開始と終了処理では，リモートサーバ上のトランザクションの Owner 状態の作成，更新をしなければならない．

トランザクションをサーバサイドで実行する利点として，トランザクション処理中の Key-Value ペアへのアクセスを，ローカルに処理できる場合がある．PPMDS のクライアントは，トランザクションを開始する時は単に PPMDS にトランザクションの実行を委譲する．例えばファイルを作成する場合は，親 ino とファイル名を PPMDS に伝えて，結果を待つ．PPMDS サーバは，トランザクションを開始する際，トランザクションの状態をローカル KVS 上に作成する．これは，3.3 節で挙げた，Key-Value ペアの格納先をハッシュではなく，sid の指定によって決定する方式の利点である．Key-Value ペアへの読み書きは，もし自分のサーバに保管されている Key-Value ペアであった場合は直接ローカルの KVS にアクセスし，リモートの場合のみ通信する．トランザクションのコミットは，必ずローカル KVS へのアクセスになる．

3.4.3 マルチリーダ化

トランザクショナル KVS では，Key-Value ペアへアクセスする前に Key-Value ペアを Open する．この時，別のトランザクションが処理中であったら，トランザクションは待たされるが，文献 [7] の Open は，書き込み，読み込みを区別しないため，読み込み Open の競合によって，無駄な待機が発生する．

提案手法では，文献 [6] を参考にして，Open を読み込み Open と書き込み Open を区別するように変更した．読み込み Open は Key-Value ペアの Owner 権の移動を行わない．代わりに，トランザクションは当該 Key-Value ペアの Key とバージョンを一時領域に保持しておく．そして，コミット処理の際にもう一度 Key-Value ペアを読み込み，バージョンを比較する．バージョンが一致しない場合，他のトランザクションが該当 Key-Value ストアを書き換えたことが分かるため，コミット処理を中断する．

3.4.4 共有ロックモードの追加

トランザクションのマルチリーダ化によって，同一の Key-Value ペアを読み込む複数のトランザクションの並行性を改善した．しかし，この最適化によって，トランザクション内に範囲問い合わせが含まれていた場合，期待した動作を達成できなくなった．マルチリーダ化によって，読み込み Open は，たとえ Key-Value ペアが既に他のトランザクションによって書き込み Open されていても，処理の終了を待つことなく続行し，競合の解決はコミット処理中に行われる．

この問題は，ディレクトリの削除トランザクションとそのディレクトリへのファイル作成トランザクションが同時に発生した場合において，以下のシナリオで発生する．

(1) ディレクトリ削除トランザクションが，ディレクトリの分散先サーバリストを削除する．尚，Key-Value ペアの削除は現在は単にヌルを書き込む．
(2) ディレクトリ削除トランザクションが，ターゲットディレクトリが空であること確認する．
(3) ファイル作成トランザクションが，ディレクトリの分散先サーバリストを取得する．ディレクトリ削除トランザクションが Owner だが，まだ Committed ではないため，コミット済みの最新の値である Old Value を読み込む．
(4) ファイル作成トランザクションが，ファイルの inode エントリを作成する．
(5) ファイル作成トランザクションが，コミット処理を行う．ディレクトリの分散先サーバリストのバージョンは読み込み Open した時点と変化していないため，コ

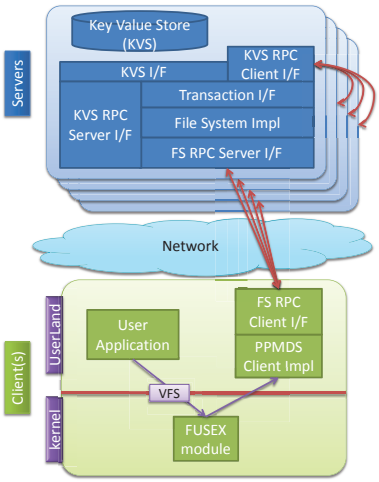

図 5 PPMDS アーキテクチャ概要

ミットは成功する．

(6) ディレクトリ削除トランザクションが，ディレクトリの inode エントリを削除する．

(7) ディレクトリ削除トランザクションが，コミット処理を行う．トランザクションの状態は Active のままなので，コミットは成功する．

このシナリオでは，ディレクトリの内容が空でなくなっているが，削除が成功してしまう．

この問題の解決のために，読み込み Open に共有ロックモードを追加する．共有ロックモードとは，読み取った Key-Value ペアが他のトランザクションによって更新されることを防ぎ，もし既に他のトランザクションによって書き込み Open されていた場合は，コンテンションマネージャによる競合解決を待つ機能である．

Key-Value ペアを共有ロックモードで読み込み Open する場合，Owner の状態をチェックし，Committed や Aborted であったら通常の読み込み Open と同様に最新のコミット済みの値を読んで，Key とバージョン番号を保持する．もし他のスレッドが Active だったら，コンテンションマネージャによる競合解決を待ち最初からやり直す．競合が解決されている場合は，Committed や Aborted に遷移しているが，再び競合が発生する可能性も考えられるので，その場合は競合解決を繰り返す．共有ロックモードで読み取った Key-Value ペアが，その後他のトランザクションによって書き込み Open を経て更新されるかもしれない．その場合は，他のトランザクションが書き込み Open に成功した時点で，Key-Value ペアのバージョン番号がインクリメントされるので，共有ロックモードで読み込み Open したトランザクションの方が，コミット処理中にバージョンの不一致を検出し Abort するため問題ない．

3.5 実装

図 5 に，PPMDS の概要を示す．PPMDS は，マルチマスタ型のサーバである．クライアントインターフェースは FUSE [9] を用いた POSIX-compliant なマウンタを実装した．

3.5.1 サーバ実装

PPMDS のサーバは，ユーザランドのデーモンとして実装した．各サーバはローカルに Kyoto Cabient [10] の Key-Value store を持つ．Key の前方一致検索が必要になるため，Kyoto Cabinet の TreeDB を使用した．トランザクションの状態や Value が保存される Key-Value ペアは前方一致検索は必要としないため，StashDB を利用した．どちらの KVS も，ロックの粒度は行レベルである．また，Key-Value ペアに対する任意の read-modify-write 操作をサポートしている．この機能を用いることで，トランザクションの実装に必要な CAS 等のアトミックオペレーションを実装できる．

クライアントや，サーバ間の通信には，MessagePack-RPC for C++ [11] を利用した．サーバのソフトウェアスタックは図 5 に示す．サーバとして外部に提供するインターフェースは，KVS にアクセスする機能と，ファイルシステム操作の機能である．

KVS の機能は，memcached が提供するような，get，set，gets，cas，add 等に加えて，いくつかの read-modify-write オペレーションを追加した．

uadd(value) ローカル KVS にまだ存在しないユニークな key を生成して，その key と value で保存するし，key をクライアントに返す．

replace(key, value) key が存在している場合のみ，value に更新する．

単一の PPMDS が提供する KVS 機能は，そのノードローカルであり，分散 KVS ではない．PPMDS のファイルシステムとしての機能を分散化しているのは，File System Impl の部分による．

File System の機能は，現在のところ，ファイルシステム初期化，ファイルの作成，参照，削除，ディレクトリの作成，参照，削除のみで，実データの書き込み等はサポートしていない．

サーバはノンブロッキングトランザクションを実装している．ターゲットとなる Key-Value ペアが，ローカルサーバにある場合，直接 Kyoto Cabinet を参照する．そうでない場合は，リモートプロシージャコールを経由してリモートサーバに問い合わせる．

3.5.2 クライアント実装

クライアントは，FUSE の低レベルインターフェスである，fuse_lowlevel_ops を用いて実装した．実装したコールバック関数を以下に挙げる．

- init
- lookup
- getattr

表 1　Client Claster のスペック

CPU	Xeon E5620 2.40 GHz
Sockets/node	2
Cores/socket	4 + Hyper-Threading
Memory	24GB
Num of Nodes	11
Kernel	Linux 3.0.0-12-server x86_64

表 2　Server Claster のスペック

CPU	Xeon E5620 2.40 GHz
Sockets/node	2
Cores/socket	4 + Hyper-Threading
Memory	24GB
Num of Nodes	14
Kernel	Linux 2.6.26-2-amd64 x86_64

- opendir
- readdir
- releasedir
- mkdir
- rmdir
- create
- unlink

PPMDS では，inode エントリを（親 ino, “名前”）で管理しているため，inode エントリの内容を取得するには，この 2 つの情報が必要となる．inode エントリの格納先を知るためにも，“名前” の情報は必要不可欠である．しかし，FUSE のインターフェースの getattr 関数は，自身の ino は取得できるものの，親 ino や，自身の名前を取得するインタフェースは用意されていない．そこで，自分自身の ino から，親 ino と自身の名前を取得する以下のインタフェースを FUSE に追加した．この機能拡張を行った FUSE を *FUSEX* と呼ぶ．

```
fuse_req_get_idpair(req, &pino, &name)
```

req は FUSE の低レベルインターフェースで使用する fuse_req_t である．呼び出しが成功した場合，pino には，親 ino が渡され，name にはエントリの名前が渡される．FUSEX は上記の関数を追加した FUSEX ユーザランドライブラリと，ライブラリに親 ino と “名前” を渡す機能を追加した FUSEX カーネルモジュールから構成される．

4. 性能評価

4.1 評価環境

評価は**表 1**，**表 2** に示すマシンで計測を行った．**図 6** に，評価環境全体の概略図を示す．ベンチマークは，メタデータサーバへのメタデータ操作のパフォーマンス測定のため，mdtest ベンチマークを利用した．mdtest は MPI を利用した HPC 向けの並列メタデータベンチマークテストで，指定したディレクトリ直下に，並列にファイルとディレクトリの create/stat/delete を行う．mdtest 1 プロセス

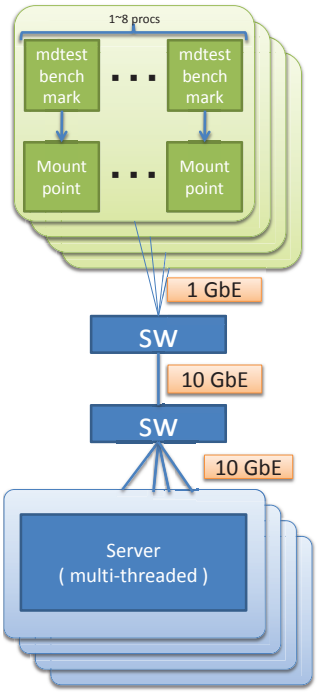

図 6　評価環境全体図

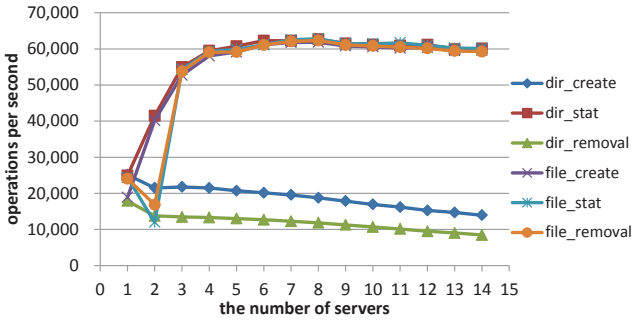

図 7　88 クライアントによる並行ファイルシステム操作

あたり 12,000 個，最大 88 プロセスから合計 1,056,000 個のファイルとディレクトリを作成する．

測定する際，mdtest の各 MPI プロセスごとに PPMDS のマウントポイントを作成した．単一マウントポイントを用いて複数クライアントから同一ディレクトリ内にファイル作成を行ったところ，同時に 1 つのリクエストしか実行されなかった為，マウントポイントを分割した．

4.2 単一ディレクトリへの並列メタデータ操作

複数のクライアントから単一ディレクトリに対して一斉にファイルシステム操作を行うワークロードについて，評価を行った．評価結果を**図 7** に示す．グラフの横軸はサーバ数を表し，縦軸は 1 秒あたりのオペレーション操作回数を示す．まず，ファイルの作成，参照，削除については，PPMDS のサーバ数を増加させるにつれて，62,000 ops/sec まで性能が上がった．サーバが 1 ノードの場合と比べて，約 2.58 倍の性能向上が得られた．ディレクトリ操作の性能は，参照についてはファイル操作と同様の性能向上が得ら

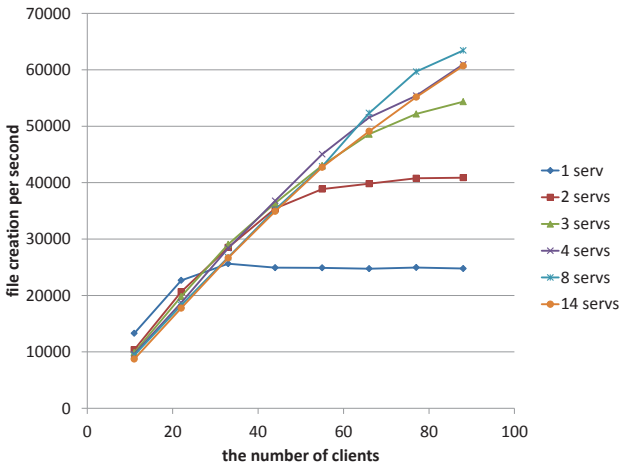

図 8 単一ディレクトリへの並行ファイル作成

れた．ディレクトリ作成と削除は，サーバ台数が増えるに従い性能が低下している．これは，ディレクトリの分散先サーバリストを全サーバに作成するコストが大きいためと考えられる．削除の場合は，ディレクトリが空であるかどうかのチェックが必要であるので，作成より性能は低下している．しかしながら，ディレクトリを大量に並行して作成したいという要求は稀であると考えるため，ディレクトリの作成，削除の性能はそれほど重要ではないと考える．

また，システム全体のスケールアウトが，サーバ 4 台でほぼ限界に達している．そこで，ボトルネックがサーバ側にあるのか，クライアント側にあるのか調査を行った．

4.3 分散メタデータサーバのスケーラビリティ

同様の評価環境で，単一ファイルへの並行ファイル作成のワークロードについて，クライアント台数とサーバ台数を変化させて評価を行った．**図 8** に評価結果を示す．グラフの横軸はクライアント数を表し，縦軸は 1 秒あたりのファイル作成数を示す．折れ線の種類はサーバ台数の違いを表す．サーバ台数とクライアント数を増加させるに従い，性能向上は得られているが，サーバ 4 台以上はほぼ同一の結果となった．一方，サーバ台数が 4 台未満の場合は，クライアント数を増やすことで，一定の性能限界を示した．この事から，システム全体の性能がクライアント側で律速されていることが分かる．より多くのノードを利用し，クライアント数を増加させる必要がある．また，クライアントは現在 FUSE を利用しているが，FUSE のオーバーヘッドにより性能が制限されている可能性がある．

5. まとめと今後の課題

本稿では，分散ファイルシステムのメタデータ管理サーバを複数ノードへ分散化を行い，スケーラビリティを向上させる手法について提案を行った．ファイルシステムのツリー状のネームスペースを，分散ノードで効率的に管理するため，親 inode 番号と，ファイル名のペアをキーとした inode の分散格納形式を提案した．また，複数の inode エントリをトランザクショナルに更新する必要があるため，既存研究で提案されているノンブロッキングなソフトウェアトランザクショナルメモリのアルゴリズムをベースとした分散トランザクションをメタデータサーバに実装した．ノンブロッキングトランザクションは，マルチリーダ化，共有ロックモードの追加，サーバサイドでのトランザクションの実行によるリモート通信回数の削減を行った．性能評価では，88 クライアントから単一ディレクトリへ並列にファイル作成を行うワークロードにおいて，最大 62,000 ops/sec，サーバ 1 台の性能と比較して，約 2.58 倍の性能向上が得られた．

今後の課題は，まずはより大規模なクライアントからの並列ファイル作成実験や，FUSEX を経由しない方法でファイルアクセスを行った場合の提案手法の有効性を検証したいと考えている．

謝辞 本研究の一部は，JST CREST「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」による．

参考文献

[1] PVFS. http://www.pvfs.org/.
[2] Braam, P. J.: Lustre. http://www.lustre.org/.
[3] Patil, S. and Gibson, G.: Scale and Concurrency of GIGA+ : File System Directories with Millions of Files, *Beaver*, No. February, pp. 177–190 (2011).
[4] Burrows, M.: The Chubby lock service for loosely-coupled distributed systems, *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, Berkeley, CA, USA, USENIX Association, pp. 335–350 (2006).
[5] Hunt, P., Konar, M., Junqueira, F. P. and Reed, B.: ZooKeeper: wait-free coordination for internet-scale systems, *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10, Berkeley, CA, USA, USENIX Association, pp. 11–11 (2010).
[6] Herlihy, M., Luchangco, V., Moir, M. and Scherer, III, W. N.: Software transactional memory for dynamic-sized data structures, *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, PODC '03, New York, NY, USA, ACM, pp. 92–101 (2003).
[7] 熊崎宏樹, 津邑公暁, 齋藤彰一, 松尾啓志: 分散キーバリューストアを対象としたオブストラクションフリートランザクションの実装. 情報処理学会研究報告, 2011-OS-118, pp. 1–7 (2011).
[8] Memcached. http://memcached.org/.
[9] FUSE: Filesystem in Userspace. http://fuse.sourceforge.net/.
[10] Hirabayashi, M.: Kyoto Cabinet. http://fallabs.com/kyotocabinet/.
[11] MessagePack. http://msgpack.org/.