CHFS アドホック並列分散ファイルシステムのアクセス性能の評価

建部 修見[1,a)]

概要：スーパコンピュータにおける I/O 性能の向上のため，計算ノードローカルのストレージを用いるアドホック並列分散ファイルシステムの研究開発が進んでいる．本研究では，CHFS アドホック並列分散ファイルシステムの性能評価をすすめ，ハッシング方式，並列 FIND の性能，シングルプロセスからのアクセス性能，単一共有ファイルのオープンコストについて評価を行い，それぞれの問題について CHFS の設計に基づき問題解決を図る．その結果，ハッシング方式をモジュラハッシングにすることにより負荷分散の問題が解決し性能も 1.4 倍から 2.5 倍となった．並列 FIND の性能は API を新設することによりスケーラブルな性能が達成できるようになった．シングルプロセスからのアクセスについては，非同期 RPC を導入することにより大幅な性能改善が可能であった．単一共有ファイルのオープンコストについては，各クライアントに対するコンタクトサーバを導入しキャッシュすることにより大幅なコスト削減が可能であった．

1. はじめに

スーパコンピュータにおける I/O 性能の向上のため，計算ノードローカルのストレージを用いるアドホック並列分散ファイルシステムの研究開発が進んでいる．アドホック並列分散ファイルシステムは，計算ノードが割当てられている間に構成する一時的な並列分散ファイルシステムである．計算ノードのローカルストレージは多くの場合占有利用が可能であり，他ユーザのアプリケーション等の影響を受けにくいため，安定して高い性能を得られるという特性を持っている．一方で，現在までに様々なアドホック並列分散ファイルシステムが提案されているものの，性能評価は十分ではない．

我々の研究グループでは CHFS アドホック並列分散ファイルシステムを提案している [9]．CHFS では，分散キーバリューストアを基盤とし，集中データ構造，逐次処理を排除することにより高いスケーラビリティの実現と，不揮発性メモリに対し永続インメモリキーバリューストアを用いることにより高い性能の実現を目指している．これまで，CHFS の比較的ナイーブな実装により IO500 ベンチマーク [2] を用いて評価を行い，プロセス毎ファイル（file-per-process）および単一共有ファイル（single-shared-file）アクセスパターンにおける並列アクセスバンド幅と，スモールファイル操作におけるメタデータ性能については一定の理解ができたが，より深く性能特性を理解するためには更なる性能評価が必要である．また IO500 ベンチマークにおける FIND ベンチマークの性能があまり高くないことについては，CHFS の設計とのトレードオフであると結論付けていたが，新たな API を設計し，並列 FIND のアルゴリズムを変えることにより，CHFS の設計を活かした高い性能を発揮することも可能であると考えられる．

本研究では，CHFS の性能評価をすすめることにより性能問題の解決を図り，性能向上を図る．CHFS はコンシステントハッシングを用いることでサーバの追加，脱退時のデータ移動を最小としていたが，コンシステントハッシングは負荷分散の問題がある．そのため，まずモジュラハッシングとの比較を行う．FIND ベンチマークについては，CHFS の設計を生かした新しい API を設計することにより，並列 FIND の性能向上を行う．また，大規模ファイルについてはチャンクサイズを大きくし，並列アクセスすることで全体のバンド幅の向上を狙っていたが，シングルプロセスからのアクセス性能については限定的なものとなっていた．これについて非同期アクセスを活用することにより解決を行う．さらに，単一共有ファイルのオープンのコストを解析し，性能向上のための解決策を提案する．

本研究の貢献は以下である．
- CHFS アドホック並列分散ファイルシステムについて性能評価を行う
- コンシステントハッシングとモジュラハッシングの比較を行う

1 筑波大学計算科学研究センター
a) tatebe@cs.tsukuba.ac.jp

ⓒ 2022 Information Processing Society of Japan

- 並列 FIND を効率的に実行するための API を設計し評価を行う
- 非同期アクセスによるシングルプロセスのアクセス性能の評価を行う
- 単一共有ファイルのオープンのコストを解析し解決策を提案する

2. 関連研究

アドホック並列分散ファイルシステムはさまざまな提案がある [1]．GekkoFS はメタデータ，ファイルデータを各ノードにモジュラハッシングで分散させた並列分散ファイルシステムである [13]．メタデータはサイズが小さいため RocksDB で管理し，ファイルデータはファイルシステムで管理している．システムアーキテクチャは CHFS と類似点が多いが，CHFS ではメタデータとファイルデータを分けず単一のキーバリューペアとして管理しており，オーバヘッドが小さい [9]．また，CHFS ではメタデータ，ファイルデータを永続インメモリキーバリューストアで管理しており，不揮発性メモリの性能を活かすことができる．

BeeOND は BeeGFS [10] を計算ノードのローカルストレージを用いてオンデマンドに配置することを可能にしたアドホック並列分散ファイルシステムである [11]．カーネルモジュールが実装されておりマウントすることが可能である．複数のメタデータサーバで分散処理も可能ではあるが，GekkoFS, CHFS に比べメタデータ性能は高くない [9]．

Gfarm/BB は Gfarm ファイルシステム [7] を軽量化し，アドホック並列分散ファイルシステムとして構成したものである [8]．自ノードのローカルストレージを優先して利用するため，特にファイル毎プロセスのアクセスパターンの書込みについては性能が高くなる．メタデータサーバはインメモリで構成されるものの，分散メタデータ構成をとることができないためメタデータ性能については限界がある．

UnifyFS [6] は BurstFS [14] をベースに設計されたアドホック並列分散ファイルシステムである．基本的には自ノードのローカルストレージにログを書込み，ログの配置情報等のメタデータはオーナノードで管理する．ファイルクローズ時にファイルの内容が共有されるセッションセマンティクスに加え，ファイルを読込み専用にすることにより読込性能の向上を図っているが，メタデータをオーナノードから取得する必要があり書込み性能ほどの性能は出ない．

DAOS [5] は不揮発性メモリとフラッシュストレージを前提として設計された分散オブジェクトストレージである．ファイルシステムのインターフェースも提供されている．ただし，計算ノードのローカルストレージの利用や一時的なオンデマンドの配置については想定されていない．

3. ハッシング方式

当初の CHFS の設計ではコンシステントハッシング [3] を用いることを想定していた．コンシステントハッシングはノードの参加，脱退時にデータ転送を最小にするという性質を持っているためである．一方で，アドホック並列分散ファイルシステムにおいては，計算ノードの障害等によりノードの参加, 脱退があるケースについては, 計算の続行のためにはアプリケーションの対応が必要であり，CHFS がサポートしたとしても利用する機会は限られると考えられる．そのため，よりハッシュのコストが軽く，負荷分散も均等になるモジュラハッシングとの比較を行った．

コンシステントハッシングはリングベースのものを用い，ノードをリング状にアドレスのハッシュ値に従って並べる．それぞれのノードが自身のハッシュ値からひとつ前のノードのハッシュ値までの区間を担当する．ただし，ノードのアドレスのハッシュ値をそのまま使ってしまうと，担当区間の範囲が非均等になってしまう問題がある．そのため，CHFS ではそれぞれのノードのアドレスに仮想サーバ名を追加したアドレスを用い，なるべく均等になるよう仮想サーバ名を設定していた．この方法では，担当区間の範囲がなるべく均等となるような仮想サーバ名の計算にコストがかかる問題があった．

一方，モジュラハッシングは，ノードをある方法で順序付し，ハッシュ値の剰余をもって担当ノードを割当てる．CHFS では，ファイルシステムのエントリのキーはパス名とチャンク番号であるが，モジュラハッシングではパス名のハッシュ値にチャンク番号を加えて剰余を計算する．この場合，単一共有ファイルについてはノード順に順番にストライピングされて配置されることとなる．また，ハッシュ値が均一であれば剰余も均一となり負荷分散の問題は起こらない．担当範囲の検索に比べ剰余計算は計算コストが軽いというメリットもある．モジュラハッシングでは，ノード数の増減によるデータ移動は多くなるが，ノード数の増減がないという仮定の下ではメリットは多い．

図 1 にコンシステントハッシングとモジュラハッシングによる IO500 ベンチマークのバンド幅スコアを示す．バンド幅スコアは，IOR [4] を用いた Easy と Hard のアクセスパターンでの並列書込，読込性能の幾何平均値である．筑波大学計算科学研究センターの Cygnus[12] を用い，ノード数を 16, 24, 32 ノードとしたときの性能を示している．IOR はノード毎のプロセス数（ppn）を 16 として実行している．CHFS/MH はモジュラハッシングを用いたものである．バンド幅はコンシステントハッシングを用いたものと比べ 1.6 倍から 1.7 倍の性能差となっている．バンド幅スコアの中では，IOR Easy については性能差は大きくないが，IOR Hard については 2.4 倍から 2.5 倍の性能差と

ⓒ 2022 Information Processing Society of Japan

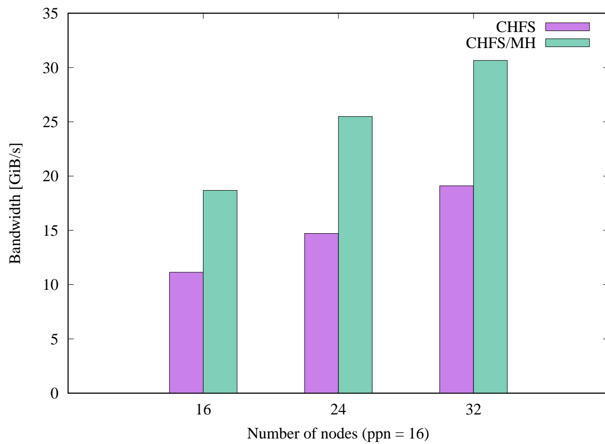

図 1 IO500 バンド幅スコアによるハッシング方式の比較.

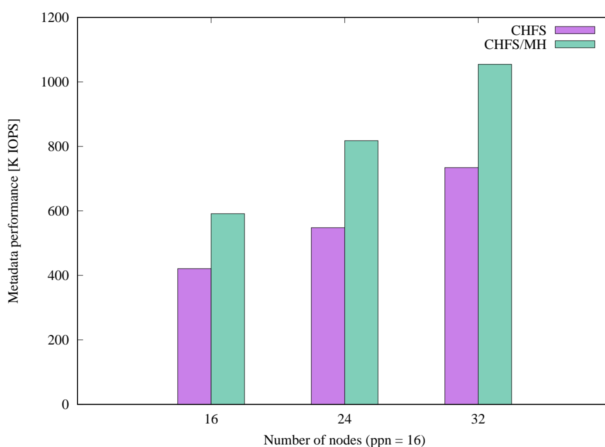

図 2 IO500 メタデータスコアによるハッシング方式の比較.

なっている．IOR Hard では小さいブロックサイズの単一共有ファイルへのアクセスであり，ブロックサイズをチャンクサイズとしているためチャンク数が多くなる．モジュラハッシングではパス名のハッシュ値にチャンク番号を加えた剰余をとっているためファイルチャンクがノードの順番に従いストライピングされて格納される．モジュラハッシングの効果というよりはファイルチャンクの分散方法の影響が大きいと考えられる．

図 2 に IO500 ベンチマークのメタデータスコアを示す．メタデータスコアは，MDtest [4] を用いた Easy と Hard のアクセスパターンにおけるメタデータ性能と並列 FIND の性能の幾何平均値である．MDtest は ppn を 16 として実行している．IO500 ベンチマークの本来のメタデータスコアには FIND の性能も含まれるが FIND についてはアクセスパターンが著しく異なるためこのグラフのスコアからは除いている．メタデータスコアについてはモジュラハッシングによる性能向上は 1.4 倍から 1.5 倍であった．メタデータのベンチマークでは全てのファイルのチャンク数は 1 となる．そのため，この性能差はコンシステントハッシングとモジュラハッシングのハッシュ計算コストと負荷分散が影響していると考えられる．

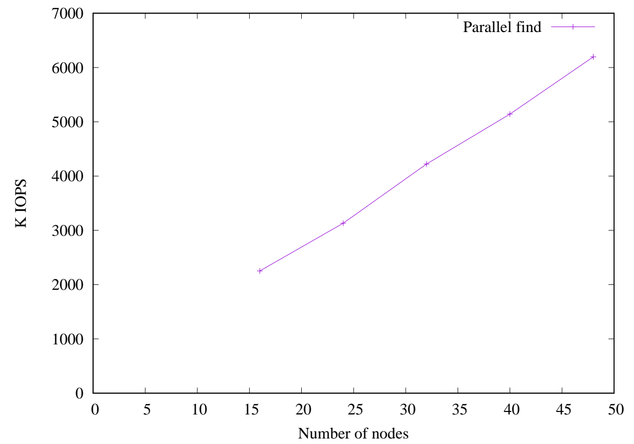

図 3 IO500 ベンチマークにおける FIND の性能.

4. 並列 FIND

IO500 ベンチマークでは並列 FIND のプログラムが準備されている．このプログラムは seekdir によりディレクトリエントリの位置を移動し並列に readdir を実行する．一方，CHFS における readdir は，各ノードのキーバリューストレージにおいて前方一致検索の結果を返すことにより実装されている．そのため，seekdir は高コストの処理となる．一方で，CHFS では各ノードのキーバリューストレージの前方一致検索についてはそれぞれ並列に実行することが可能である．従って，ノードを指定して readdir を実行する API を新設することにより，並列 FIND が可能となると考えられる．新設する API では，readdir API に対し，ノード番号を示す index を入力引数に追加する．この時，並列 FIND はノード数と同数のプロセスを起動し，各プロセスがランク番号の index を指定して FIND を実行する．

図 3 に IO500 ベンチマークの FIND の性能を示す．IO500 ベンチマークでは，IOR Easy/Hard, MDtest Easy/Hard で作成した全ファイルから，指定ファイルより新しく，ファイル名が*01*と一致し，ファイルサイズが 3,901 バイトのファイルの検索を行う．本評価ではノード数を 16 ノードから 48 ノードまで変え，毎秒の検索ファイル数を示している．グラフから分かるようにほぼリニアの性能向上を達成している．

5. シングルプロセスのアクセス性能

大規模ファイルについてはチャンクサイズを大きくし，並列アクセスすることで全体のバンド幅の向上を目指していた．その一方で，シングルプロセスからのアクセス性能については限定的なものとなっていた．

シングルプロセスからのアクセス性能を向上させるためには，非同期アクセスが有効である．非同期アクセスのために，read, write の RPC を非同期 RPC とし，RPC のリクエストと完了待ちに分ける．RPC のリクエストはブロッ

ⓒ 2022 Information Processing Society of Japan

```
for (i = 0; i < nchunks; ++i) {
  async_write_request(chunk[i], buf, s[i],
    &req[i]);
  buf += s[i];
}
ss = 0;
for (i = 0; i < nchunks; ++i) {
  async_write_wait(&s, &req[i]);
  ss += s;
}
return (ss);
```

図 4 非同期 RPC を用いた pwrite の疑似コード．

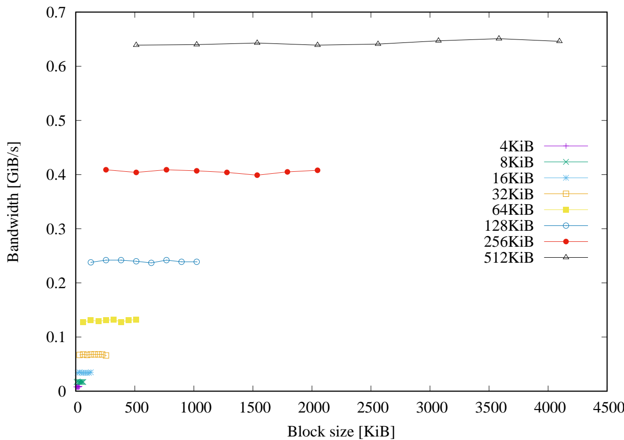

図 5 非同期 RPC を用いないシングルプロセスの書込み性能．

クせずに実行できるようにすることにより，複数の RPC リクエストの発行が可能となり，RPC の遅延時間の隠ぺいを計ることができる．

とはいえ，非同期アクセス API を準備したとしても，非同期アクセス API を用いたアプリケーションしか性能向上しない．そのため，pread, pwrite の API において，読込/書込チャンク数が複数ある場合に，非同期に複数チャンクに対しアクセスリクエストを発行することを検討する．これまで pread, pwrite では，チャンクを順番に処理し，成功したら次のチャンクを処理していたため，複数チャンクのアクセスであっても逐次的な処理であった．これに対し，複数のチャンクに対するアクセスのリクエストを全て発行し，発行した順に完了を待つようにする．

pwrite の疑似コードを**図 4** に示す．まず，チャンク数分のリクエストを発行する．`chunk[i]` は i 番目のチャンクであり，`s[i]` はそのチャンクに対する書込みサイズが入っている．`req[i]` にはのちに `async_write_wait` で完了を待つためのリクエストオブジェクトが入る．リクエストの発行が終了したら，順番に完了を待つ．`async_write_wait` は書込んだサイズ s を返す．i 番目のチャンクに書込んだサイズ `s[i]` と異なる場合は書込みエラーとなるが疑似コードからは省かれている．

図 5 に非同期 RPC を用いないときのシングルプロセスの書込み性能を示す．X 軸は pwrite に与えるサイズ（ブロックサイズ）を示し，Y 軸はバンド幅を示す．CHFS は 8 ノードで構成し，1 プロセスからの書込み性能を示している．チャンクサイズについては 4 KiB から 512 KiB までサイズを変化させている．結果から，バンド幅はチャンクサイズにより決定され，ブロックサイズを変えても変わらないことがわかる．

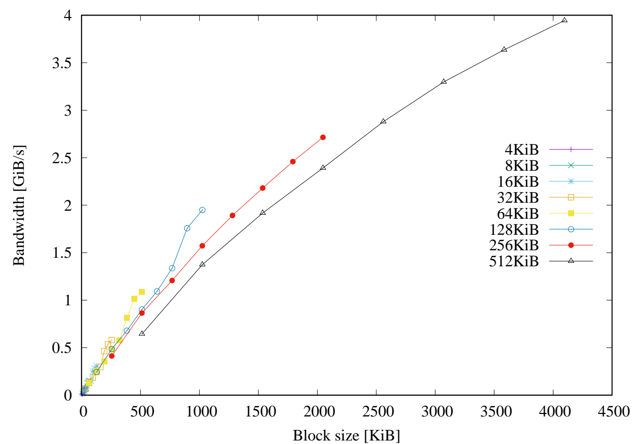

図 6 非同期 RPC を用いたシングルプロセスの書込み性能．

図 6 に非同期 RPC を用いたときのシングルプロセスの書込み性能を示す．同様の設定での性能評価であるが，こちらについてはブロックサイズを増やすことにより性能が向上している．ブロックサイズが大きいと，より多くのチャンクに対して非同期 RPC を発行することができるため，RPC の遅延時間の隠ぺいが可能となるためである．

6. 単一共有ファイルのオープン

単一共有ファイルのオープンでは，ファイルをオープンする全プロセスがオープンを発行する．このとき，CHFS ではオープンする全プロセスが一斉にファイルの先頭チャンクをアクセスしてしまう．例えば，32 ノードで各ノードに 16 プロセスが実行される場合，$32 \times 16 = 512$ クライアントからのアクセスが発生する．

通常，単一共有ファイルはファイルサイズが大きくなるため，オープンの時間はアクセス時間に比較すると大きなファクタとなるとは考えにくいが，プロセス数が多くなるとそれなりにオーバヘッドとなってしまう．また，全プロセスによるアクセス集中により想定を超えるコストとなることも考えられる．

基本的にそれらのアクセスは全プロセスで同一のものであるため，一プロセスが処理を行い，そのほかのプロセスはその結果を共有できればよい．MPI-IO における `MPI_Open_file` を用いれば，コミュニケータによりプロセスの集合が分かるためそのような最適化は可能である．しかしながら，一般の POSIX の場合はプロセスの集合がわからないためそのような最適化は難しい．

従って，CHFS のサーバ側でキャッシュすることを考え

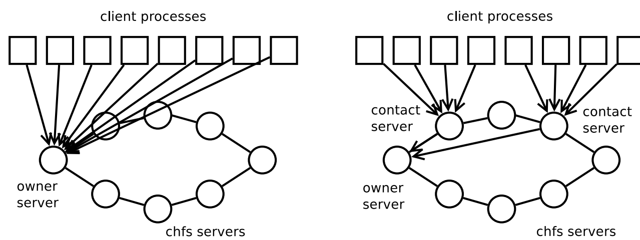

図 7 単一共有ファイルのオープン時の処理．左はナイーブな実装であり担当サーバに全クライアントが直接アクセスし，アクセス競合が発生している．右はサーバ側でのキャッシュを導入した場合であり，各クライアントは割当てられたコンタクトサーバにアクセスする．コンタクトサーバでは結果をキャッシュしクライアント間で共有する．

る．CHFS クライアントはファイルの先頭チャンクを直接アクセスしていたが，このままでは集中を回避することはできない（**図 7** の左図）．そのため，各 CHFS クライアントに対しコンタクトサーバとなる chfs サーバを割当て，各クライアントはこのコンタクトサーバにアクセスすることとする．chfs サーバは，自サーバの担当ではないキーについては担当サーバにリダイレクトするため，どのサーバにアクセスしても問題ないように設計されている．ここで，複数のクライアントプロセスでコンタクトサーバを共有し，結果をキャッシュすることにより集中を回避することができる（図 7 の右図）．一方で，共有するプロセス数が多すぎるとコンタクトサーバに対するアクセス集中問題が発生してしまうため，コンタクトサーバはある程度分散させる必要がある．

そのため，まずはなるべく集中回避を優先させ，コンタクトサーバは同一ノードのサーバとする．同一ノードにサーバが動作していない場合はなるべくネットワーク的に近いノードとし，このとき各コンタクトサーバに対しクライアント数が均一に割当てられることが望ましい．

コンタクトサーバでは，ファイルオープンのリクエストに対し，そのパス名に対する一度目のリクエストであればファイルの先頭チャンクをアクセスし，コンタクトサーバのキャッシュに登録する．二度目以降のアクセスについては，キャッシュにエントリが存在したらキャッシュの内容を返す．一方で，この方法では，一度しかアクセスしないプロセス毎ファイルのアクセスについてもキャッシュに登録してしまうこととなり，キャッシュのエントリに無駄が生じる．

プロセス毎ファイルと単一共有ファイルのオープンを区別するため，一度目のリクエストではキャッシュには登録せず，一時的に静的バッファに情報を保持する．二度目のリクエストがあれば単一共有ファイルのアクセスであると判断し，そこで初めてキャッシュに登録することとする．

図 8 に IO500 の IOR Hard Write ベンチマークにおける単一共有ファイルのオープン時間を示す．全てのケースで

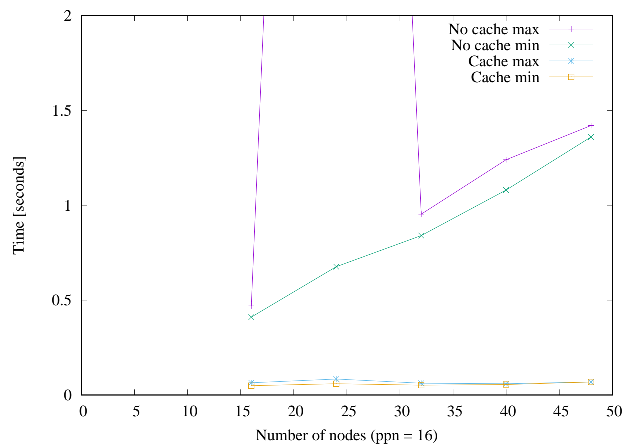

図 8 単一共有ファイルのオープン時間の最大値と最小値．

3 回実行を行い，最大値と最小値を示している．IOR Hard Write は，ブロックサイズは 47,008 バイトの単一共有ファイルに対し並列書込みを行うベンチマークである．

キャッシュを利用しない場合はノード数が増えるに従いオープン時間が増加しており，48 ノードでは 1.36 秒であった．最大時間に関しては 24 ノードでは 10.87 秒もかかっており，アクセス競合の状況により極端に時間がかかることがあることも分かる．

一方，キャッシュを利用する場合はノード数を増やしてもほとんどオープン時間は変わらず 0.06 秒から 0.08 秒の間であった．キャッシュの効果が大きいことがわかる．

7. まとめ

本研究では，CHFS アドホック並列分散ファイルシステムに対し，性能評価をすすめることによりさまざまな性能問題の解決を図った．ハッシング方式については，モジュラハッシングを用いることで特に単一共有ファイルのアクセス，メタデータアクセスについて 1.4 倍から 2.5 倍に性能向上することが分かった．また，readdir にノード番号を示す index を追加することにより効率的な並列 FIND の実装が可能となり，ノード数に対しスケーラブルな性能を達成することができた．シングルプロセスからのアクセス性能については，非同期 RPC を用いて性能向上を図った．その結果，ブロックサイズに応じて性能向上を達成することができた．単一共有ファイルのオープンについても，コンタクトサーバにおいてキャッシュすることにより性能改善ができ，ノード数によらず一定の時間となった．

今後，より大規模な計算機環境により，より多くのアプリケーションワークロードを用いて性能評価を進め，実運用につなげていきたい．本成果については，https://github.com/otatebe/chfs.git において公開している．

謝辞 本研究の一部は，JSPS 科研費 22H00509，筑波大学計算科学研究センターの学際共同利用プログラム

Vol.2022-HPC-185 No.31
2022/7/29

(Cygnus)，国立研究開発法人新エネルギー・産業技術総合開発機構（NEDO）の委託業務（JPNP16007）および富士通との共同研究の助成を受けたものです．

参考文献

[1] Brinkmann, A., Mohror, K., Yu, W., Carns, P., Cortes, T., Klasky, S. A., Miranda, A., Pfreundt, F.-J., Ross, R. B. and Vef, M.-A.: Ad Hoc File Systems for High-Performance Computing, *Journal of Computer Science and Technology*, Vol. 35, No. 1, pp. 4–26 (2020).

[2] IO500 Steering Committee: IO500, IO500 Foundation (online), available from ⟨http://www.io500.org/⟩ (accessed 2022/1/23).

[3] Karger, D., Lehman, E., Leighton, T., Levine, M., Lewin, D. and Panigrahy, R.: Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web, *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pp. 654–663 (1997).

[4] Loewe, W., McLarty, T. and Morrone, C.: HPC IO Benchmark Repository, (online), available from ⟨https://github.com/hpc/ior/⟩ (accessed 2022/6/24).

[5] Lofstead, J., Jimenez, I., Maltzahn, C., Koziol, Q., Bent, J. and Barton, E.: DAOS and Friends: A Proposal for an Exascale Storage System, *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 585–596 (online), DOI: 10.1109/SC.2016.49 (2016).

[6] Moody, A., Sikich, D., Bass, N., Brim, M. J., Stanavige, C., Sim, H., Moore, J., Hutter, T., Boehm, S., Mohror, K., Ivanov, D., Wang, T. and Steffen, C. P.: UnifyFS: A Distributed Burst Buffer File System, (online), available from ⟨https://github.com/LLNL/UnifyFS⟩ (accessed 2022/6/22).

[7] Tatebe, O., Hiraga, K. and Soda, N.: Gfarm Grid File System, *New Generation Computing*, Vol. 28, No. 3, pp. 257–275 (2010).

[8] Tatebe, O., Moriwake, S. and Oyama, Y.: Gfarm/BB — Gfarm File System for Node-Local Burst Buffer, *Journal of Computer Science and Technology*, Vol. 35, No. 1, pp. 61–71 (2020).

[9] Tatebe, O., Obata, K., Hiraga, K. and Ohtsuji, H.: CHFS: Parallel Consistent Hashing File System for Node-Local Persistent Memory, *International Conference on High Performance Computing in Asia-Pacific Region*, HPCAsia2022, New York, NY, USA, Association for Computing Machinery, p. 115–124 (online), DOI: 10.1145/3492805.3492807 (2022).

[10] ThinkParQ and Fraunhofer: BeeGFS, (online), available from ⟨https://www.beegfs.io/⟩ (accessed 2022/6/22).

[11] ThinkParQ and Fraunhofer: BeeOND: BeeGFS On Demand, (online), available from ⟨https://www.beegfs.io/wiki/BeeOND⟩ (accessed 2022/1/23).

[12] University of Tsukuba: Cygnus supercomputer, Center for Computational Sciences (online), available from ⟨https://www.ccs.tsukuba.ac.jp/eng/supercomputers/#Cygnus⟩ (accessed 2022/1/23).

[13] Vef, M.-A., Moti, N., Süß, T., Tacke, M., Tocci, T., Nou, R., Miranda, A., Cortes, T. and Brinkmann, A.: GekkoFS — A Temporary Burst Buffer File System for HPC Applications, *Journal of Computer Science and Technology*, Vol. 35, No. 1, pp. 72–91 (2020).

[14] Wang, T., Mohror, K., Moody, A., Sato, K. and Yu, W.: An Ephemeral Burst-Buffer File System for Scientific Applications, *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 807–818 (online), DOI: 10.1109/SC.2016.68 (2016).

ⓒ 2022 Information Processing Society of Japan