Vol.2019-HPC-168 No.22
2019/3/7

ノードローカルバーストバッファのための MPI-IO の設計

杉原 航平[1,a)]　建部 修見[2]

概要：HPC アプリケーションの IO アクセスを改善し，計算ジョブの実行効率を高める手法としてバーストバッファがある．バーストバッファはアプリケーションとファイルシステムの間に高速なバッファとして介在し，アプリケーションに代わって IO を行う．近年では NVMe SSD などの高速かつ平均故障間隔 (MTBF) が改善されたストレージの登場に伴い，計算ノードにもローカルストレージが導入されるようになった．バーストバッファにノードローカルストレージを取り入れて IO 性能を高める手法はこれまでにも提案されてきたが，本研究では，計算ノード上にローカルストレージをスプール領域とした Gfarm ファイルシステムを構築した上で，スケーラビリティを維持しながら IO 性能を向上するための並列 IO やそのインタフェースについて設計を行う．実験の結果，ノード数の増加に伴って IO 性能は線形にスケールすることを確認した．

1. はじめに

科学技術計算システムはエクサスケールを目指しており，スーパーコンピュータの計算ノードの性能向上はもちろん，構成要素となるストレージシステムの性能向上も必要不可欠となっている．近年の科学技術計算ではコア数の増加やアクセラレータの性能向上などから，計算ノードにおける計算性能が飛躍的な向上を遂げた一方で，ファイルシステムノードにおけるストレージ性能は計算ノードと並ぶ性能には至っていない．この乖離は計算システム全体のボトルネックとなるため，ストレージ性能を向上させることは計算システム全体の性能向上に対して寄与できると考えられる．

計算システムのストレージ性能向上に関する手法としてバーストバッファがある．バーストバッファはアプリケーションとファイルシステムの間に介在し，アプリケーションに対して高速で巨大な IO バッファを提供することで性能を高める．バーストバッファが配置される方式としては，専用の IO ノードとして動作するものや計算ノード上で動作するものがある．前者はレイテンシが小さく高速なストレージを大規模に並列配置したものを計算ノードやファイルシステムノードと独立した高速なファイルシステムとして構築することで，IO の性能を高める [2], [3]．一方で後者はファイルシステムとして動作するもの [1], [7], [11] や，ファイルシステムとアプリケーションの中間レイヤで動作する IO ライブラリに組み込んで提供されるもの [5], [12] の二種類に分類され，これらは下位レイヤのファイルシステムやディスクに対する IO のアクセスパターンを置き換えることで IO の効率化を行う．

近年のストレージ技術においては応答性能の向上による高速化にとどまらず，平均故障間隔 (MTBF) が改善されたことによって保守コストも削減された．これにより，従来は主記憶装置と演算装置のみの構成が主流だった計算ノードにも NVMe SSD などの大容量でレイテンシの小さいストレージが導入されるようになった．ノードローカルストレージをバーストバッファに適用し，かつ高いスケーラビリティを実現するには，実行されるアプリケーションプロセスの IO リクエストを精査し，ローカルストレージ間で IO リクエストを適切に分散する必要がある．本研究では，バーストバッファに NVMe SSD などの高速な計算ノードローカルストレージを適用するための，アプリケーションとバーストバッファ間の効率の良い並列 IO アクセス方式とそのインタフェースについて設計を行う．

2. アクセスパターン

ファイルに対して発生する IO アクセスを種別ごとに整理したものをアクセスパターンという．アクセスパターンを**図 1** に示す．アクセスパターンには，N 個のプロセスが N 個のファイルに対して別々に IO アクセスを行う N-N パターンと，N 個のプロセスが単一のファイルに対して IO アクセスを行う N-1 パターンの二種類に大別され，さらに N-1 パターンは N-1 Segmented パターンと N-1 Strided パターンの二種類に分類される．N-1 Segmented パターンは

[1] 筑波大学情報学群情報科学類
[2] 筑波大学計算科学研究センター
[a)] sugihara@hpcs.cs.tsukuba.ac.jp

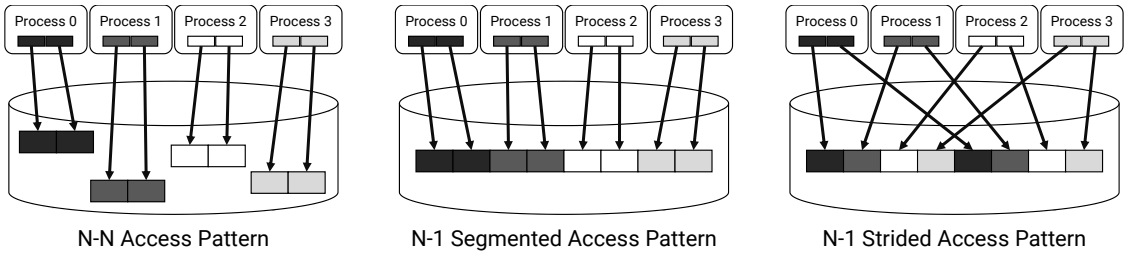

図 1 アクセスパターンの分類. ([1] より要約)

プロセスごとに連続で同一サイズの領域にアクセスを行う場合のことを指し，N-1 Strided パターンはプロセスごとに非連続な複数領域にアクセスを行う場合のことを指す．HPC アプリケーションにおいては N-N アクセスパターンがよく用いられてきたが，ノード数やプロセス数が増えた場合にメタデータ性能がボトルネックとなるため，複数のプロセスでファイルをまとめることが一般的となった．例えば，HPC アプリケーションにおけるデータのシリアライゼーションライブラリとして広く用いられる netCDF や HDF5 を利用して計算結果をファイルに出力する場合も，多くの場合で N-1 アクセスが発生する．しかしながら，多くのファイルシステムでは N-1 アクセスパターンの場合に IO 性能はうまくスケールしないため，アプリケーションとファイルシステムの間で IO アクセスを効率化するなどのさまざまな工夫が用いられてきた．

3. 関連研究

HPC アプリケーションとファイルシステム間の中間レイヤとして動作し，ファイルの配置方法や IO リクエストに手を加えることでアプリケーションの IO アクセスを効率化する手法は，バーストバッファやそれ以外にもこれまで数多く提案されてきた．

PLFS (Parallel Log Structured File System) [1] は，計算ジョブのチェックポイント書き込みにおける IO アクセスを再配置して効率化を図る手法である．計算ジョブにおけるチェックポイントとは，計算ノードの障害などによるシステムダウンに備え，計算の途中経過などのアプリケーションの状態をファイルシステムに書き出す操作である．PLFS はアプリケーションの IO をハンドリングし，アプリケーションが単一ファイルとして書き込んだものを内部で分割し，オフセットなどのメタデータとともに配置する．PLFS は FUSE を利用して POSIX ファイルシステムとして実装される．

Two-Phase I/O [4] は，複数プロセスがファイルに対して非連続にアクセスする場合の IO アクセスの効率化手法で，代表プロセスがファイルの連続する領域をバッファに読み，それらを複数プロセス間で交換することによって IO アクセスを効率化する．Thakur ら [10] は，これを MPI-IO ライブラリの実装のひとつである ROMIO に実装した．

木村ら [12] は FileView と呼ばれるデータ構造を利用してファイルを分割する手法を提案した．FileView とは MPI-IO で定義されるデータ構造で，複数プロセスが単一のファイルに対してアクセスする場合に，ファイルがアクセスする領域を定義する．FileView の情報はファイルシステムにメタデータファイルとして保存され，アプリケーション終了後であってもメタデータを参照することで POSIX アプリケーションからの読み書きを可能にする．

これらの手法はアプリケーションと同じ計算ノード上で動作するものや，計算ノードと独立した専用の IO ノードとして実装されるものがある．また，IO インタフェースに着目すると，FUSE (Filesystem in Userspace) などによって POSIX ファイルシステムとして実装されるものや，ファイルシステムのユーザライブラリとして提供されるもの，MPI-IO ライブラリとして実装されるものがある．

FUSE を利用した方式では API を POSIX の体系によって隠蔽することで高い移植性を提供することができる一方，POSIX API ではアプリケーションに特化した最適化を行うことは難しい．ファイルシステムのユーザライブラリとして実装する方式は，独自の API 体系の定義や拡張によってアプリケーションに特化した最適化を施せる反面，設計や移植性に関して注意を払う必要がある．MPI-IO ライブラリとして実装する方式では，MPI や MPI-IO で定義されるコミュニケータやヒントなどの情報を活用することで MPI アプリケーションに特化した最適化を行えるだけでなく，下位のファイルシステムの独自 API を MPI-IO の関数として隠蔽できるため高い可搬性と柔軟な拡張性を両立することができる．

4. ノードローカルバーストバッファの設計

本章ではノードローカルストレージを活用したバーストバッファの設計について述べる．ノードローカルストレージを性能向上に活かすためには，プロセスが実行されるノードのローカルストレージに対して IO リクエストを発行することが必用であると考えられる．しかしながら，HPC におけるアプリケーションのアクセスパターンによっては，ノードローカルストレージを利用すると負荷が単一ノードに集中する場合があるため，ファイル配置の方法を

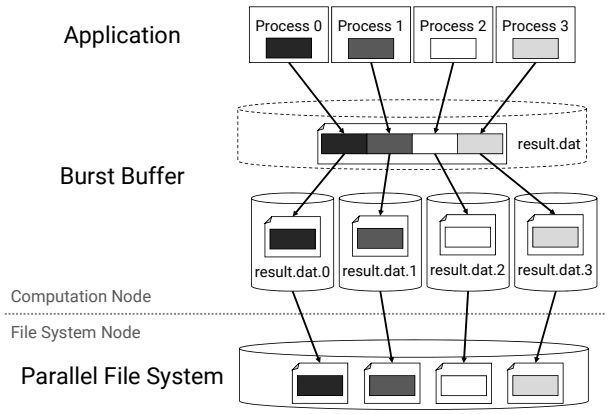

図 2　ファイル分割による N-1 Segmented アクセスパターンの解消

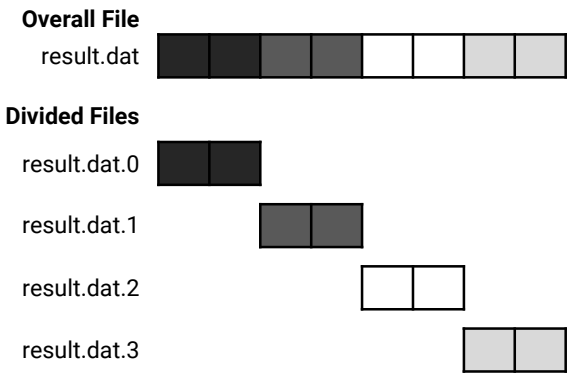

図 3　Resize がない場合の N-1 Segmented パターンの分割結果

工夫して負荷を分散する必要がある．本研究では，ファイルを分割して配置することによってこれの解消を図る．

4.1　プロセスランクに基づいたファイルの分割

バーストバッファでノードローカルストレージを利用するにあたっては，プロセスのローカリティに基づいてファイルの粒度を調節する必要がある．ノードローカルで完結できる IO リクエストを全てローカルで済ませ，無駄な通信を発生させないようにすることで，オーバヘッドを最小限にできると考えられる．

本研究では**図 2** に示すように，プロセスランクに基づいてファイルを分割する方式を提案する．バーストバッファ対するファイルアクセスは，全てプロセスランクごと暗黙に分割された上で行われる．バーストバッファはプロセスランクごとに異なる名前をつけてファイルを分割し，これを中間表現としてノードローカルストレージや下位の並列ファイルシステムに配置する．また，ここでの分割とは，ファイルを意味のある単位で分けることを指し，ストライピングのようにブロックごとに散布することとは異なる．本手法は N-1 Segmented アクセスパターンと N-1 Strided パターンどちらにでも適用することができるが，ファイル生成時の Resize 呼び出しの有無によって統合時のオフセット計算方法を切り替える必要がある．オフセット情報を分割されたファイルのメタデータとして保存すれば，アプリケーションの終了後であっても読み書きが可能になる．**図 3** と**図 4** に示すのは，Resize がない場合の N-1 アクセスパターンに対して本手法を適用した場合の分割結果である．分割後のファイルは，プロセスが最初に書き込んだオフセットから生成される．N-1 Strided の場合，プロセスが書き込まない領域はシークされ，その部分は `NULL` データとなる．一方で**図 5** に示すのは Resize がある場合に生成されるファイルである．HDF5 でファイルを生成する場合がこの方式にあたり，全てのプロセスが同じファイルサイズで Resize をコールする．このケースでは分割された全てのファイルの開始オフセットは 0 になる．この手法は，複数のプロセスがファイルの同じ箇所に対して書き込む場合には適用できないことに注意が必要である．

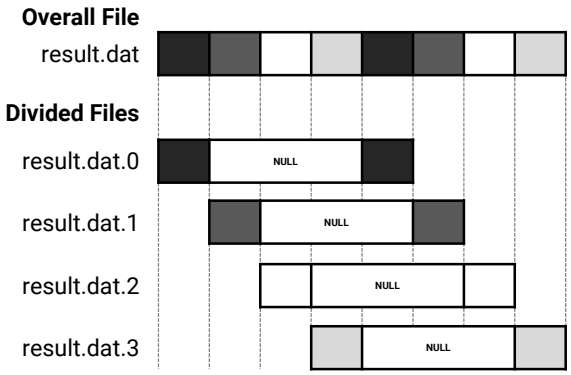

図 4　Resize がない場合の N-1 Strided パターンの分割結果

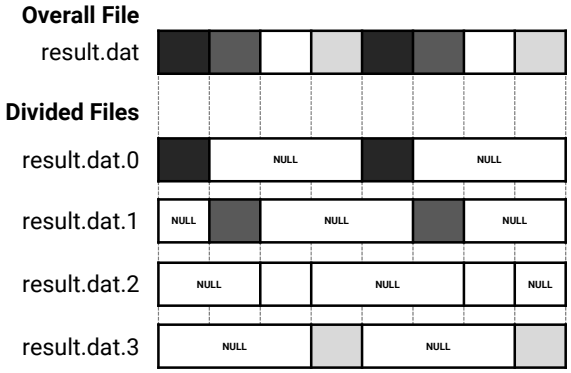

図 5　Resize がある場合の N-1 Strided パターンの分割結果

4.2　Gfarm ファイルシステムの利用

本研究では Gfarm ファイルシステム [8] を中間表現の格納先として利用する．Gfarm は他のファイルシステムと

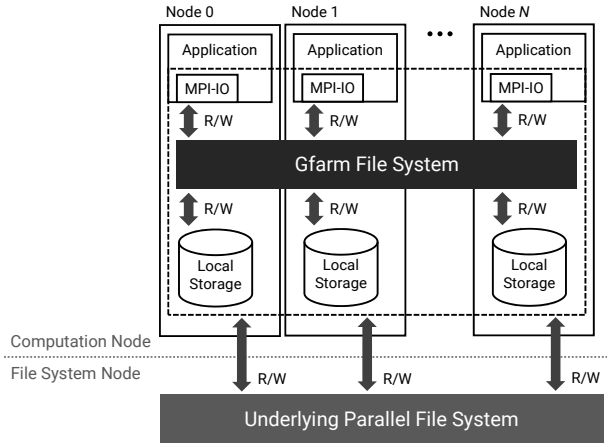

図 6　提案手法の概要.

異なりストライピングを行わない設計思想であり，ファイルを意味のある単位で分割し，ローカリティを考慮した配置やプログラミングを容易にする．バーストバッファの実装においては，ファイル IO のハンドリングやバッファの割り当てだけにとどまらず，ノード間の通信やメタデータ管理など，多くの要素を実装する必要がある．これらの要素は並列ファイルシステムがもつ機能と同じであるため，Gfarm ファイルシステムを内包することでバーストバッファの実装を行うことができると考えられる．

4.3　MPI-IO による実装

本研究では中間表現の格納先として Gfarm ファイルシステムを利用するが，バーストバッファのインタフェースをどのようにしてアプリケーションから利用するかについても検討する必要がある．本研究では，提案手法を MPI-IO ライブラリの実装のひとつである ROMIO に対して実装する．提案手法を MPI-IO ライブラリの実装中に導入することで，MPI-IO に実装された Collective IO などの最適化手法を併用して IO 性能を向上できるだけでなく，既存のアプリケーションを変更することなく本手法を適用することができる．加えて MPI ではアプリケーションに関する情報をヒントというデータ構造を用いて IO レイヤに伝達することができ，アプリケーションに特化した最適化を行うことができる．

図 6 は提案手法の概要で，破線部が今回設計した並列 IO に対応する．アプリケーションは MPI-IO を通してファイルシステムの代わりにバーストバッファに IO リクエストを発行する．バーストバッファの内部では Gfarm がファイルの情報を管理し，中間表現に基づいてファイルをローカルストレージに書き込む．通常，バーストバッファに書き込まれた内容は下位のファイルシステムノードにも書き込まれるが，この部分に関しては今回は未実装である．

ROMIO は実装に ADIO (Abstract-Device Interface for I/O) [9] という機構を持ち，ファイルシステムをドライバとして抽象化する．提案手法の実装の流れとしてはまず Gfarm を ADIO ドライバとして実装し，次に Gfarm ドライバに提案手法であるファイル分割を導入する．

4.3.1　Gfarm ドライバの実装

Gfarm ドライバの実装は，以下の ADIO 関数で Gfarm のファイル操作関数をラップすることによって行う．

- `ADIOI_GFARM_Open`
- `ADIOI_GFARM_ReadContig`
- `ADIOI_GFARM_WriteContig`
- `ADIOI_GFARM_Fcntl`
- `ADIOI_GFARM_Close`
- `ADIOI_GFARM_Flush`
- `ADIOI_GFARM_Resize`
- `ADIOI_GFARM_Delete`

ここで，関数の `ADIOI_GFARM_` という prefix の意味は ADIO の内部 (internal) で実装される Gfarm 関数であることを示している．`Open` 関数はファイルを開いて MPI-IO のディスクリプタを生成して返す関数である．`ReadContig` と `WriteContig` 関数はそれぞれ，ファイルディスクリプタに対して連続に (contiguous) 読み書きを行う関数である．`Fcntl` 関数はファイルサイズなどのファイルの情報を返却し，`Close` 関数はファイルディスクリプタを閉じる．`Flush` 関数はファイルのバッファをフラッシュする関数で，`Resize` 関数はファイルのリサイズを行う．`Delete` 関数はファイルの削除を行う関数である．

4.4　ファイル分割手法の適用

次に，Gfarm ドライバにファイル分割を導入する．ファイル分割と，分割したファイルの操作は以下の関数の動作を変更することによって実装される．

- `ADIOI_GFARM_Open`
- `ADIOI_GFARM_ReadContig`
- `ADIOI_GFARM_WriteContig`
- `ADIOI_GFARM_SeekIndividual`
- `ADIOI_GFARM_Fcntl`
- `ADIOI_GFARM_Delete`

ここでの主な変更点は，`Open` 関数内でファイル分割を行う点である．その操作に対応して `Fcntl` では複数に分割されたファイルに対して分割前のメタデータを正しく返却し，`Delete` では分割されたファイルを一括で削除する必要がある．Read や Write に関係する関数は，独立ファイルポインタを利用する MPI-IO 関数を使用する場合は手を加える必要はないが，共有ファイルポインタとオフセットを明示的に操作する MPI-IO 関数を用いる場合については別途オフセットの計算ルーチンを実装する必要がある．今回は独立ファイルポインタのみで実験を行うため，それ以外の方式については実装していない．

図 7 に示すのは `Open` 関数にファイル分割を適用した例

である．関数の引数として渡される fd は ADIO におけるファイルオブジェクトであり，fd->filename は MPI-IO におけるファイルパスの書式である <prefix>:<filename> の後半部分に対応する．ファイル分割は，プロセスランクごとにオリジナルのファイル名 filename を別名 fn として読み替えることによって行う．ここで，gfs_pio_open(), gfs_pio_create() はそれぞれ Gfarm のライブラリ関数で，ファイルのオープンと作成を行う．Open 関数が呼ばれたときはアクセスモードを確認し，O_CREAT フラグが定義されていた場合のみファイルを作成する．ファイル作成時は，分割後のファイル fn だけでなく，元のファイル名 filename と同じ名前のファイルも作成し，複数プロセス間で O_CREAT 付きで Open が呼ばれたことを共有する．O_CREAT 無しで Open が呼ばれた場合でも，ファイル f の存在を確認する関数 isExist(f) の結果が真であれば，アクセスモードに関係なくファイルを作成する．これにより，**図 8** のような代表プロセスのみがファイルを作成する場合でもうまくこの手法を適用できる．isExist() は全プロセスがファイルを開く度に呼ばれる可能性があるが，内容は軽いメタデータ操作であるためスケーラビリティに大きな影響を与えることはないと考えられる．Gfarm ライブラリによるファイル作成では，ファイル配置先ノードをユーザが直接選択することはなく，Gfarm ライブラリが暗黙的に行う．ファイル配置先ノードの選択にあたっては原則的にローカルストレージが選出されるが，ノードの負荷状況やレイテンシなどを考慮して別のノードとなる場合もある．

```
void ADIOI_GFARM_Open(ADIO_File fd, ...)
{
    // ...
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    // ...
    sprintf(fn, "%s.%d", fd->filename, myrank);
    // ...
    if (fd->access_mode & ADIO_CREATE){
        // fd->filename indicates O_CREAT is set
        gfs_pio_create(fd->filename);
        // create file for rank = £myrank
        gfs_pio_create(fn, ...);
    } else {
        if (isExist(fd->filename)){
            gfs_pio_create(fn, ...)
        } else {
            gfs_pio_open(fn, ...)
        }
    }
    // initialize ADIO file object
    // ...
}
```

図 7 Open 関数内でのファイル分割.

```
int amode = MPI_MODE_RDWR;
if (myrank == 0){
    int m = amode | MPI_MODE_CREATE;
    MPI_File_open(comm, "result.dat", m, ...);
    MPI_Barrier(comm);
} else {
    MPI_Barrier(comm);
    MPI_File_open(comm, "result.dat", amode, ...);
}
```

図 8 代表プロセスがファイルを作成する MPI プログラムの例.

5. 性能評価

評価は TSUBAME 3.0 で行い，NVMe SSD で構成される計算ノードローカルのスクラッチ領域を使用する．実験では，IOR ベンチマークを MPI-IO インタフェースに対して実行し，Read/Write におけるバンド幅の測定を行う．このときの資源タイプは f_node を使用し，他の計算ジョブが同じノードに割り当てられないようにする．IOR は各ノードあたり 1 プロセスで実行する．IOR では single-shared-file 方式でアクセスし，各プロセスが単一ファイルに対して 20 GiB 読み書きする．この操作を 10 回行い，バンド幅の平均を求める．

5.1 予備実験: ブロックサイズのチューニング

最初に予備実験として，ローカル SSD と，並列ファイルシステムである Lustre，共有スクラッチ領域の BeeOND の性能を調査する．BeeOND は TSUBAME 3.0 で標準提供されるファイルシステムで，割り当てられた計算ノードのローカル SSD を束ねてひとつの共有ファイルシステムを構成する．実験はシングルノードで，ブロックサイズを 4 KiB, 8 KiB, ..., 32 MiB と 4 KiB の倍数で変化させながら，SSD は POSIX API，Lustre と BeeOND は MPI-IO を用いて，最も性能が出る値について調査を行う．Lustre については全ての OST でストライピングを行う設定で，ストライプサイズは 1 MiB に設定する．MPI-IO の実装は MPICH v3.3 のソースコードをビルドして使用する．MPICH の実装に含まれる最適化手法である Lock-Ahead [6] は今回は利用していない．

まず，SSD における IO バンド幅を**図 9** に示す．図中の点は 10 回実行したときの平均値，エラーバーはバンド幅の 90% 信頼区間を表している．SSD の場合は，ブロックサイズが 8 MiB のときが最も良い結果であった．

次に，Lustre の結果を**図 10** に示す．Lustre は 4 MiB の場合が最も良い性能を示し，ストライピングを行う分ばらつきの大きい結果となった．また 4 KiB の場合は指定した時間内に結果が出なかったため，途中でジョブを終了した．

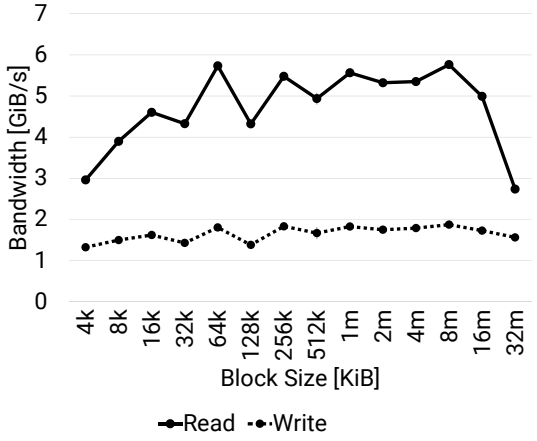

図 9　シングルプロセスに対するローカル SSD のバンド幅

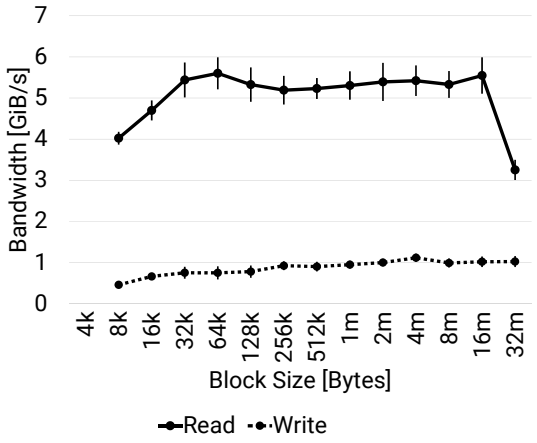

図 10　シングルプロセスにおける Lustre のバンド幅

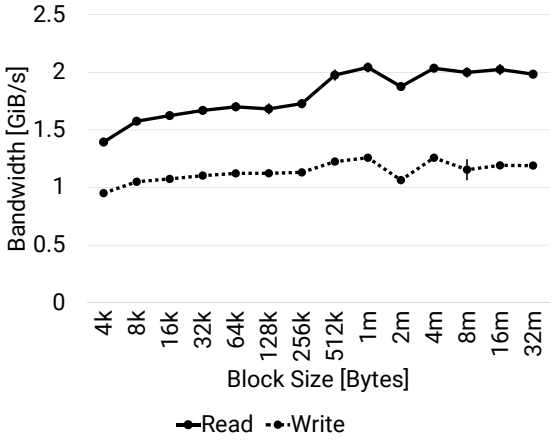

図 11　シングルプロセスにおける BeeOND のバンド幅

最後に，BeeOND の結果を図 **11** に示す．BeeOND の場合は 16 MiB の場合に読み込み性能が最も良い結果になったが，書き込み性能が低下した．書き込み性能は 8 MiB が最も性能が良い結果であった．

5.2　提案手法の N-1 アクセスパターンに対する性能評価

提案手法の評価として，N-1 アクセスパターンにおける Read/Write のバンド幅について評価を行う．ブロックサイズはローカル SSD の性能測定結果をもとに 8 MiB に設定し，インタフェースは MPI-IO を利用して行う．ノードあたりのプロセス数は 1 とし，ノードを 1, 2, 4, ... と 16 ノードまで増やしながら調査する．MPI-IO の実装は MPICH v3.3 のソースコードをビルドして使用した．Gfarm はメタデータの永続化なしのバーストバッファモードで計算ノード上に起動する．使用する計算ノードのうちの一台で，メタデータノードとストレージノードが共存する構成になっている．また，性能比較として Lustre と BeeOND における N-1 アクセスパターンの評価も行う．評価にあたっては，先ほど調査したブロックサイズをそのシステムで最も性能が出ると考えられる値として，Lustre は 4 MiB，BeeOND は 8 MiB に設定する．それ以外の条件は先ほどのパラメータチューニングで行った実験と同じものを用いる．

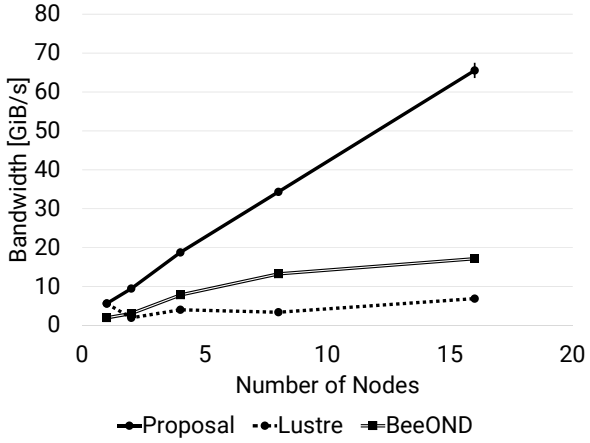

図 12　N-1 アクセスにおける Read バンド幅

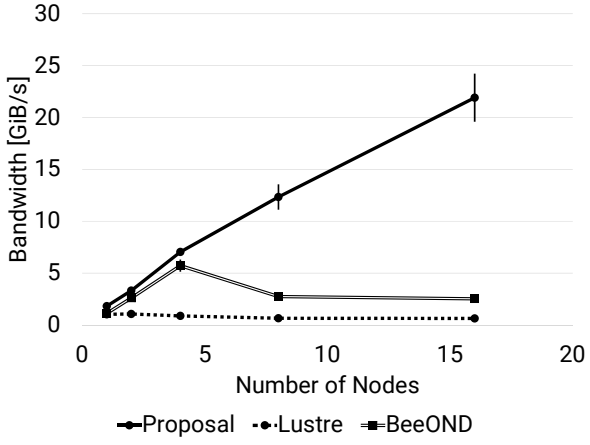

図 13　N-1 アクセスにおける Write バンド幅

実験結果を図 **12** と図 **13** に示す．提案手法はノード数が増えた場合にも Read/Write において線形にスケールしたが，Write においてはばらつきが大きくなった．一方で Lustre は，Read/Write におけるバンド幅は不安定のままで，性能はスケールしなかった．BeeOND は，Read は

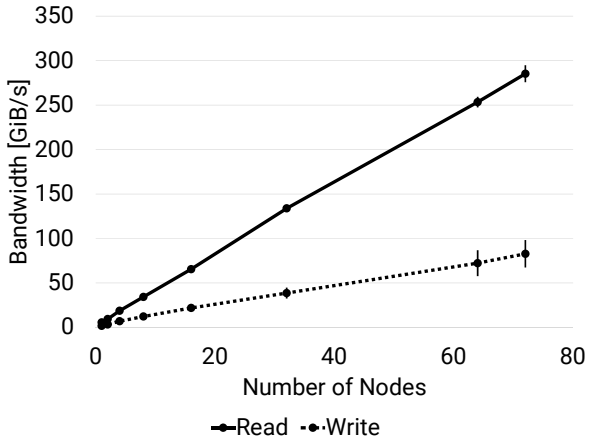

図 **14** N-1 アクセスにおける提案手法のバンド幅

ノード数に比例してバンド幅が高くなった一方で，Write は 4 ノードを境に性能が低下した．

5.3 提案手法におけるスケーラビリティの評価

また追加実験として，提案手法のスケーラビリティの調査を行った．ノード数を更に増やし，72 ノードを上限として測定を行う．72 ノードは TSUBAME 3.0 における通常の計算ジョブで指定できるノード数の上限である．結果を図 **14** に示す．ノード数が 72 の場合の Read バンド幅は 285.3 GiB/s となった．グラフの形状から，性能は 72 ノードまで線形にスケールしていることが分かる．Write の場合も性能は線形にスケールしており，72 ノードのときのバンド幅は 82.8 GiB/s となった．

6. まとめ

本研究では，ノードローカルストレージを活用したバーストバッファについて議論を行った．提案手法においてはノードローカルストレージを利用して IO 効率を高めるため，計算ノードとプロセスのローカリティを考慮したファイル分割を導入した上で，プロトタイプを実装して性能評価を行った．性能評価においてはストレージ性能を評価する IOR ベンチマークを用いて評価を行い，評価の結果，ノード数が増えた場合でも線形にスケールすることを示した．

謝辞 本研究の一部は，JST CREST JPMJCR1303，JST CREST JPMJCR1414，JSPS 科研費 17H01748，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) および富士通研究所との共同研究の助成を受けたものです．

参考文献

[1] Bent, J., Gibson, G., Grider, G., McClelland, B., Nowoczynski, P., Nunez, J., Polte, M. and Wingate, M.: PLFS: a checkpoint filesystem for parallel applications, *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pp. 1–12 (online), DOI: 10.1145/1654059.1654081 (2009).

[2] Cray, inc.: Cray DataWarp™ Applications I/O Accelerator, Cray, inc. (online), available from ⟨https://www.cray.com/products/storage/datawarp⟩ (accessed 2019-01-12).

[3] Data Direct Networks, inc.: Infinite Memory Engine®, Data Direct Networks, inc. (online), available from ⟨https://www.ddn.com/products/ime-flash-native-data-cache/⟩ (accessed 2018-10-24).

[4] del Rosario, J. M., Bordawekar, R. and Choudhary, A.: Improved Parallel I/O via a Two-phase Run-time Access Strategy, *SIGARCH Comput. Archit. News*, Vol. 21, No. 5, pp. 31–38 (online), DOI: 10.1145/165660.165667 (1993).

[5] Moody, A., Bronevetsky, G., Mohror, K. and Supinski, B. R. d.: Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System, *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, Washington, DC, USA, IEEE Computer Society, pp. 1–11 (online), DOI: 10.1109/SC.2010.18 (2010).

[6] Moore, M., Farrell, P. and Cernohous, B.: Lustre Lockahead: Early experience and performance using optimized locking, *Concurrency and Computation: Practice and Experience*, Vol. 30, No. 1, p. e4332 (2018).

[7] Rajachandrasekar, R., Moody, A., Mohror, K. and Panda, D. K. D.: A 1 PB/s File System to Checkpoint Three Million MPI Tasks, *Proceedings of the 22nd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '13, New York, NY, USA, ACM, pp. 143–154 (online), DOI: 10.1145/2493123.2462908 (2013).

[8] Tatebe, O., Hiraga, K. and Soda, N.: Gfarm Grid File System, *New Generation Computing*, Vol. 28, No. 3, pp. 257–275 (online), DOI: 10.1007/s00354-009-0089-5 (2010).

[9] Thakur, R., Gropp, W. and Lusk, E.: An abstract-device interface for implementing portable parallel-I/O interfaces, *Proceedings of 6th Symposium on the Frontiers of Massively Parallel Computation (Frontiers '96)*, pp. 180–187 (online), DOI: 10.1109/FMPC.1996.558080 (1996).

[10] Thakur, R., Gropp, W. and Lusk, E.: Data sieving and collective I/O in ROMIO, *Proceedings. Frontiers '99. Seventh Symposium on the Frontiers of Massively Parallel Computation*, pp. 182–189 (online), DOI: 10.1109/FMPC.1999.750599 (1999).

[11] Wang, T., Mohror, K., Moody, A., Sato, K. and Yu, W.: An Ephemeral Burst-buffer File System for Scientific Applications, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, Piscataway, NJ, USA, IEEE Press, pp. 69:1–69:12 (online), available from ⟨http://dl.acm.org/citation.cfm?id=3014904.3014997⟩ (2016).

[12] 木村浩希，建部修見：MPI-IO/Gfarm：分散ファイルシステム Gfarm のための MPI-IO の実装と評価，情報処理学会論文誌，Vol. 52, No. 12, pp. 3239–3250 (2011).

正誤表

下記の箇所に誤りがございました．お詫びして訂正いたします．

訂正箇所	誤	正
6 ページ 図 11