社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2008－HPC－116 (37)
2008／8／7

マルチコアクラスタ向け並列ファイルシステムアーキテクチャ

太田一樹[†] 石川裕[†,††]

マルチCPU・マルチコアCPUが一般的になり、クラスタ内で実行される計算プロセスの数が増大している。各プロセスは連続したI/O要求を発行する傾向に有るが、全てのプロセスが同時にI/Oを発行すると、他のノードの要求に割り込まれ、非連続的なディスク操作が行われる。これにより並列ファイルシステムのI/Oノードでのディスクシーク回数が増加しパフォーマンスの低下に繋がる。

この問題を解決するため、Gather-Arrange-Scatter (GAS) と呼ばれるI/Oアーキテクチャを提案する。GASでは計算ノードで一旦要求が整列化されてからI/Oノードに並列で要求が送信される。これにより整列化しないケースに比べてBTIOベンチマークで12.7%の性能改善を達成することが示された。

Parallel File System Architecture for Multi-Core Clusters

KAZUKI OHTA[†] and YUTAKA ISHIKAWA[†,††]

In a cluster of multiple processors or cpu-cores, multiple processes may run on each compute node. Each process issues contiguous I/O requests, but they are interrupted by the requests of other processes in the same node. The I/O nodes in parallel file systems receive these requests as non-contiguous fashion. This increases the disk seek time and causes performance degradation.

In order to overcome the problem, a new I/O architecture for parallel file systems, called the Gather-Arrange-Scatter(GAS) architecture, is proposed. In GAS, the I/O requests in the same node are gathered locally. Those are arranged in a better order, and scattered to the remote disks in parallel. A prototype is implemented and evaluated using the BTIO benchmark. Arranged-case results in up to 12.7% performance improvement compared to the non-arranged case.

1. はじめに

CPUの演算能力の増大に伴って、科学計算アプリケーションが扱うデータ量が年々増加しており、テラバイト級のデータを扱うこともある。しかし、そのようなサイズのデータを扱うには単一ディスクのI/Oバンド幅は狭すぎるため、複数のディスクを単一のファイルシステムとして扱うための研究が数多くなされてきた[1),3),7),9)]。

科学計算アプリケーションは高性能なCPU・ネットワークを持つクラスタ上で実行されることが多い。クラスタは通常、計算ノードとI/Oノードで構成され、計算ノードには高い計算能力を持ったCPU、I/Oノードには大容量・高バンド幅なディスクが配置される。最近ではマルチコアCPUが一般的になり、計算ノード上で実行される計算プロセスの数が増大している傾向にある。

各計算プロセスは連続したI/O要求を出しやすいという傾向があるが、もし全てのプロセスが同時にI/Oを行った場合、他のプロセスの要求に割り込まれてしまう。その結果、並列ファイルシステムのI/Oノードでのディスクシーク回数が増加し、深刻なパフォーマンス低下に繋がる。

そこで本研究ではマルチコアクラスタ上の並列ファイルシステムにおいて、I/Oノードでのパフォーマンス低下を低減する手法Gather-Arrange-Scatter (GAS)を提案し実装する。このシステムでは、アプリケーションが発行した要求は一旦、計算ノードに集められる。そして計算ノード上で並び替え・併合を行ってから、I/Oノードに要求が送信される。これにより、整列化された要求をI/Oノードが処理することが出来る。

本論文ではまずGather-Arrange-Scatterの設計について述べ、次に実装について述べる。そしてNas Parallel Benchmark BTIO[12)]を用いて実装を評価する。

† 東京大学大学院情報理工学系研究科
Graduate School of Information Science and Technology, The University of Tokyo
†† 東京大学情報基盤センター
Information Technology Center, The University of Tokyo

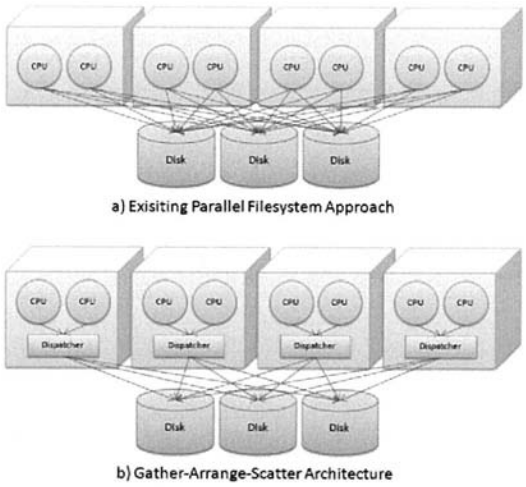

図 1 The difference between a) Existing Parallel File System and b) Gather-Arrange-Scatter I/O Architecture

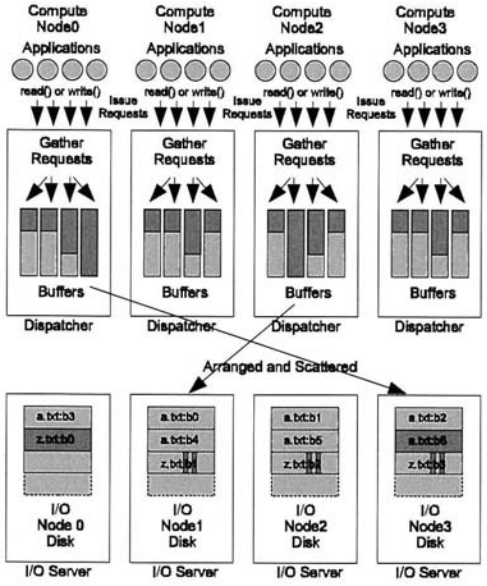

図 2 Detailed View of the Gather-Arrange-Scatter (GAS) Architecture

2. 設　　計

前述した問題を克服するため、本研究では Gather-Arrange-Scatter (GAS) と呼ばれる並列ファイルシステムのアーキテクチャを提案する。既存の並列ファイルシステムと GAS の違いを図 1 に示す。従来、各計算のノード上の複数プロセスから発行された要求は個別に並列 I/O ノードに送られていた。GAS では、計算ノードから発行された要求を一時的にバッファリングし、並び替え・併合を行ってから、並列に I/O ノードに送信する。これにより、あるプロセスが発行した連続的な要求は、その順序が保たれたまま I/O ノードに送信される。以降、計算プロセスからの要求を集めるフェーズを Gather フェーズ、並び替え・併合を行うフェーズを Arrange フェーズ、整列化された要求を I/O サーバーに送信するフェーズを Scatter フェーズと名づけることにする。

より詳細な GAS の仕組みを図 2 に示す。GAS では、**ディスパッチャー**と **I/O サーバー**という 2 つのサーバーが重要な役割を果たす。ディスパッチャーは全ての計算ノード上に常駐し、Gather-Arrange-Scatter の 3 つのフェーズの処理を行う。I/O サーバーはディスパッチャーで整列化された要求を受け取り、ディスクに対する操作を行う。以下では Gather フェーズ、Arrange フェーズ、Scatter フェーズの 3 つのフェーズで行われる具体的な処理を述べる。

2.1 Gather フェーズ

アプリケーションが `write()` システムコールか `read()` システムコールを呼び出すと、その要求がディスパッチャーに送信される。ディスパッチャーでは次の Arrange フェーズで並び替えを実現するために、発行された要求を一時的にバッファリングする。発行した I/O の完了をアプリケーションが逐一待っていると、連続した要求をバッファリングできないため、非同期 I/O を用いる。

非同期 I/O を用いると、`write()` や `read()` システムコールが呼ばれた際、要求が発行された直後にアプリケーション側に制御が戻る。これによりディスパッチャーは要求のバッファリングを行えると共に、アプリケーションは I/O の完了を待たずに計算処理に戻ることが出来る。

非同期 I/O は科学計算アプリケーションと相性が良い。なぜなら多くのアプリケーションでは計算フェーズと I/O フェーズが交互にやってくるため[8]、非同期で I/O を行うと I/O と計算を同時に行うことができ、結果的に実行時間を縮めることが出来る。

これは大規模なアプリケーションにおいて、現在の計算結果を出力するスナップショットや、故障時に備えて現在のステートを出力するチェックポインティングがよく用いられることからも言える。

2.2 Arrange フェーズ

収集された要求はメモリにバッファリングされる。そのためのバッファ領域は、I/O ノードと同じ数だけのサブバッファーに分かれており、各々は要求が向かう I/O ノードに対応している。通常、並列ファイルシステムではファイルは同じサイズのブロックにストライピングされる。要求がどのサブバッファーに向かうかはファイル名やオフセット等の情報を元に決定され

るが、詳細は実装による。

サブバッファーが溢れたり一定時間が経過すると、バッファリングされた要求はシーク回数が少なくなるように並び替えられる。また連続した要求はマージされる。性能を出すためには要求数を減らすことも重要だからである[11]。

図2は上記の一連の操作を示している。この図では4つの計算ノードと4つのI/Oノードがある場合が示されており、ディスパッチャーの持つ4つのサブバッファーはそれぞれのI/Oノードに対応している。この図では計算ノード2の持つサブバッファーが溢れているので、並び替えられてI/Oノード1に要求が送信されている。I/Oノード1では送信されてきた要求を受け取り、実際のディスク操作を行う。

2.3 Scatter フェーズ

並び替えられた要求は対応するI/Oサーバーに並列に送信される。この要求はI/Oサーバーによって1つ1つ同期的に実行される。

3. 実　　装

提案手法の有効性を評価するため、Parallel Gather-Arrange-Scatter ファイルシステム (PGASFS) をLinux上に実装した。現在の実装では、読み込みは同期的に行われる。またメタデータの扱いは外部のNFSサーバーに任せる形になっている。

PGASFSは**ディスパッチャー、I/Oサーバー、システムコールフックライブラリ**の3つのコンポーネントから成っている。以下ではそれぞれの実装について詳しく見ていく。

3.1 ディスパッチャー

ディスパッチャーはマルチスレッドサーバーで、ユーザー空間で動作する。アプリケーションからの要求は`UNIX`ドメインソケットで受け取る。UNIXドメインソケットを使用したのは、コネクション指向である点・ローカルでの転送速度が速い点が理由である。アプリケーションが`open()`システムコールを呼ぶと、ローカルのディスパッチャーへの接続が開始される。逆に`close()`システムコールを呼ぶと接続は終了される。ディスパッチャーはアプリケーションプロセスをProcessIDで識別する。

PGASFSでは他の並列ファイルシステムと同様、ファイルは同じサイズのブロックにストライピングされる。各ブロックはhash(FilePath) + ($\frac{\text{Offset}}{\text{BlockSize}}$%I/Oノードの数) で算出されるランクを持つI/Oサーバーに保持される。デフォルトブロックサイズは4096バイトだが、変更可能である。

アプリケーションから送信された要求は上記の数式で決定されるサブバッファーにバッファリングされる。もし要求が複数のブロックに渡る場合は分割される。サブバッファーが溢れるか一定時間が経過すると、

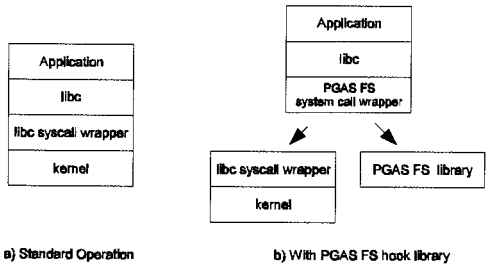

図3 Hooking System Calls

バッファリングされた要求が並び替えられる。並び替える際には、まずファイル名についてソートし次にオフセットについてソートする。この操作により連続する要求は隣り合いやすくなる。次に連続する要求については1つの要求に出来る限りマージする。これによりI/Oサーバーがファイルシステムに発行するI/O要求数が削減され、総実行時間の短縮につながる。

3.2 I/Oサーバー

I/Oサーバーもマルチスレッドサーバーで、ユーザー空間で動作する。ディスパッチャーから送られてきた要求を受け取り、1つ1つそれを実行する。I/Oサーバーはディスクをブロック単位で管理する。新しい要求がきたら、まずその要求の操作が行われる領域に対してブロックが割り当てられているかどうかをチェックする。割り当てられていたらそこに対してディスク操作を行い、そうでなければ空きブロックリストから空の領域を取得し、そこに対してディスク操作を行う。

3.3 システムコールフックライブラリ

PGASFSでは、ファイルシステムにアクセスするための独自APIを提供する代わりにシステムコールをフックするという方法を取っている。これにより既存のアプリケーションについてはソースコードを変更することなくPGASFSを利用することが可能になる。ただし、実行する際に特別なライブラリをロードする必要がある。図3aはPGASFSを利用しない場合のシステムコールの実行フロー、図3bは利用する場合の実行フローが示されている。ライブラリをロードする際には`LD_LIBRARY_PATH`という環境変数に、フック用のライブラリのパスを設定する。またファイルパスの先頭に"pgas:"という接頭辞を付ける必要もある。

PGASFSを利用するためのコマンドラインを図4に示す。最初のコマンドはfoo.txtをローカルディスクからPGASFS上にコピーするコマンドである。次のコマンドはその内容を読み出すコマンドである。

システムコールフックライブラリでは、ファイルシステム関連のシステムコール (`open()`, `creat()`, `lseek()`, `read()`, `write()`, `close()`) をフックする。

`open()` と `creat()` ではまず指定されたパスが"pgas:"という接頭辞を持っているかどうかをチェッ

```
$ LD_PRELOAD=/path/to/libsyshook.so \
  cp foo.txt pgas:/path/to/bar.txt
$ LD_PRELOAD=/path/to/libsyshook.so \
  cat pgas:/path/to/bar.txt
```

図 4 Using the PGAS file system

クする。もし持っているなら、ディスパッチャーに対する接続を張りファイルディスクリプタを返す。このディスクリプタに対する `read()` や `write()` についてはこのコネクションを通じてディスパッチャーに送信される。

それ以外のシステムコールでは、渡されたディスクリプタが PGASFS 用のものかどうかをチェックし、それによって処理を分岐する。ディスクリプタのオフセットについてはフックライブラリの中で管理されており、`lseek()` が呼ばれてもカーネルモードにスイッチしないようになっている。

またフックライブラリがロードされた際、I/O 用のスレッドが 1 つ用意される。`write()` システムコールが呼ばれるとその内容がキューに詰まれる。この時点でシステムコール自体は終了し、アプリケーションは計算に戻ることができる。キューに詰まれた要求は I/O スレッドによって 1 つ 1 つディスパッチャーに転送されていく。以上の仕組みにより非同期書き込み機能が実現されている。

3.4 一貫性問題

ここでは PGASFS の一貫性について述べる。ほとんどの科学計算アプリケーションでは、書き込み時には書き込み専用モードで `open()` システムコールが呼ばれる。したがって、POSIX I/O のセマンティクスに完全準拠せず、ある程度緩めた一貫性をサポートすることで高性能な I/O を行う事が出来るようになる。PGASFS では書き込み専用モードで `open()` された際は上記の非同期書き込みを使用して I/O が行われる。それ以外の場合に関しては同期的に操作が行われる。

また `close()` を呼んだ際にアプリケーションが行った I/O 操作がディスクに反映されている事を保障するために、`close()` システムコールが呼ばれるとそのプロセスが発行した要求がすべて I/O サーバーによって処理されるのを待つという処理を行う。

4. 評　　価

4.1 実験環境

実験に用いた環境を図 5 及び表 1 に示す。全ての計算ノードは 10Gbps の Myrinet[2] と 1Gbps のイーサネットで接続されている。ディスクのバンド幅は hdparm を用いて計測した。

4.2 ソフトウェア環境

MPI の実装は MPICH2[6] バージョン 1.0.5 を用いた。MPICH とベンチマークプログラムに関しては Intel C Compiler バージョン 10.0.0 を用いてコンパイルした。他のプログラムに関しては GCC バージョン 4.1.1 を用いてコンパイルした。

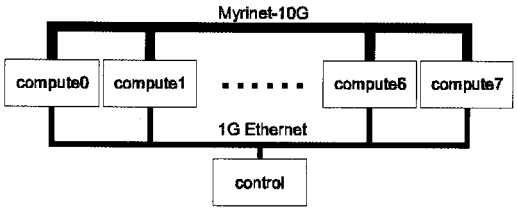

図 5 ネットワーク構成

表 1 ハードウェアスペック

8 Compute Nodes	
CPU	Opteron 2212HE(dual-core, 2.0GHz)*2
Memory	6 GB
Chipset	nVidia nForce Professional 3600+3050
HDD	Serial ATA Disk (50.28 to 53.21 MB/sec)
OS	Linux 2.6.18-8.1.8.el5 SMP
I/O Scheduler	CFQ I/O Scheduler
1 Control Node	
CPU	Opteron 2218HE(dual-core, 2.6GHz)*2
Memory	16 GB
Chipset	nVidia nForce Professional 3600+3050
HDD	Serial ATA Disk (50.61 MB/sec)
OS	Linux 2.6.18-8.1.8.el5 SMP
I/O Scheduler	CFQ I/O Scheduler

比較のために PGASFS 以外にも、ローカルファイルシステム・NFS(非同期モード)・PVFS2[4] を用いた。ローカルディスクは単純に計算ノードのローカルディスクに書き出しているのみである。

NFS は複数ディスクを単一のファイルシステムとして見せかけることが出来ないため、コントロールノードのディスクのみを使用した。他のファイルシステムに関しては 8 つのディスクを使用しているが、NFS のときは 1 つしか使用していない。

PVFS2 に関してはバージョン 2.7.0 を用いた。`pvfs2-genconfig` が生成したデフォルトの設定を使用した。ベンチマークは PVFS2 の VFS モジュールを使用して行った。I/O サーバーを計算ノードに走らせ、メタデータサーバーはコントロールノードに置いた。

MPICH2 は Myrinet-10G を使用するように設定した。PVFS・NFS・PGASFS 内での通信については 1Gbps イーサネットを用いている。

4.3 ベンチマーク

並列 I/O 性能を測定するために BTIO ベンチマーク[12] としてとして知られる、Block-Tridiagonal (BT) Nas Parallel Benchmark バージョン 3.3 を用いた。BTIO は NASA Advanced Supercomputing Division によって開発され、Fortran で記述されている。BTIO では MPI I/O Collective Buffering, MPI I/O,

Fortran I/O の3つのオプションが用意されているが、今回はPOSIXインターフェースを用いているFortran I/Oを選択した。Fortran I/Oでは40byteの書き込みを大量に発行するようなパターンになっている。

4.3.1 他のファイルシステムとの比較

表2 BTIO Benchmark Result (8 nodes / 16 processes)

Total Execution Time (sec)				
Class	Local	NFS	PVFS2	PGASFS
A (0.4GB)	57.82	686.41	385.20	65.22
B (1.6GB)	114.48	1858.21	1404.58	241.74
C (6.8GB)	422.16	N/A	5358.19	1123.54
Write Bandwidth (MB/sec)				
Class	Local	NFS	PVFS2	PGASFS
A (0.4GB)	25.46	0.65	1.18	341.31
B (1.6GB)	38.57	1.22	1.30	321.24
C (6.8GB)	39.39	N/A	1.36	338.19

BTIOベンチマークを用いて、PGASFSと他のファイルシステムの実行時間を比較した。表2はその結果である。PGASFSのWrite Bandwidthは、非同期I/Oを行っているためにメモリコピーの速度となっており、実際のI/O速度には対応していない。また非同期I/Oによって計算とI/Oをオーバーラップしているにも関わらずローカルディスクよりも遅いのは、ディスパッチャーのいずれかのフェーズにボトルネックがあるからと考えられる。これには更なる調査が必要である。

4.3.2 GASフェーズの影響

次に要求の並び替えをする場合としない場合において、I/Oサーバーでの`lseek()`の回数・I/O要求数・総実行時間を比較した。ベンチマークの際にはBTIOを8ノード・16プロセスで走らせた。ブロックサイズとサブバッファーのサイズはそれぞれ64Kバイトに設定した。

表3 BTIO (8n/16p) The Impact of Request Arrangement

Number of lseek() at I/O servers (times)			
Class	NotArranged	Arranged	Reduced
A (0.4GB)	2287540	666259	70.9%
B (1.6GB)	8512074	1713626	79.9%
C (6.8GB)	27801033	4374196	84.3%
Number of I/O Requests (times)			
Class	NotArranged	Arranged	Reduced
A (0.4GB)	10490880	666575	93.6%
B (1.6GB)	42468984	3057766	92.8%
C (6.8GB)	170142519	11229406	93.4%
Total Execution Time (sec)			
Class	NotArranged	Arranged	Reduced
A (0.4GB)	74.71	65.22	12.7%
B (1.6GB)	260.47	241.74	7.2%
C (6.8GB)	1281.92	1123.54	12.4%

表3はその結果を示している。約80%のlseek()回数が減っていることが分かる。また約90%の要求が上手くマージされている。その結果、約7.2%〜12.7%の実行時間改善が行われた。

4.3.3 サブバッファーサイズの影響

最後にサブバッファーのサイズが及ぼす影響を調査した。図6は`lseek()`の回数とサブバッファーサイズ、図7はI/O要求数とサブバッファーサイズの関係を示している。両方のケースにおいてサブバッファーのサイズを大きくすると減少しているが、大きくするに従って減少の度合いが下がっていく。

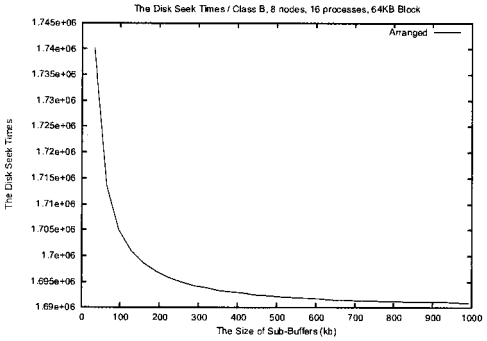

図6 `lseek()` Times vs. Sub-Buffer Size

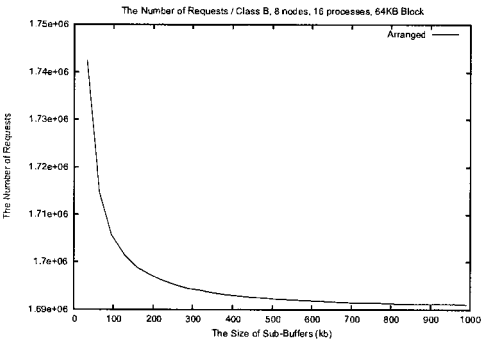

図7 Number of Requests vs. Sub-Buffer Size

図8は総実行時間との関係を示している。サブバッファーサイズを大きくすると総実行時間は徐々には改善されることが分かる。

5. 関連研究

PVFS[3),4)]、GPFS[7)]、Lustre[1)]は本研究と同様に複数のディスクを1つのファイルシステムと見せ、高性能な並列ファイルシステムを構成しようとする試みである。図1の上側のアーキテクチャを採用しており、1章で指摘したような問題が発生すると考えられる。こ

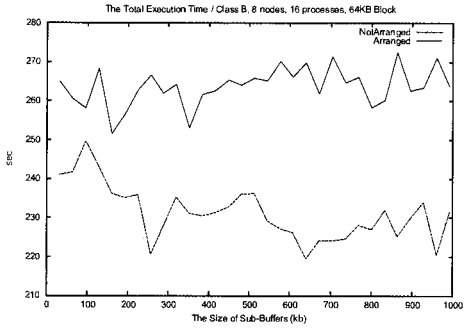

図 8 Total Time vs. Sub-Buffer Size

れらは計算ノードとデータノードので構成されるクラスタで動作させられることを意図して作られており、GAS の仕組みを取り込むことが可能であると考える。

Message Passing Interface(MPI) バージョン 2 では、MPI-IO[10] と呼ばれる並列 I/O のためのインターフェースが規定された。MPI-IO では集団で I/O を行うためのプリミティブが提供されている。MPI-IO では I/O を行う前にプロセス間で明示的に同期し情報を交換することでより効率的な I/O を行える。TwoPhase I/O[5] では全てのプロセスが発行しようとしている I/O 要求の情報をアグリゲーターと呼ばれるプロセスに集め、連続した要求を併合する。さらにそれをプロセスに再分配することによって、効率的な I/O が行えるようになる。離れたプロセスが連続した要求を出している場合に最も効率的に働く。MPI-IO を使用するためには MPI-IO の複雑なインターフェースを使用してプログラムを記述する必要があるため、既存のアプリケーションについては改変が要求される。我々のファイルシステムのアプローチでは、そうではない。

6. おわりに

本研究では、並列ファイルシステムにおいてディスクシーク回数とリクエスト数を削減する Gather-Arrange-Scatter (GAS) と呼ばれるアーキテクチャを提案し、Linux 上に実装した。GAS では、マルチコアクラスタ上で多数のプロセスが一度に連続した I/O リクエストを発行した際にも、I/O ノードが連続した形でディスク操作を行えるようになる。整列化した場合、BTIO ベンチマークで約 12.8%の実行時間改善を達成出来ることが示された。

今後の課題としては、クライアントサイドの整列化だけではなくサーバーサイドの整列化も行い並列ファイルシステム用のリクエストスケジューリング機構を構築すること、そしてより多くの I/O ワークロードに対して性能評価を行うこと、が挙げられる。

参考文献

1) Lustre file system. http://lustre.org/.
2) Myrinet. http://www.myri.com/.
3) Philip H. Carns, Walter B. Ligon III, Robert B. Ross, and Rajeev Thakur. Pvfs: a parallel file system for linux clusters. In *ALS'00: Proceedings of the 4th conference on 4th Annual Linux Showcase & Conference, Atlanta*, pages 28–28, Berkeley, CA, USA, 2000. USENIX Association.
4) The PVFS Community. Parallel virtual file system, version 2. http://www.pvfs.org/.
5) Juan Miguel del Rosario, Rajesh Bordawekar, and Alok Choudhary. Improved parallel i/o via a two-phase run-time access strategy. *SIGARCH Comput. Archit. News*, 21(5):31–38, 1993.
6) Argonne National Laboratory. Mpich2 : High-performance and widely portable mpi. http://www.mcs.anl.gov/research/projects/mpich2/
7) Frank Schmuck and Roger Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proc. of the First Conference on File and Storage Technologies (FAST)*, pages 231–244, January 2002.
8) Evgenia Smirni and Daniel A. Reed. Lessons from characterizing the input/output behavior of parallel scientific applications. *Perform. Eval.*, 33(1):27–44, 1998.
9) Osamu Tatebe, Youhei Morita, Satoshi Matsuoka, Noriyuki Soda, and Satoshi Sekiguchi. Grid datafarm architecture for petascale data intensive computing. In *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2002)*, 2002.
10) Rajeev Thakur, William Gropp, and Ewing Lusk. On implementing MPI-IO portably and with high performance. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, pages 23–32, 1999.
11) Rajeev Thakur, William Gropp, and Ewing Lusk. Optimizing noncontiguous accesses in MPI-IO. *Parallel Computing*, 28(1):83–105, 2002.
12) Parkson Wong and Rob F. Van der Wijngaart. Nas parallel benchmark i/o version 2.4, nas technical report nas-03-002, nasa ames research center, moffett field, ca 94035-1000.