Vol.2014-HPC-144 No.13
2014/5/27

並列ファイルシステムに対する I/O リクエスト調停機構の提案

大野 善之[1]　堀 敦史[1]　石川 裕[2,1]

概要：HPC システムが大規模化することにより，ストレージサーバあたりの負荷が増えている．ストレージサーバの負荷が増えることで I/O 性能が低下するため，I/O 性能がボトルネックとなりスケーラビリティを阻害することが問題とされている．そこで我々は，既存の並列ファイルシステムについて，ストレージサーバの負荷に応じて，I/O 性能がどのように変化するについて評価を行った．評価により，ストレージサーバに対して同時にアクセスするクライアント数が多い場合に，I/O 性能が低下するという問題があることがわかった．そこで，本問題の解決策として，複数のクライアントノードからストレージサーバに対する I/O 要求を一定値以上に増やさないように調停することで，ストレージサーバの負荷の低減する I/O 調停機構を提案する．「京」でベンチマークプログラムを用いてスループットを測定したところ，通常の I/O では，16 クライアントプロセスが同時に 1GB のデータを write したときに，同時にアクセスするクライアントプロセス数を 2 に制限すると，スループットが 25 %向上することを確認した．

キーワード：I/O 調停機構，並列 I/O 最適化，並列ファイルシステム

1. はじめに

現在のスーパーコンピュータの多くは，Lustre [1]，PVFS [2]，GPFS [3] といった並列ファイルシステムを採用している．並列ファイルシステムは，ファイルシステム全体の名前空間や各ファイルごとのメタデータを管理するメタデータサーバと，各ファイルの実体を管理するストレージサーバとで構成される．これら，並列ファイルシステムを構成するサーバ群は，スーパーコンピュータの各計算ノードから並列にアクセスされる．スーパーコンピュータが大規模化するにつれて，並列ファイルシステムのサーバあたりの計算プロセス数が増加し，並列ファイルシステムの各サーバに対する I/O 要求も増大する．そのため，ファイルシステムの負荷が大きくなり I/O 性能が低下する．I/O 性能の低下がボトルネックとなることが，スーパーコンピュータのスケーラビリティを阻害する問題の１つとなっている．

我々は上記の並列ファイルシステムに対する アプリケーションの I/O のスケーラビリティに関して研究を行っている．メタデータサーバへの負荷集中に関するは，File Composition Library [4] を提案している．本稿では，ストレージサーバに対する負荷の集中をターゲットとする．

まず我々は，既存の並列ファイルシステムについて，ストレージサーバの負荷に応じて，I/O 性能がどのように変化するについて評価を行った．評価には，多くのスーパコンピュータシステムで採用されている Lustre ファイルシステムをベースとする FEFS (Fujitsu Exabyte File System) [5] を用いた．評価では，並列アプリケーションの全プロセスが同時にそれぞれプロセスごとに異なるファイルにアクセスするような I/O を対象とした．この I/O パターンは，多くの並列アプリケーションで採用されている I/O パターンであるが，同時に全てのクライアントプロセスがストレージサーバにアクセスするため，ストレージサーバの負荷が増えるため性能が低下する．

Lustre では，クライアントとストレージサーバ間の通信は，一定のデータサイズごとのブロックに分割されて，そのブロックが 1 つの RPC によって転送される．上記のようなアクセスパターンでは，並列に動く RPC の数が多いほど性能が低下するだけでなく，ストレージサーバに対するクライアント数が多いときほど性能が低下するという結果になった．

そこで本論文では，ストレージサーバに並列に I/O を発行するクライアント数，および，並列に動作する RPC 数

[1] 理化学研究所 計算科学研究機構
RIKEN AICS
[2] 東京大学
The University of Tokyo

を制限するような調停を行うことでストレージサーバの性能低下を防ぐ I/O 調停機構を提案する．

以下，第 2 章では Lustre ファイルシステムについて，ストレージサーバの負荷の違いによる性能差についての測定結果を報告する．第 3 章では我々の提案する I/O 調停機構について述べ，第 4 章では I/O 調停機構を用いた場合の I/O 性能の評価結果を述べる．そして第 5 章において関連研究を紹介し，第 6 章でまとめと今後の課題について述べる．

2. 予備評価

既存の並列ファイルシステムについて，ストレージサーバの負荷に応じて性能がどのように変化するについて評価を行った．評価には，我々が保有する FX10 クラスタ と並列ファイルシステム FEFS (Fujitsu Exabyte File System) [5] を用いた．

2.1 Lustre File System

Lustre は，オープンソースであり，性能も高いという理由で多くのスーパーコンピュータシステムで採用されている並列ファイルシステムである．Lustre は，１台の MDS（Metadata Server）と複数台の OSS（Object Storage Server）の二種類のサーバで構成される．MDS は，ファイル実体の格納先，および，ファイルの情報（作成時間・更新時間など）のメタデータを管理格納している．一方 OSS は OST（Object Storage Target）というディスクデバイスを管理しており，OST にファイルの実体が格納される．Lustre は，1 つのファイルをオブジェクトという単位で分割し，複数の OST に分散して格納するストライピングを採用している．ストライピングにより，複数の OST に並列に I/O を行うことができるため，高いスループットを得ることができる．ストラピングのパラメタはユーザがあらかじめ定められた上限のなかで自由に設定でき，I/O 性能のチューニングを行うことができる．分割するデータサイズであるストライプサイズや，分散させる OST の数であるストライプカウントがパラメタとして変更される．また，どの OST から順に配置するかという，ストライプオフセットもパラメタとして設定できるが，値を変更して I/O 性能をチューニングすることはあまり行われておらず，システムがファイル作成時にラウンドロビンで決定する値を使うことが多い．

次に，Lustre において，クライアントが write 関数を呼んだ際の クライアントとサーバ間のプロトコルを，図 1，2 を用い簡単に説明する．図 2 は，O_SYNC 等の同期オプションをつけて，同期モードでファイルアクセスをする場合，図 1 は，同期モードではなくクライアント側でバッファが用いられる場合を示している．

まず，バッファを用いる場合を説明する．アプリケーションが write 関数を呼ぶと，クライアントプロセスのメモリ上にある max_dirty_mb の大きさをもつバッファに一時格納される．このときバッファからあふれたり，sync 関数が呼ばれたときに，OSS に対してデータが転送される．このデータ転送は，Remote Procedure Call (RPC) によりストレージサーバの I/O スレッドを起動して行われ，I/O スレッドは，クライアントからサーバへのデータ転送の受信，および，OST に対してのデータ書き込みを行う．1 回の RPC では，max_pages_per_rpc で設定されるページ数のデータが転送され，クライアントとサーバの間では，複数の RPC が同時に実行される．1 クライアントと 1 サーバの間で同時に実行される RPC の数は，max_rpcs_in_flight で設定される．これら，max_dirty_mb や max_pages_per_rpc，max_rpcs_in_flight はシステム管理者が設定するもので，一般ユーザが変更することはできない．

一方，O_SYNC 等の，同期オプションを伴う場合は，write 関数を呼ぶと RPC が即座に起動され，バッファを介さずにストレージサーバに対してデータの転送が行われる．また同期オプションが伴う場合は，バッファが用いられないというだけでなく，stripe_size ごとに書き込みが同期されるという振る舞いになる 2.

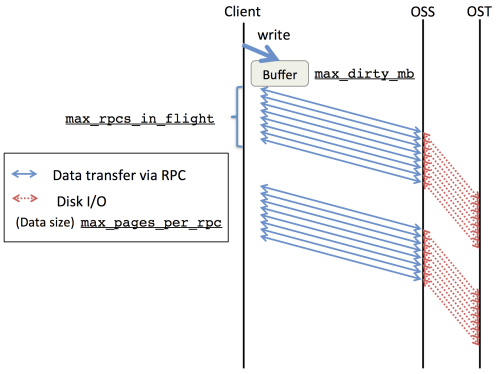


図 1 Lustre での Write プロトコル (バッファモード)
Fig. 1 Write protcol on the Lustre File system (Buffered)

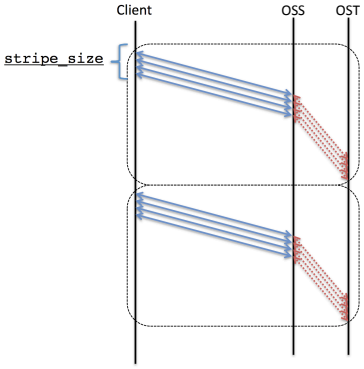


図 2 Lustre での Write プロトコル (同期モード)
Fig. 2 Write protcol on the Lustre File system (Synchronous)

表 1　ファイルシステム設定値
Table 1　Parameters of File System

max_dirty_mb	8 MB
max_pages_per_rpc	128 pages (1 MB)
max_rpcs_in_flight	8 rpcs

表 2　評価パラメタ
Table 2　Parameters of Experiment

Number of Clients	1, 2, 4, 8, 16
Stripe Count	1, 2, 4
Stripe Size	1 MB, 2 MB, 4 MB, 8 MB
File Size	64 GB / Number of Client
Write Size	512 MB
O_SYNC Option	On, Off

2.2　ストライピングによる I/O 性能の測定

まず，ストライピング等のパラメタの違いによる，ファイルの書き込み性能の違いを 2OSS/4OST を用いて測定した．本評価においては，管理者権限が必要であるようなパラメタについては変更せずにシステムインストール時の設定を用いることとし，ユーザレベルで変更できるパラメタについてのみパラメタ値を変更して測定を行った．ファイルシステムの設定値を表 1 に，ユーザレベルで変更できる評価パラメタを表 2 に示す．

実験では，各クライアントプロセスそれぞれが 1 つずつファイルを作成し同時に書き出しを行った際のトータルスループットを測定した．測定結果を図 3 に示す．

棒グラフの色の違いはクライアント数の違いである．クライアント数が 1 から 2，4 と増えるに従って性能がよくなりピークを境に低下している．

次に，同期の有無の違いであるが，図 3 の左半分が O_SYNC をつけない場合であり，右半分が O_SYNC をつけた場合である．O_SYNC の有無により山のピークが異なり，O_SYNC をつけい場合はクライアント数 4，O_SYNC の場合クライアント数 8 が山のピークとなっている．

ストライピングによる性能差であるが，ストライプカウントが 1 の場合に，山のピークの性能が最も高く，ストライプカウントが大きいほど性能が低下した．また，ストライプサイズによる性能差はみられなかった．

ストライプカウントが大きくなるにつれ，1OST あたりにアクセスしてくるクライアントの数が増える．例えば 4 クライアントと 4OST の場合，ストライプカウントが 1 の t ときは，各 OST ともに 1 クライアントからアクセスされる．一方で, ストライプカウントが 4 の場合は，各 OST ともに 4 クライアント全てからアクセスされる．OST に対してアクセスしてくるクライアント数が増えると性能が低下するものと考えられる．

2.3　並列 RPC 数の違いによるスループット性能の測定

先の実験では，OST に同時にアクセスするクライアント数と性能に相関があるという結果になった．先の実験では，1 クライアントプロセスと 1 サーバ間の 並列な RPC の数が固定であった．つまり，OST に同時にアクセスするクライアント数に比例してサーバで同時に処理する RPC が増えていた．そこで，書き込み先の OSS および OST を 1 台に限定し，また，クライアント数，および max_rpcs_in_flight を変更することで，1 OST に対して同時に処理される RPC 数を制御してスループットの違いを測定した．

実験では，クライアント数を 1 から 8，max_rpcs_in_flight を 1 から 8 と変更して，全クライアントから 1OST に対する同時書き込みスループットを測定した．本実験においては，バッファの影響をなくすために，O_SYNC オプションをつけて同期的に I/O を行うようにし，Stripe Size ごとに同期されてしまうが，同期の影響を軽減するために，1MB × max_rpcs_in_flight より十分大きい 128 MB とした．

図 4 に測定結果を示す．横軸に，クライアント数と max_rpcs_in_flight の積をとり，縦軸にスループットをプロットしている．クライアント数ごとに，プロットのシンボルを変えている．横軸の，クライアント数と max_rpcs_in_flight の積は，OST に対して同時に書き込みを行う RPC の個数の最大値である．この値が 4 をピークに，値が増えるにつれてスループット性能が低下している．また，クライアント数と max_rpcs_in_flight の積が同じ場合でも，クライアント数が 2 の場合が最も性能が良いのを境にクライアント数が増えるにつれて性能が低下している．

図 5 は，本実験の実行中に，OSS にて iostat コマンドを実行し，OST に対する書き込み負荷を測定した値を示している．この値は，100 に近いほど，ディスクデバイスの性能が上限に達していることを表している．クライアント数が 1 のときや，クライアント数と max_rpcs_in_flight の積が 4 未満の場合は，値が小さく，ディスクデバイスの性能を使い切れていないことがわかる．これらの場合は，クライアントからサーバへのデータの転送スループットが，ディスクデバイスへの書き込みスループットより小さくなっているといえる．一方で，ディスクデバイスに対する書き込みリクエストが増えると，ディスクデバイスの性能が上限に達している．クライアント数が多い場合や，書き込みリクエストそのものが増えると，ディスクへの write リクエストがキューにてマージされにくくなり，ディスクへの書き込みサイズが小さくなりがちになるために，書き込み性能が低下していくものと考えられる．

OST への書き込み性能を最大限に引き出すためには，ディスクデバイスに負荷をかけすぎないよう，クライアン

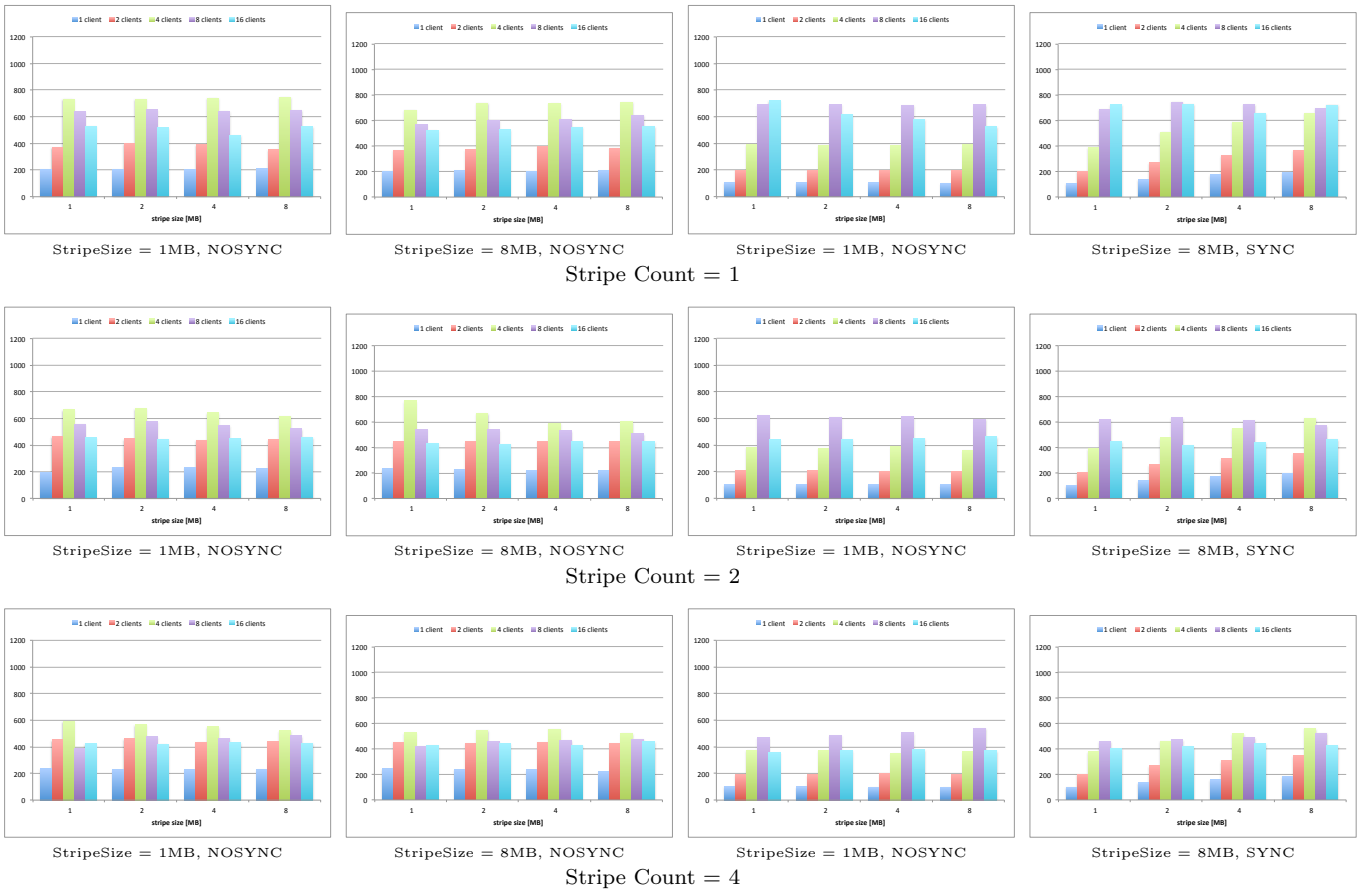


図 3 パラメータの違いによるスループット
Fig. 3 Throughputs over various parameters

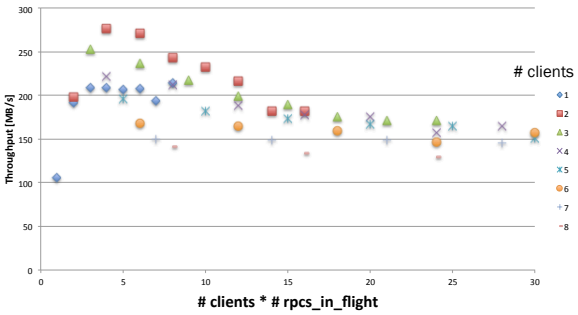


図 4 max_rpcs_in_flight とスループットの関係
Fig. 4 Relation of Throughput and max_rpcs_in_flight

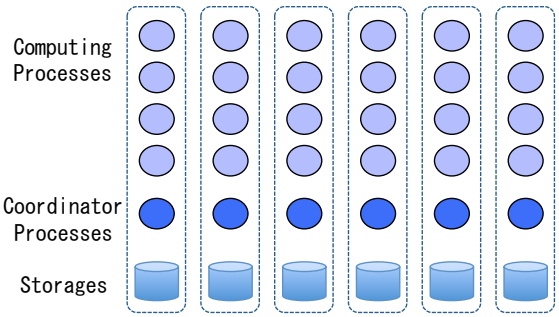


図 6 I/O 調停機構のプロセス構成
Fig. 6 Process Organizations of I/O Coordination

# rpc_in_flight \ # client	1	2	3	4	5	6	7	8
1	31.11	60.01	80.41	92.85	97.34	98.82	99.63	99.87
2	56.72	88.03	97.43	99.44	99.77	99.92	99.95	99.99
3	61.01	96.34	99.49	99.90	99.96	93.74	99.97	99.50
4	61.33	98.28	99.81	99.97	100.00	100.00	99.91	99.99
5	61.15	98.91	99.96	99.96	99.99	99.99	100.00	99.19
6	60.46	99.18	99.61	99.94	99.95	99.98	99.99	99.99
7	66.60	99.44	100.00	99.97	98.18	100.00	99.99	99.33
8	63.21	99.85	99.98	99.99	99.99	99.99	99.99	99.99

図 5 iostat コマンドの結果
Fig. 5 results of iostat command

トから I/O の集中を防ぐことが有効であると考えられる．

3. I/O 調停機構

前章の評価結果を踏まえ，I/O 調停機構を提案する．I/O 調停機構では，ストレージサーバに並列にアクセスするアプリケーションプロセスからの I/O の数を制限するように，アプリケーションプロセスからの I/O 要求を調停し，ストレージサーバの負荷を制御する．

図 6 に I/O 調停機構に関わるプロセス構成を示す．I/O 調停機構では，計算プロセスおよびストレージをグループに分割し，グループ毎にグループ内のストレージにのみアクセスをするようにする．そして，各グループに 1 つの調

Algorithm 1 Computing Processes

```
Send WRITE_REQUEST to Coordinator
Wait ACK_REQUEST from Coordinator
Write data to Storage Target
Send COMPLETION_MESSAGE to Coordinator
```

Algorithm 2 Coordinator Processes

```
Set RequestQueue empty.
writer_cnt ← 0.
while Computing Processes are alive. do
  Wait message from Computing Processes.
  if message is WRITE_REQUEST then
    Get request_proc_id from message.
    Enqueue request_proc_id to RequestQueue.
  else
    /* message is COMPLETION_MESSAGE */
    writer_cnt ← writer_cnt - 1.
  end if
  while writer_cnt < MAX_WRITER_CNT and RequestQueue is not empty do
    Dequeue request_proc_id from RequestQueue.
    Send ACK_REQUEST to Computing Process shown by request_proc_id.
    writer_cnt ← writer_cnt + 1.
  end while
end while
```

停プロセスを設ける．この調停プロセスは，グループ内のストレージに対する計算プロセスからの I/O 要求をうけ，ストレージサーバに同時にアクセスする計算プロセス数を制限するように調停を行う．

以下に，計算プロセスと調停プロセスの処理の流れを説明する．アルゴリズム 1 に計算プロセスの処理を示し，アルゴリズム 2 に調停プロセスの処理を示す．

計算プロセスの write 関数を hook しており，write 関数のはじめに，まず対応する調停プロセスに *WRITE_REQUEST* を送る．そして，調停プロセスからストレージアクセス許可となる *ACK_REQUEST* が届くまで待機する．その後，計算プロセスは，*ACK_REQUEST* が届くことででストレージサーバ にアクセスすることができる．そして，ストレージサーバへのデータ書き込みが完了すると，調停プロセスに対して *COMPLETION_MESSAGE* を送り，書き込み操作を完了する．

調停プロセスの役割は，対象のストレージに対する負荷を管理するために，計算プロセスからの I/O 要求を調停する事である．ストレージの負荷を管理するために，調停プロセスは，ストレージにアクセス中である計算ノード数 (アルゴリズム 2 中の writer_cnt) をカウントしている．調停プロセスは，計算プロセスからのメッセージを受けて動作し，メッセージが無い限り何もしない．調停プロセスは，計算プロセスから *WRITE_REQUEST* を受けたとき， *WRITE_REQUEST* を送ってきた計算プロセスを識別する request_proc_id を RequestQueue に登録する．一方， *COMPLETION_MESSAGE* を受けた場合は，ストレージにアクセス中である計算ノード数を 1 減らす．これらメッセージを受けて上記処理を行った後に，ストレージにアクセス中である計算ノード数が，規定値以下であり，かつ，RequestQueue が空でない限り，RequestQueue から request_proc_id を取り出し， request_proc_id に対応する計算ノードに対して *ACK_REQUEST* を送り， ストレージにアクセス中である計算ノード数を 1 増やす．そして，再び計算プロセスからのメッセージを待ち続ける．

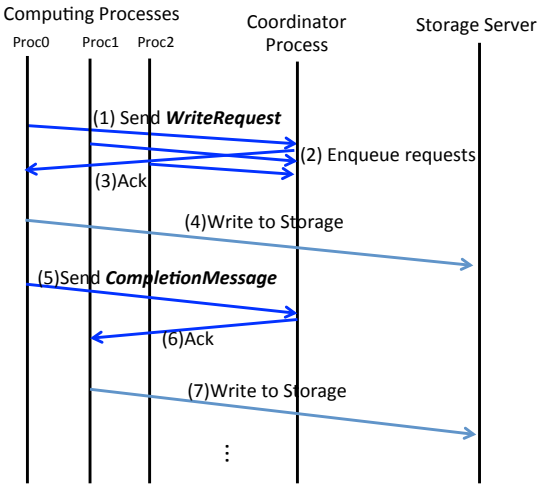


図 7 I/O 調停機構を用いた場合のシーケンス
Fig. 7 Sequence of I/O Coordination Technique

図 7 に，3 プロセスが同時に，1 つのストレージサーバに書き込みを行う場合を例に，I/O 調停を行う場合のシーケンスを示す．この例では，ストレージサーバに対して同時に書き込みできるプロセスの数を 1 に制限している．

(1) 各計算プロセスは，ストレージサーバに書き込む前に， 調停プロセスに対し，*WRITE_REQUEST* を送信する．(2) 調停プロセスは，届いた *WRITE_REQUEST* を順に RequestQueue に追加し，(3) RequestQueue から 1 つリクエストを取り出し，そのリクエストの送信元であるプロセス 0 に対して，*ACK_REQUEST* を送信する．(4) プロセス 0 は，調停プロセスからの *ACK_REQUEST* を受けて，ストレージサーバに対して データの書き出しを行う．(5) プロセス 0 は，データの書き出し完了後，調停プロセスに対して，*COMPLETION_MESSAGE* を送信して，Write 処理が完了する．(6) 調停プロセスは，*COMPLETION_MESSAGE* を受けて，RequestQueue から 1 つリクエストを取り出し，そのリクエストの送信元であるプロセス 1 に対して，*ACK_REQUEST* を送信する．(7) プロセス 1 は，調停プロセスからの *ACK_REQUEST* を受けて，ストレージサーバに対してデータの書き出しを行う．

以上の処理により，調停プロセスは，計算ノード I/O 要求を調停し， ストレージに同時にアクセスできる計算ノードの数を規定数以下にする．

我々は，これら調停処理を行う I/O Coordination Library (IOC Library) を実装した．我々の IOC Library では TCP Socket 通信を用いており，調停プロセスは，アプリケーションプロセスと計算ノードで共存できるように軽量な実装としている．write, pwrite 等のシステムコールは，IOC Library 用に置き換えており，アプリケーション側で書き換えなく利用できる．ストレージに対する同時アクセスノード数 *MAX_WRITER_CNT* は，パラメタとしてユーザが設定することができる．

4. 性能評価

I/O 調停機構の評価のために，スーパーコンピュータ「京」[6] において IOC Library を用いてベンチマークプログラムを実行した．

4.1 スーパーコンピュータ「京」

「京」では，使用計算ノード数を指定すると，ジョブスケジューラによって各ジョブに対して 3 次元トーラスでつながれた計算ノードが割当てられる．図 8 で示すように，ジョブが割り付けられた計算ノードに応じて利用できる OSS/OST を分割しており，ジョブ間の I/O の衝突を軽減している．例えば，図 8 は，Job(a) は，他のジョブからは独立してストレージ資源を占有して利用できる．ただし，全てのジョブが独立してストレージ資源を利用できるわけではなく，ジョブの計算ノード数によっては，Job(b) と Job(c) のように複数のジョブで共有することもある．これは，ジョブが割り当てられた計算ノードと 3 次元ネットワークにおいて同一 Z 軸上にある OSS/OST のみを利用するという仕組みで実現している．我々の IOC Library の実装では，この仕組みを応用し，各計算プロセスは，計算ノードと 同一 Z 軸上の OST にファイルを格納するようにし，Z 軸に 1 プロセスずつ調停プロセスを設けることにした．なお，調停プロセス用に物理的にノードを用意はせず，1Z 軸に 1 計算ノードは，計算プロセスと調停プロセスが共存して実行される．

4.2 Micro Benchmark

まず，IOC Library を適用する前に，前章で説明した *MAX_WRITER_CNT* を決定するために，事前測定を行った．本評価では，京の Z 軸を占有する 192 ノード (1Z 軸あたり 16 ノード × 12Z 軸) を用い，対応する 6OSS/12OST を用いて実験を行った．実験で用いたプログラムは，1 ノード 1 プロセスの MPI アプリケーションで，各プロセスが 1GB の write を行うものである．この並列プログラムを，Z 軸あたりのプロセス数を 1 から順に 1 ずつ 16 まで増やして実行し，OST あたりのスループットを測定した．今回の実験では，プロセス数が OST 数より十分大きいため，ストライピングは行わないようにストライプカウント 1 とした．またストライプサイズは 16MB とした．

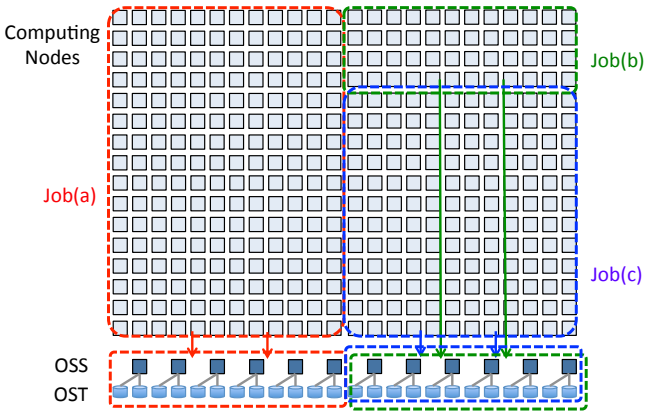


図 8 京におけるジョブ間の I/O の分離例
Fig. 8 An example of I/O splitting between jobs on the K computer

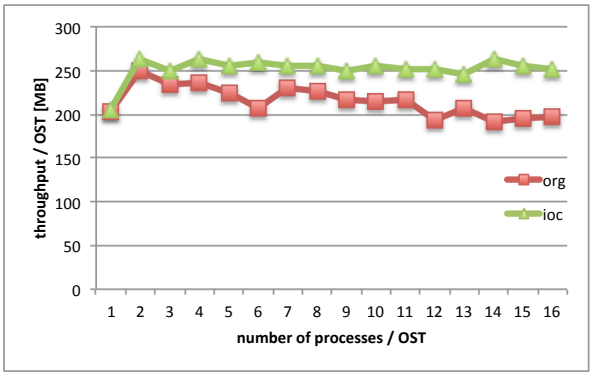


図 9 「京」におけるスループット性能
Fig. 9 Throughput in K Computer

図 9 の org がその結果である．OST に対するクライアント数が 2 の場合が最も性能が良い．そこで，以降の我々の実験においては，*MAX_WRITER_CNT* を 2 にした．

図 9 の ioc は，先の評価条件において，IOC Library を適用した場合の結果である．I/O の調停を行うことで，クライアント数が増えても性能が低下することなく，org の場合の最高性能を維持できていることがわかる．

図 10 には，本実験における，各計算ノードでの I/O の所要時間を昇順にソートして表示している．各グラフの最も右側の棒が最も I/O が遅く終わったプロセスの 実行時間である．図 10 の左図は，I/O の調停を行わない場合の write 関数 にかかった時間を表示している．右図は，I/O 調停を行った場合で，調停プロセスにリクエストを送ってから，調停プロセスから許可がでるまでの時間 (wait) と，ストレージサーバに対してデータを書き出す時間 (write) を積み重ねグラフで表示している．

I/O の調停を行うと，1OST に対して 2 プロセスずつのアクセスに制限するので，2 プロセスの write が終わると，次の 2 プロセスが OST に対してのアクセスを開始する．従って，I/O の完了時間は図 10 の右図のように階段状に

なる．一方，調停を行わない場合は，OST に対してアクセスが集中するため，全プロセスともに I/O が完了する時間が長くなってしまう．このように，1 OST に対する 同時にアクセスするノード数を制限することで，最終的に全ノードの write が完了する時間が短縮できていることがわかる．

4.3 アプリケーションへの適用

実アプリケーションを用いた性能評価として，気候シミュレーションプログラム NICAM (Nonhydrostatic ICosahedral Atmospheric Model) [7] に I/O Coordination Library を適用した．NICAM は，正 20 面体格子を用いた 全球雲解像実験用の気候シミュレーションプログラムである．NICAM は，MPI で並列化されており，地球の全球を表現した正 20 面体格子を計算プロセス数で分割し，各計算プロセスでは割当てられた格子内の計算を行う．そしてシミュレーション結果をプロセス個々のファイルに書き出すようなアプリケーションである．

NICAM では，History-file と Restart-file と呼ばれる 2 ファイルをあらかじめ定められた時間間隔で定期的に各計算プロセスが書き出す．Hisotry-file は，アプリケーションユーザが関心のあるシミュレーション結果を残すためのファイルであり，Restart-file は，シミュレーションの続きを再実行するために必要な情報を残すためのファイルである．これら 2 ファイルとも，PaNDa (packaged NICAM data format) という，独自のバイナリデータフォーマットで書き出しのたびに追記される．

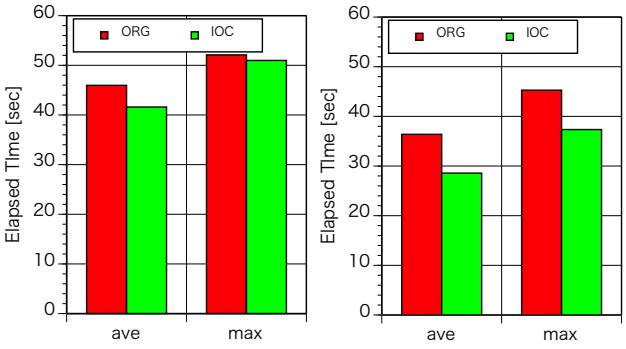
表 3 に，実験に使用した環境，および，実験設定を記載する．我々は，glevel-10 (Grid division level 10)，および glevel-11 と呼ばれる問題サイズで NICAM を実行した．それぞれ，地球全体を 7.0 km 格子 (glevel-10)，3.5 km 格子 (glevel-11) の解像度でシミュレーションをすることができる．これらのサイズのシミュレーションを，「京」の 320 ノード (glevel-10) および，1280 ノード (glevel-11) を用いて実行した．実験では，320 ノードの場合，4x3x28 ノード，24 OST を，1280 ノードの場合，2x27x24 ノード，108 OST がシステムにより割当てられた．

実験で用いた設定では，どちらのノード規模においても，1 ノードあたりの格子数は等しく，出力するデータサイズは全ノード同じである．History-file は，1 回のファイル出力では 15 変数 (計 146MB) を連続で出力し，シミュレーション中に 5 回出力を行う．Restart-file は，シミュレーションの最後に 1 回 29 変数 (計 400MB) を連続で出力する．1 変数あたりのデータサイズは 3 次元データで 12.1MB，24.7MB のどちらか，2 次元データで 132KB，264KB，528KB のいずれかである．

132KB，264KB，528KB の書き出しに関しては，調停プロセスを経由させることによるコストの方が大きいと考えられる．IOC Library のパラメタとして，調停プロセスを経由させる Write Size を 8MB 以上 と設定することで，これらの小さいサイズの write 関数呼び出しについては調停プロセスを経由させないようにした．

表 3 評価環境および設定
Table 3 Enviornments and configurations of experiments

Number of processes		320	1280
Number of computing nodes allocated		4x3x28	2x27x24
Number of available OSTs		24	108
Data Size	History-file	146MB × 5 times	
	Restart-file	400MB × 1 time	

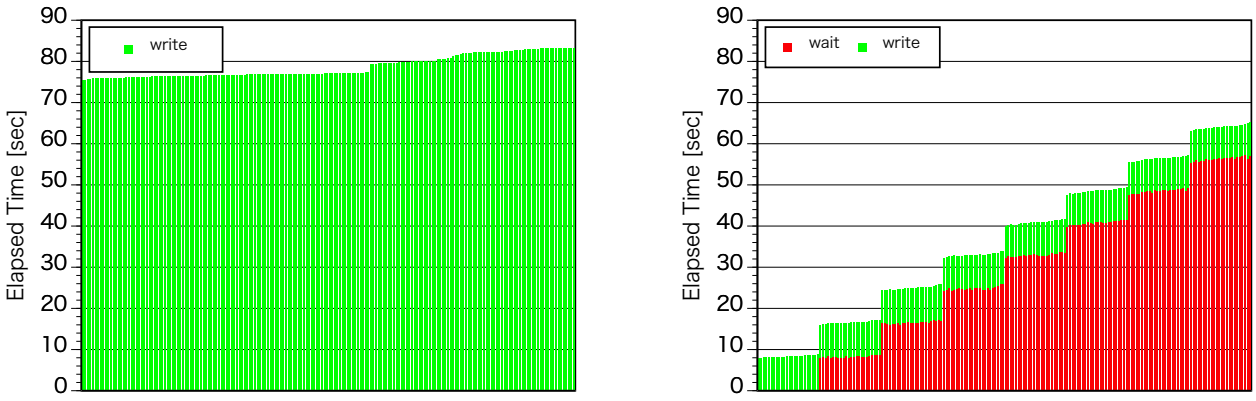


(a) 320 Nodes (b) 1280 Nodes

図 11 NICAM におけるファイル書き出し時間
Fig. 11 File Output Time in NICAM

この実験設定において，ファイル書き出し時間を各計算ノードで測定した．比較対象として「京」のランクディレクトリ機能を使用した場合を測定した．ランクディレクトリとは，ジョブ内のプロセス間の I/O の衝突を軽減する仕組みで，「京」に実装されている機能である．各並列プロセスごとに計算ノードと同一の Z 軸にある OSS および OST にデータをが格納されるようなストライピングの設定を，あらかじめシステム側で固定することで OST の負荷の分散を行っている．なお，2014 年 1 月より，仕様が変更され，各プロセスにはストライプカウントが 1 のディスクイメージファイルが割り当てられ，ディスクイメージをループバックマウントしたものをランクディレクトリとして提供している．ディスクイメージのファイルは，仕様変更前と同様に 同一 Z 軸にある OSS および OST に格納される．なお，本実験は仕様変更前のランクディレクトリを使用したものである．

図 11 に，全ノードのファイル書き出し時間の平均 (ave) および，最大値 (max) を示す．図中の ORG がランクディレクトリを使用した結果で，IOC が IOC Library を用いた結果である．ファイル書き出しにかかる時間は，平均で，320 ノードの場合 10%，1280 ノードの場合 20 % 短縮することができており，実アプリケーションにおいても I/O 調停機構は効果があるといえる．



(a) Original (b) I/O Coordination

図 10 Write にかかる時間
Fig. 10 Elapsed Time of Write Operation

5. 関連研究

Song ら [8] は，並列ファイルシステムに対する I/O リクエストを並列ファイルシステムのサーバで調停する Server-side I/O Coordination を提案している．並列 I/O にかかる時間とは，最も遅く I/O が完了したプロセスの完了時間と考えることができる．Server-side I/O Coordination では， 複数アプリケーションからの I/O リクエストを調停し，アプリケーションごとに I/O リクエストの処理タイミングをファイルサーバ間で揃えることで，並列アプリケーションの各プロセスの I/O の完了時間を均一化している．複数のアプリケーションが共有して並列ファイルシステムにアクセスする場合に，それぞれの並列アプリケーションごとにプロセス間の I/O 完了時間をばらつきを抑えることで，全並列アプリケーションの平均 I/O 時間の削減を達成している．

並列ファイルシステムの負荷状況を管理する事で並列ファイルシステムの性能を向上する手法として，Adaptive I/O がある [9]．Adaptive I/O ではファイルシステムの負荷状況を観察し，並列ファイルシステムの負荷の大きいストレージサーバから，負荷の少ないストレージサーバに，動的に負荷をシフトすることでシステム全体の性能向上をはかっており，ストレージサーバの負荷がインバランスな場合に特に効果を発揮する．

I/O forwarding [10,11] は，計算ノードが発行する I/O リクエストを I/O 専従のノードに移譲し，I/O 専従ノードのみがファイルシステムにアクセスすることで，ファイルシステムの負荷を権限している．また，I/O 専従ノードにおいて，計算ノードからの I/O リクエストを集約，スケジューリング，キャッシュを行うことができ，並列ファイルシステムの負荷を軽減している．

我々の提案する I/O Coordination の特徴は，並列アプリケーションプロセス間だけで，I/O の調停を行う．そのため，特別にファイルシステムに対して追加する必要もなくユーザレベルにて I/O 性能を向上することができる．また，ファイルシステムのクライアントプロセスとサーバとの間の データ転送には関与しないため，調停用の小さいメッセージのやりとりだけで，追加のデータ転送が必要とならないため，ネットワークの負荷をかけることがない．

6. まとめ

本論文では，並列ファイルシステムのスケーラビリティ問題の 1 つとして，ストレージサーバに対して同時にアクセスするクライアント数が増えるにつれて，I/O 性能が低下するという問題を提示し，解決策として I/O 調停機構を提案した．

I/O 調停機構では，調停プロセスを設け，調停プロセスに計算ノードからの I/O 要求を集め，ストレーシサーハにして I/O を行うするクライアント数を制限するような調停をすることで，ストレージサーバの負荷を軽減し，I/O 性能低下を防ぐことができる．我々は，I/O 調停機構を実装し，マイクロベンチマークプログラムを用いて性能評価した．その結果，調停を行わな場合と比べて write スループットが 25 % 向上することを示した．また，実アプリケーションとして 気象シミュレーションアプリ NICAM に適用したところ，I/O にかかる時間を最大 20 % 削減できることを示した．

I/O の調停を行う事で，性能低下なく ストレージサーバの I/O 性能を最大限利用できることを示した．

本稿では，I/O パターンを全プロセスが個別のファイルに対して write を行う場合に限定して議論した．しかし，プロセス間で 1 つのファイルを共有して I/O を行う場合，read を含めた I/O など，I/O パターンを広げて議論していく必要がある．

謝辞 本論文の結果の一部は，理化学研究所のスーパーコンピュータ「京」を利用して得られたものである．

参考文献

[1] Sun Microsystems, Inc: *Lustre File System: High-Performance Storage Architecture and Scalable Cluster File System* (2007).
[2] Haddad, I. F.: PVFS: A Parallel Virtual File System for Linux Clusters, *Linux Journal*, Vol. 2000 (2000).
[3] Fadden, S.: *An Introduction to GPFS Version 3.5* (2012).
[4] 石川裕大野善之：並列ジョブのファイル I/O をひとつのファイルに集約する方式の提案と予備評価，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. 2011, No. 34, pp. 1–6 (2011).
[5] Sakai, K., Sumimoto, S. and Kurokawa, M.: High-Performance and Highly Reliable File System for the K computer, *FUJITSU SCIENTIFIC and TECHNICAL JOURNAL*, Vol. 48, No. 3, pp. 302–309 (2012).
[6] Miyazaki, H., Kusano, Y., Shinjou, N., Shoji, F., Yokokawa, M. and Watanabe, T.: Overview of the K computer, *FUJITSU SCIENTIFIC and TECHNICAL JOURNAL*, Vol. 48, No. 3, pp. 255–265 (2012).
[7] Satoh, M., Matsuno, T., Tomita, H., Miura, H., Nasuno, T. and Iga, S.: Nonhydrostatic icosahedral atmospheric model (NICAM) for global cloud resolving simulations, *J. Comput. Phys.*, Vol. 227, No. 7, pp. 3486–3514 (2008).
[8] Song, H., Yin, Y., Sun, X.-H., Thakur, R. and Lang, S.: Server-side I/O coordination for parallel file systems, *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, New York, NY, USA, ACM, pp. 17:1–17:11 (2011).
[9] Lofstead, J., Zheng, F., Liu, Q., Klasky, S., Oldfield, R., Kordenbrock, T., Schwan, K. and Wolf, M.: Managing Variability in the IO Performance of Petascale Storage Systems, *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. 1–12 (2010).
[10] Iskra, K., Romein, J. W., Yoshii, K. and Beckman, P.: ZOID: I/O-forwarding infrastructure for petascale architectures, *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, New York, NY, USA, ACM, pp. 153–162 (2008).
[11] Ali, N., Carns, P. H., Iskra, K., Kimpe, D., Lang, S., Latham, R., Ross, R. B., Ward, L. and Sadayappan, P.: Scalable I/O forwarding framework for high-performance computing systems., *CLUSTER*, IEEE, pp. 1–10 (2009).