Vol.2009-HPC-121 No.14
2009/8/4

ファイルステージングシステム Catwalk の MPI-IO 実装

堀　敦史[†1]　鴨志田　良和[†1]　松葉浩也[†1]
安井　隆[†2]　住元真司[†3]　石川　裕[†1]

我々はファイルステージングをオンデマンド化し，パイプラインでファイルを転送することでスケーラブルなバンド幅を得ることができる Catwalk を既に開発している．本稿では，この Catwalk をベースに MPICH2 の ROMIO 用の ADIO デバイス，Catwalk-ROMIO，を開発した．ファイルの読込性能は，Catwalk 同様，パイプライン転送によりバンド幅を向上させている．本稿ではファイルの書込においては，各ノードのローカルディスクにジャーナルファイルを生成し，ファイルの close あるいは flush 時に，各ノードのジャーナルファイルの内容をサーバノードに転送し，書込んだ内容をサーバ上のファイルに反映させる，という方式を提案する．Catwalk-ROMIO による MPI-IO でのファイルの書込はノードのローカルディスクに対して独立であり，通信が発生しないため，スケーラブルに実行可能である．実装された Catwalk-ROMIO は T2K（東大） 上で評価し，MPI-IO を通して単一のファイルサーバ上のファイルを効率的に並列 IO 可能であることを示す．

An Implemenation of an MPI-IO Using Catwalk File Staging System

Atsushi Hori,[†1] Yoshikazu Kamoshida,[†1]
Hiroya Matsuba,[†1] Takashi Yasui,[†2]
Shinji Sumimoto[†3] and Yutaka Ishikawa[†1]

A new ROMIO ADIO device, Catwalk-ROMIO, is proposed and developed based on the Catwalk on-demand file staging system where high aggregate bandwidths of read operation are obtained by using the pipelined file transfer. In this papaer, it is proposed that the write transaction data are journaled on local disks of compute node and the journal records are merged so that the target file on the file server is updated at the close or sync operation. The local journaling is completely localized and therefore can be parallelized, and scalable aggregate bandwidths can be obtained. The implemented Catwalk-ROMIO is evaluated on the T2K cluster (The University of Tokyo). The results will show an effective parallel IO is implemented with single file server node.

1. 概　　要

我々はファイルステージングをオンデマンド化することで，分散あるいは並列ファイルシステムと同様の使い勝手を実現したファイルステージングシステム，Catwalk，を開発した[10]．Catwalk においての入力ファイルのステージイン（ファイルサーバから計算ノードへのファイルコピー）は，リング状の分散プロセス構造（図 2）に沿ったパイプライン転送により実現されているため，転送速度はノード数に依らずサーバの転送速度により決まるという性質を持つ．このため，ひとつのファイルを並列ジョブが参照する場合には，単一サーバの性能を最大限に引き出すことが可能である．しかしながら，出力ファイルに対しては，各プロセスが個別に出力したファイルしか扱えないという欠点があった．このため，入力ファイルは並列ジョブの個々のプロセスで共有できるが，出力ファイルは共有できない，という非対称性がある．また，あるジョブが出力したファイルを別のジョブで読込む，といったジョブ間のファイル経由によるデータの受け渡しが簡単ではない．このためオンデマンド化によりステージングの記述が不要でかつ共有ファイルシステムのように見えるという特性を十分に活かすことができていなかった．

本稿ではこの単一の出力ファイルが共有できないという欠点を克服するために，プロセスからの書込を，いったんジャーナル形式でローカルディスクに保存し，ファイルの close 時あるいは sync 要求のあった時点で，各ノードのジャーナルファイルをサーバーノードに転送し，書込内容をサーバ上のファイルに反映させるという方式を考える．この方式の利点は，書込は各ノードで独立かつ並列におこなうことができ，スケーラビリティが確保できることである．ファイルの close や flush 時には，Catwalk の分散リングに沿ってサーバノードに転送されるので，その転送速度はノード数に依らずにサーバノードのファイル書込速度あるいはネットワークの速度で決まるものと期待される．

この方式を，オンデマンドファイルステージングシステム Catwalk をベースに，MPICH2

†1 東京大学 / The University of Tokyo
†2 日立製作所 / Hitachi, Ltd.
†3 富士通研究所 / Fujistu Laboratories, Ltd.

の MPI-IO の実装である ROMIO の ADIO デバイスとして実装した．実装された ADIO デバイスは Catwalk-ROMIO と呼ばれる．本稿では実装方式，予備評価による基本性能の確認，そして今後の方向性について述べる．

2. 背　　景

図 1 はクラスタの規模とユーザ数の関係を示したピラミッド図である．PVFS[1] や Gfarm[8] といった並列ファイルシステムは，複数のファイルサーバとクラスタと接続するネットワークにより高い並列アクセスバンド幅を実現している．しかしながら，小規模なクラスタでは，並列ファイルシステムに投資するよりも，計算ノード数を増やす傾向にあり，NFS に代表される並列でない分散ファイルシステムに依存することが多い．

昨今のノードのコア数の顕著な増加は，NFS により高い負荷をかけ，４ノードという小規模なクラスタでも，使うに耐えない結果となることがある[2]．図 1 にあるように，小規模クラスタの台数およびユーザ数は，潜在的ではあるが，かなり多いと考えられ，決して無視できるものではない．

我々が開発した Catwalk は TCP/IP ネットワークと単一のファイルサーバしか仮定していないため，どのようなクラスタでも実行可能であると同時に，NFS より高い性能を示すことが既に示されている[2]．

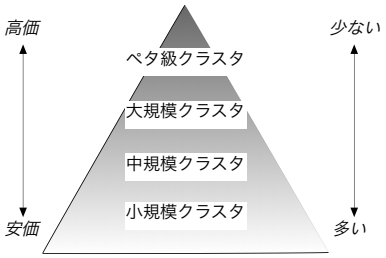

図 1　クラスタのユーザピラミッド

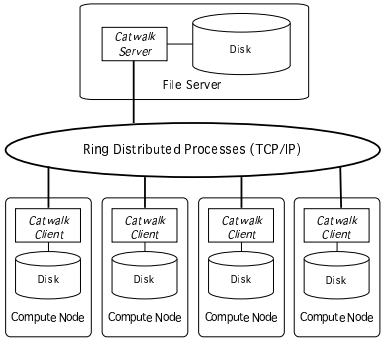

図 2　リング分散プロセス構造

3. Catwalk から Catwalk-ROMIO へ

Catwalk の大きな特徴として，内部の通信トポロジーとして図 2 に示した分散リング構造がある．特に読込みファイルの転送には，このリングに沿ったパイプライン転送により，ノード数に比例した総バンド幅を実現可能である[10]．Catwalk のもうひとつの特徴は，オンデマンド化であり，これによりユーザはファイルステージングの記述から解放される．オンデマンド化においては Linux の `LD_PRELOAD` 機構を用いて `open()` 等のシステムコールに hook することで実現されている．これにより，ユーザプログラムになんら変更を加えることなく，Catwalk 経由でサーバのファイルをアクセスすることができる．

Catwalk では，読込 `open()` があった場合，ファイルサーバーにあるファイルを全てのノードにパイプラインでコピーするが，`read()` システムコールを hook していないため，読込み範囲の情報がないこともあって，ファイル全体が各ノードにコピーされる．しかしながら，パイプライン転送の特徴から，１回のファイルコピーの手間で，全ノードにコピーできるため，わざわざ `read()` に hook する必要性は少ないと考えられている．一方，書込においては，各ノードの書込 `open()` の hook により，書込があったファイルの情報を得ることができ，書込があったファイルはジョブの終了時にサーバにコピーされる．この時，書込ファイルはユーザプロセス間で共有するしても一貫性は保証されないため，ユーザプロセスのそれぞれがファイルに書込むような場合は，別なファイルとするのが現実的である．この結果，Catwalk の入力ファイルはプロセス間で共有できるが，出力ファイルは共有できない，という非対称はファイル IO となっている．

ここで，個々のプロセスのファイル書込をジャーナル形式でローカルディスクにバッファする方式を考える[⋆1]．ファイルの `close()` 時に個々のジャーナルの内容をサーバーのファイルに書き戻すようにすれば，結果として書込ファイルを共有することが可能になる．これを Catwalk で実現するためには，`write()` システムコールに hook するだけでなく，`printf()` などの `write()` システムコールを呼ぶ glibc の関数全てに hook が必要になる．これは，glibc が glibc から呼ばれる glibc の関数に対して，`LD_PRELOAD` による hook を許していないからである．ここでの大きな問題は，例えば `printf()` 関数呼び出しにおいて，どのように `write()` 関数が呼ばれるかは自明でなく，`printf()` 関数の挙動を模擬する必要があ

⋆1 一般的にジャーナルとは，なんらかのタイミングで，まとめて更新作業をおこなうために，更新要求（ファイルでは書込に相当）をロギングすることである．

る．printf() は複雑な処理をおこなう関数なので，これは簡単なことではない．

MPI-IO におけるファイルへのアクセスは MPI ライブラリの関数として提供されているため，先に述べたローカルなファイル書込のジャーナリングを実現するためには，hook 関数の必要性がないという観点から都合が良い．そこで本稿では，MPICH2 での MPI-IO の実装である ROMIO[7] の ADIO デバイスとして Catwalk をベースに Catwalk-ROMIO を開発した結果について述べる．

ファイルステージングは地球シミュレータや理研 RSCC システムで実用化されているが，東京大学，筑波大学，京都大学に設置された T2K クラスタでは並列ファイルシステムによる運用がなされている．また，MPI2 においても MPI-IO が規格化されていることから，MPI-IO が並列ファイルシステムの違いを吸収し，有効な並列 IO プログラミングの手段のひとつと考えられる．このため，Catwalk もファイルステージングだけでなく MPI-IO の実装が望まれる．これは図 1 において，ピラミッドの全ての階層でシームレスなプログラミング環境を提供するという意味でも重要と考えられる．

本稿で提案する Catwalk-ROMIO が必要とするハードウェア要件は，Catwalk の同じく，単一のファイルサーバと TCP/IP ネットワークであり，ほとんどのクラスタで実行可能である．

4. Catwalk-ROMIO の実装

4.1 ROMIO

ROMIO[7] は MPI-2 の規格における MPI-IO におけるデファクト標準的な実装であり，ADIO と呼ばれる仮想 IO デバイスをサポートすることで，NFS や PVFS など多数のファイルシステムがサポートされている．ADIO デバイスを実装するには，ADIO 用に最大 22 の関数を定義すれば良い．基本的には POSIX の IO に準拠したデフォルトの関数が別途定義されており，POSIX と挙動が異なる関数のみ新たに定義するだけで ADIO デバイスが実装できる．ちなみに Catwalk-ROMIO では表 1 に示す 5 つの関数が定義されているだけである．

ROMIO には共有ファイルアクセスにおける一貫性に対し，1) ADIOI_CATWALK_Flush() 関数を呼ぶ以前に発行されたファイルへの書込は，以後の読込みに反映される，2) atomicity モードでの書込が完了した後の読込には，何もしないでも書込んだ内容が反映されている必要がある，という 2 種類が提供されている．前者は緩い一貫性を保証し，後者は逐次 IO と同様の強い一貫性を保証している．本稿では，atomicity モードは逐次プログラムからの移行を容易とすることのみを目的とし，性能を追求することが目的ではないと考え，本稿の範囲外とした．また ROMIO には shared (file) pointer というグローバルに共有され，かつ読込や書込で自動的に更新されるファイルポインターがあるが，これについても本稿の範囲外とする．

表 1 Catwalk-ROMIO で定義された関数

ADIOI_CATWALK_Open	ファイルの open 処理
ADIOI_CATWALK_WriteContig	ファイルの連続書込
ADIOI_CATWALK_Close	ファイルの close 処理
ADIOI_CATWALK_Flush	ファイルの同期
ADIOI_CATWALK_Resize	ファイルの truncate 処理

4.2 分散プロセス構造

Catwalk-ROMI は Catwalk 同様，図 2 に示すようなリング状の分散プロセス構造を持つ．ステージングの元となるファイルサーバおよび各ノード上にそれぞれひとつの Catwalk プロセスが生成される．隣接するプロセスは TCP/IP により通信が可能になっている．これらのプロセスは ADIOI_CATWALK_Open() 関数により MPI-IO のファイルが open された時点で生成され，ADIOI_CATWALK_Close() 関数により消滅する．以下の説明で，ファイルサーバ上の Catwalk プロセスを「(Catwalk) サーバプロセス」．ノード上の Catwalk プロセスを「(Catwalk) クライアントプロセス」と呼ぶ．

4.3 ファイルの書込

Catwalk-ROMIO の書込 open 処理の概略を図 3 に示す．

(1) ユーザプロセスが ADIOI_CATWALK_Open() 関数により書込 open を要求した場合，図 2 にあるような Catwalk-ROMIO の分散プロセス構造が生成される．引き続きジャーナル用のファイルが生成され，以後，ユーザプロセスの書込は，ファイルポインタ，データ長，データから成るジャーナルレコードとして記録される．なお，ジャーナルファイルはユーザプロセスの数だけ生成される．以後，ADIOD_CATWALK_WriteContig() 関数は，このジャーナルファイルへのジャーナルレコードの書込をおこなう．

(2) ADIOI_CATWALK_Close() が呼ばれると，Catwalk クライアントプロセスに flush 要求として通知される．

(3) Flush 要求はリングに沿って同期され，最終的にサーバプロセスに届く．

(4) サーバプロセスは，リングの最初のクライアントプロセスに対し，ジャーナルレコー

ドを送出するよう要求する．最初のクライアントプロセスは自ノードのジャーナルレコードをソートし，隣のクライアントに渡す．以後，クライアントプロセスでは，前のプロセスから受け取ったジャーナルレコードと，自ノードのジャーナルレコードを比較しながら，一番小さいファイルオフセットのレコードから順次隣に送る．

(5) サーバプロセスは，ソートされて送られてきたジャーナルレコードの内容に従い，順次ファイルに書込む．全てのジャーナルレコードを受け取ったらリングに沿って終了した旨，各クライアントプロセスに通知する．終了を受け取ったクライアントプロセスはジャーナルファイルを delete する．

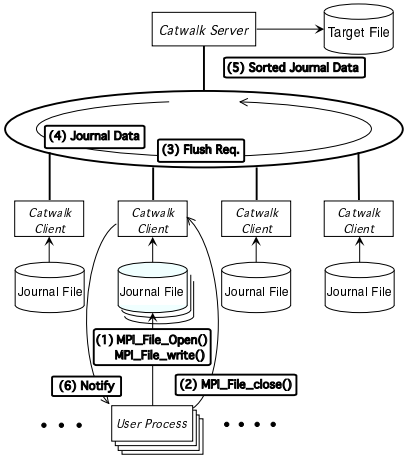

図 3 書込の手順

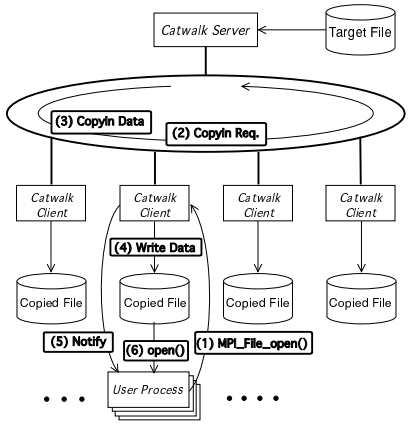

図 4 読込の手順

この手順 4 の処理を分かり易いように図示したものが図 5 である．前隣のノードから送られて来たジャーナルレコードと，自ノードの各ユーザプロセスにより生成されたジャーナルファイルのレコードの中から，最も小さいファイルオフセットを持つものを後隣のノードに送出する．ここではジャーナルファイルのその時点でのオフセットにあるレコードのみを比較し，ジャーナルファイル中の全てのレコードをソートすることは処理時間の高速化のため，おこなわないものとする．ここで，もし各ユザープロセスが生成したジャーナルレコードのオフセットが昇順になっていた場合，最終的に Catwalk サーバが受け取るジャーナルレコードも昇順に並ぶことになる．これは基本的にマージソートと同じ処理である．これにより，ジャーナルレコードをオリジナルのファイルに反映させる際，無駄な seek を減らすことが可能となる．ここでもし，ユーザプロセスがジャーナルレコードを昇順で，かつ，書込む範囲が他のプロセスと完全に排他的であり，間隙がない場合には，seek 操作を全く伴わないで逐次書込みが可能となる．また，ローカルなジャーナルレコードが完全にソートされていない場合でも，ソートすることで seek の回数や移動する距離が減るものと期待できる．

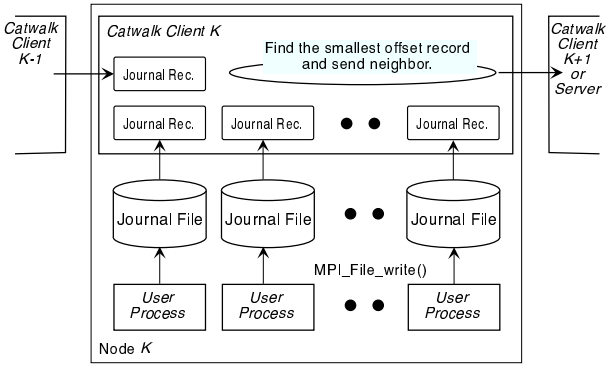

図 5 ジャーナルレコードのソート

4.4 ファイルの読込

Catwalk-ROMIO の読込処理の概略を図 4 に示す．この処理は，open 時にリングプロセス構造を作るという点を除いて，Catwalk[10] と同じである．

(1) ユーザプロセスが `ADIOI_CATWALK_Open()` 関数により読込 open を要求した場合，図 2 にあるような Catwalk-ROMIO の分散プロセス構造が生成される．

(2) ファイルの読込 open の要求がリングに沿って同期され，最終的にサーバノードに到達する．

(3) サーバノードでは，要求のあったファイルを open し，その内容をリングに沿って転送する．

(4) クライアントプロセスは受け取ったデータを次のクライアントに渡すと同時にローカルディスクのファイルに書込む．

(5) 全てのデータを受け取ったクライアントプロセスは，ローカルディスクのファイルを close し，終了をユーザプロセスに通知する．

(6) ファイル転送の完了を受け取ったユーザプロセスは，新たに生成されたローカルディスクのファイルを open し， `ADIOI_CATWALK_Open()` 関数が終了する．以後，このファイルに対する読込みは全てローカルディスクにコピーしたファイルへのアクセスとなる．

Catwalk-ROMIO をリング状の分散プロセス構造としたのは，1) ファイルのブロードキャストをパイプライン処理し，バンド幅を稼ぐ，2) 逐次的な処理のためプログラミングが比較的容易，という理由からである[10]．このようなパイプライン処理では計算ノード数に比例したパイプライン段数となり，ファイルサーバからのデータ出力バンド幅が一定にも関わらず，理論上はファイルが十分に大きければ計算ノード数に比例したバンド幅が得られるという利点がある．また，バッファキャッシュの flush による影響を避けるため，計算ノード上で書込むファイルを `O_DIRECT` フラグで open している点も同じである．

Catwalk-ROMIO におけるファイルの並列書込では，write 操作はローカルディスクへの書込のみが発生し，close 時にローカルなジャーナルファイルの内容が全てサーバノードに集められる．読込時には，open 時に対象とするファイルを全てのノードにコピーし，read 時にそのローカルなコピーからファイルが読込まれる．どちらの場合も write および read はローカルなディスクへのアクセスだけであり，書込時は close の時に，読込時は open の時に，リングに沿ってサーバに対して IO が発生する．

4.5 ファイルの同期

`ADIOI_CATWALK_Flush()` で定義されるファイルの同期は，基本的には上記の書込 close 処理と読込 open 処理を連続しておこなうのと同じである．ただし，対象となるファイルは既に open 済みであるため，Catwalk の分散リングを生成する必要はない．

4.6 ファイルサイズの変更

`ADIOI_CATWALK_Resize()` 関数（Linux における `truncate()` と同じ処理）が呼ばれた場合は，各ノードにおいてジャーナルファイルの各レコードを調べ，切り詰めたい長さよりレコードのファイルオフセットが大きい場合にはそのレコードを廃棄し，レコードの途中で切り詰める必要がある場合は，そのレコードを切り詰める処理となる．

表 2 Spec. of T2k (Tokyo) Node

CPU	AMD Barcelona, 2.3 GHz, 4 Cores, 4 Sockets
Memory	32 GB
Local Disk	SATA
Ethernet	Intel E1000 (1 Gbps) ×2
	Myrinet 10G ×2
OS	RHELS 5.1
File System	EXT3
	NFS3 (`async` export option)
	Bonnie++, Seq. Block Input: 49.52 MB/sec
	Bonnie++, Seq. Block Output: 39.76 MB/sec

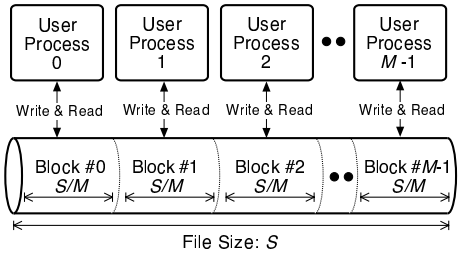

図 6 評価に用いたファイルアクセスパターン

5. 予備評価

T2K オープンスパコン（東大）[6] の 17 ノード（ファイルサーバ 1 台と計算ノード 16 台）を用いた．計算ノードの概要を表 2 に示す．ファイルサーバはもともと計算ノードであり，計算ノードとも同じスペックである．本クラスタには 1 Gbps の Ethernet と並列計算用の Myrinet 10G の 2 種類のネットワークがあるが，今回の計測では 1 本の Ethernet のみが MPI-IO に関わるネットワークになっている．表 2 には Bonnie++[5] で計測したファイディスクの性能も併せて示した．

評価に用いたプログラムは，決められた大きさのファイルを rank 数 M で割った大きさのブロックを，各プロセスが seek してから読み書きする（図 6）．

図 7 は書込性能のスケーラビリティを検証するために，MPI-IO の `MPI_File_write()` のバンド幅を計測し，横軸にノード数，縦軸に総バンド幅をノード数で割った値をプロットしたものである．`MPI_File_write()` は 1 回の呼出で，与えられたブロックを全て書込む

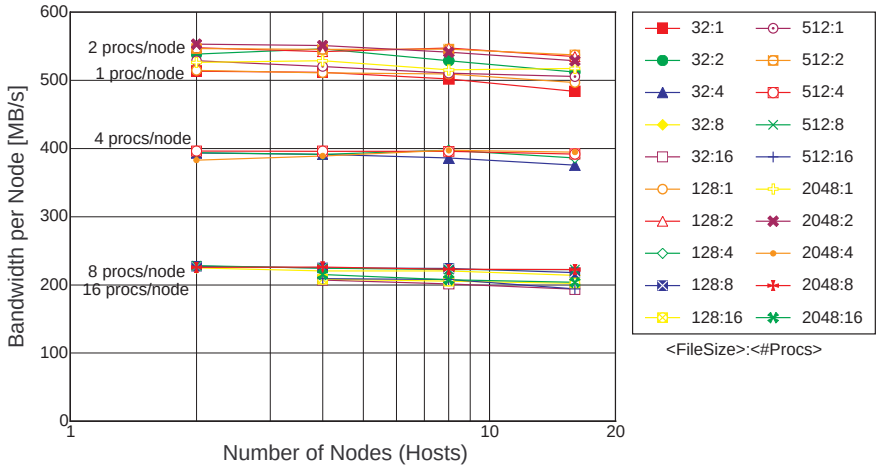

図 7 書込性能のスケーラビリティ

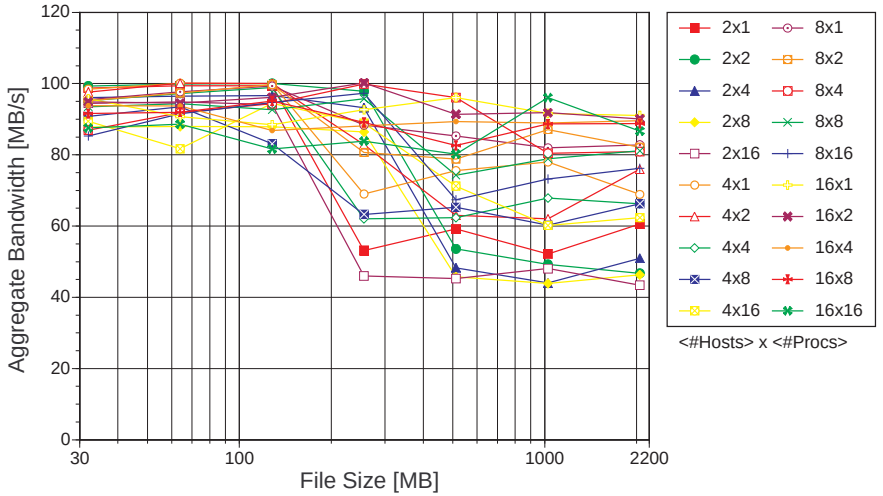

図 8 書込バンド幅

ものとする．図中の凡例で，”S:P” とあるのは，サイズ S（単位は Mbyte）のファイルをノードあたり P プロセスで書込んだことを示している．基本的にローカルな処理であるため，ノード数に比例したバンド幅を得ることができた．アクセスするファイルの大きさがメモリよりも十分に小さいため，バッファキャッシュへの書込となっており，実際のディスクより高いバンド幅が計測されている．またこの図から，ノード上のプロセス数によりバンド幅が異なっている点である．各ノードから出力されるブロック長の総和は，プロセス数に依らず一定であるため，ノード単位での書込バンド幅はプロセス数に依存することを図 7 は示している．プロセス数が 1 または 2 の時はほとんど同じバンド幅を示し，以後，プロセス数が増えるにつれバンド幅が減少し，プロセス数が 8 または 16 の時にノードあたり，およそ 200 MB/s のバンド幅を得ることができた．

図 8 は，`MPI_File_close()` 処理の時間まで含めた時にバンド幅を，ファイルの大きさを横軸に，総バンド幅を縦軸にプロットしたものである．図中の凡例で “NxP” とあるのは，ノード数 N でそれぞれのノードでプロセス数 P を起動したことを意味する．図 7 と異なり，ファイルサーバのファイルに対し，ジャーナルの内容を反映させている時間も含まれている．この場合もファイルの大きさが最大で 2GB とメモリよりも十分小さいため，実際の書込の大半はバッファキャッシュへの書込となっているものと予想される．しかしながら，図 7 で得られたバンド幅よりも小さい値であり，最大でおよそ 100 MB/s になっている．これは用いているネットワークのバンド幅（125 MB/s）がボトルネックとなっているためと推測される．また図 8 からは，大きいファイルサイズでかつノード数が少ない場合（特に 2 ノード）に性能低下が顕著であるが，この理由はいまのところ不明である．

図 9 は `MPI_File_read()` 処理のバンド幅を計測した結果を，図 7 と同様の方式でプロットした図である（凡例の読み方も同じ）．また図 7 の場合と同様，読込むべき大きさのブロックを 1 回の /tt MPI_File_read() で読込んでいる．同じ縦軸，横軸の取り方であるが，グラフの様子は図 7 と大きく異なっている．これは open 時に各ノードにファイルサーバにあるファイルを転送する際，各ノードにおける `open()` が `O_DIRECT`，つまりバッファキャッシュをバイパスしているため，このファイルを read する際，ディスクからの読込が発生していることが関係しているものと推測される．図 9 の中で，太い線でプロットされているものは，ひとつのプロセスが読込む大きさが 1 MB 以下とと比較的小さく，かつ，ノード内のプロセス数が 4 以上の場合を示しており，この条件に合致するケースでバンド幅が大きく低下していることが分かる．これは，小さいファイルを同時に読込むことを意味し，ディスクに対しランダムアクセスに近い状況となり，バンド幅が低下したものと推測される．また，ノード数が増える程，バンド幅が低下している傾向がみられるが，これも同じ理由と考えられる．

Vol.2009-HPC-121 No.14
2009/8/4

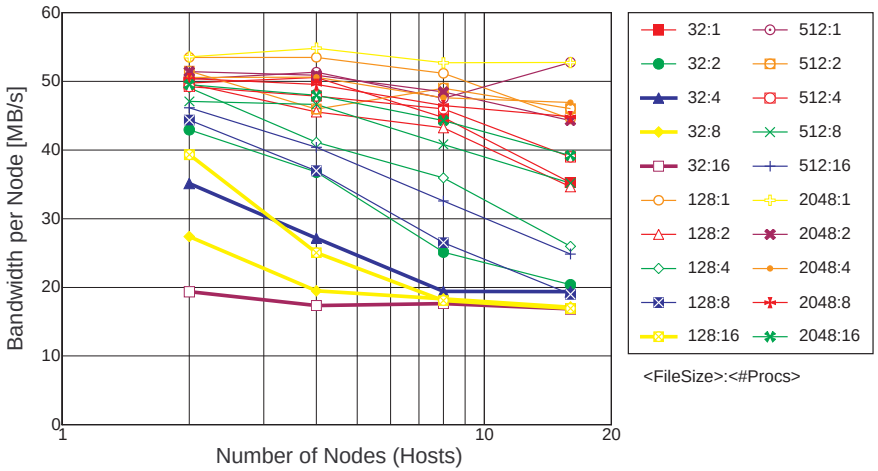

図 9 読込性能のスケーラビリティ

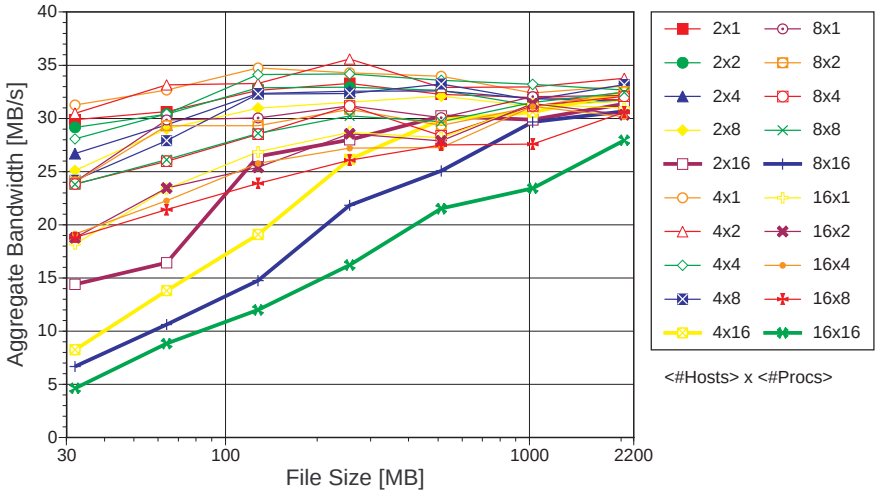

図 10 読込バンド幅

図 10 は時間の計測において `MPI_File_read()` に加え `MPI_File_open()` の時間を含めたときのバンド幅をプロットしたものである．`MPI_File_open()` 時には，リング状の分散プロセス構造を生成していることがオーバヘッドとなり，特に小さなファイルの読込時に，またノード内のプロセス数が 16 の時に速度の低下が顕著となって現れている．一方，今回の計測における最大の 2 GB のファイルの読込み時には，およそ 30MB/s のバンド幅が得られている．この値は[10] で報告されている値の 20 MB/s よりも大きいが，各ノード上でのローカルディスクへの書込処理が，マルチスレッド化とバッファサイズの調整等により改善された結果である．

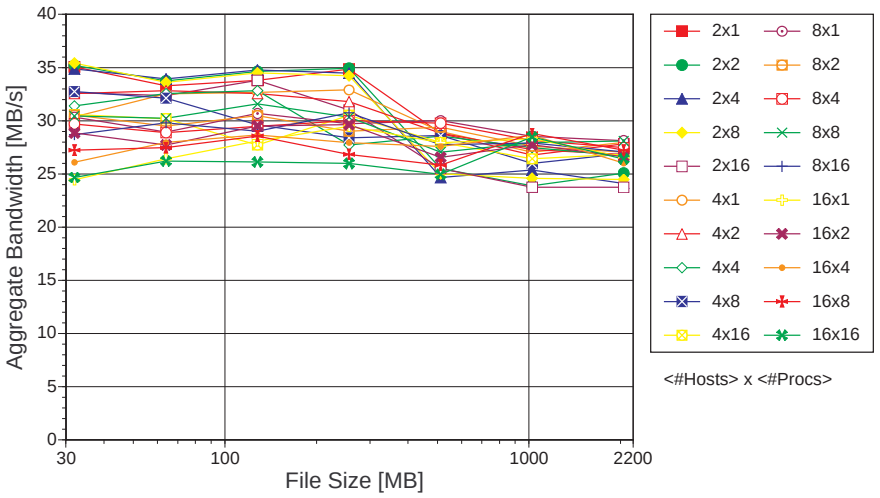

図 11 Sync のバンド幅

図 11 は，ファイルを `MPI_File_write()` が書込んだ後に，`MPI_File_flush()` を呼び，その時間を計測し，書込んだファイルの大きさと時間から，バンド幅としてプロットしたものである．この時間は，書込み後に `MPI_File_close()` を呼び，再びそのファイルを `MPI_File_open()` で open した時間から，リング生成時間を除いたものにほぼ等しいはずである．図 8 と図 10 から読込がボトルネックになるものと考えられる．図 11 の 30 MB/s 弱という値は，ファイルサイズに依らずほぼ一定であり，先の読込バンド幅の値の最高値とほぼ合致している．

6. 考　　察

ジャーナルファイルについて

図 7 では，ノード内の複数のプロセスが同時に複数のジャーナルファイルに書込む際の性能低下が示された．ジャーナルレコードを書込む際，Catwalk クライアントプロセスを経由させた場合を想定してみると，ユーザプロセスと Catwalk クライアントプロセスとの間で生じるプロセス間通信のオーバヘッドが生じるものの，同時書込による性能の劣化が回避できる．しかし，たとえユーザプロセスが連続に書込んだとしても，一本化されたジャーナルレコードの並びが連続しているとは限らないため，最終的にサーバでの書込速度が seek のオーバヘッドにより低下する可能性がある．

今回の Catwalk-ROMIO の実装では，close 処理において，サーバ側で `sync()` を呼んではいないものの，ノード上のジャーナルレコードは全てサーバに送られたことを保証している．NFS を含む多くの分散あるいは並列ファイルシステムで，close 処理を実際には非同期的に処理し，見掛けの処理速度を向上させる方法は広く用いられている．Catwalk-ROMIO においても close 処理の終了をユーザプロセスに通知した後に，背景処理としてファイルーサーバに転送することは可能であり，これにより見掛けのバンド幅を大幅に向上させることが可能である．しかしながら，Catwalk-ROMIO はファイルを `MPI_File_open()` から `MPI_File_close()` の間までしか管理できないため，`MPI_File_close()` を呼んだ後に，同じファイルに対して `MPI_File_open()` が呼ばれた時点で，先の close 処理が実際に完了していることを保証する機構が別途必要になる．

Collective IO について

Catwalk-ROMIO では，`MPI_File_open()` 時に分散リング構造を生成するためのオーバヘッドが無視できないことが，図 10 で示された．Catwalk では並列ジョブの起動時にリングを生成するため，プログラムの実行時にはそのオーバヘッドは見えてこない．しかしながら Catwalk-ROMIO では，collective な操作があることから，MPI-IO を open する MPI の Communicator と同じノード集合で Catwalk のリング構造を生成する必要がある．このオーバヘッドには，Catwalk クライアントプロセスを fork&exec するコストも含まれるため，最初に `MPI_File_open()` が呼ばれたとき，あるいはデーモンプロセスとしてクライアントプロセスを事前に立ち上げておく，などの方法でリング生成のオーバヘッドを低減することが可能であると考える．

7. 関 連 研 究

並列ファイルシステムをキャッシュを用いて高速化する技法はいくつか提案されている[9]．一方，論文[3] ではファイル書込において書込順序で書込要求をソートして書込性能の向上を目的とする研究である．本稿で提案する Catwalk=ROMIO のように，ジャーナル形式で書込のトランザクションをいったん保存し，ファイルポインタでソートし書込性能を向上させようとする研究は後者に近い，Catwalk-ROMIO の特徴は，1) ローカルディスクを用いたこと，2) リングトポロジーに沿ってジャーナルレコードをソートする 2 点であろう．

おわりに

オンデマンドファイルステージングシステムである Catwalk を，MPICH2 の MPI-IO 実装である ROMIO の ADIO デバイスとして拡張した．Catwalk-ROMIO についての実装方式と予備評価の結果を示した．Cwatwalk=ROMIO では，ユーザプロセス毎に書込トランザクションをジャーナルし，close 処理でサーバにある目的とするファイルにトランザクションを適用することで，Catwalk では不可能だった書込ファイルの共有問題を解決した．また予備評価から，ノード数に比例した総バンド幅を実現できた．

一方で Catwalk-ROMIO には，まだいくつかの性能向上の余地があることが判明したが，それらを解決するであろう方式案も提示した．本稿における予備評価からは Catwalk-ROMIO の実装方式が有効であることが示されたものと考える．Catwalk および Catwalk-ROMIO は単一のファイルサーバを仮定しているものの，本稿で提案されたジャーナリングを用いた並列ファイルアクセスの高速化は，複数ファイルサーバの場合にも適用可能と考える．

今後は，並列ファイルシステムのベンチマークプログラムを用いた評価をベースに，Catwalk-ROMIO の改良と他の並列ファイルシステムとの比較評価などに取り組む予定である．Catwalk-ROMIO は SCore[4] の次バージョンに組込まれ，オープンソースとして配布される予定である．

謝　　辞

本研究の一部は文部科学省「e-サイエンス実現のためのシステム統合・連携ソフトウェアの研究開発」からの支援を受けている．

Vol.2009-HPC-121 No.14
2009/8/4

参 考 文 献

1) Carns, P.H., III, W. B.L., Ross, R.B. and Thakur, R.: PVFS: A parallel file system for linux clusters, *In Proceedings of the 4th Annual Linux Showcase and Conference*, USENIX Association, pp.317–327 (2000).
2) Hori, A., Kamoshida, Y., Matsuba, H., Ohta, K., Yasui, T., Sumimoto, S. and Ishikawa, Y.: On-Demand File Staging System for Linux Clusters, *Proceedings of IEEE Cluster Conference* (2009). (to appear).
3) Ohta, K., Matsuba, H. and Ishikawa, Y.: Improving Parallel Write by Node-Level Request Scheduling, *Proceedings of IEEE CCGrid2009* (2009).
4) PC Cluster Consortium: SCore. http://www.pccluster.org/.
5) Russell Coker: Bonnie++. http://www.coker.com.au/bonnie++/.
6) T2K Open Supercomputer Alliance: T2K. http://www.open-supercomputer.org/.
7) Thakur, R., Lusk, E. and Gropp, W.: Users Guide for ROMIO: A High-Performance, Portable MPI-IO Implementation, Technical Memorandum ANL/MCS-TM-234, Argonne National Laboratory (2004).
8) 建部修見，森田洋平，　松岡聡，関口智嗣，曽田哲之：ペタバイトスケールデータインテンシブコンピューティングのための Grid Datafarm アーキテクチャ，情報処理学会論文誌：ハイパフォーマンスコンピューティングシステム，No.Sig 6 (HPS 5), pp. 184–195 (2002).
9) 太田一樹，　石川裕：ファイルサーバー独立な並列ファイルキャッシュ機構，研究報告「ハイパフォーマンスコンピューティング（HPC）」No.2009-HPC-119，情報処理学会，pp.43–48 (2009).
10) 堀敦史，鴨志田良和，松葉浩也，　安井隆，住元真司，　石川裕：ファイルステージング再考：オンデマンド化と高速化に向けたプロトタイプ実装の評価，研究報告「ハイパフォーマンスコンピューティング（HPC）」No.2009-HPC-119，情報処理学会，pp. 38–42 (2009).