クラスタシステムにおける高速ファイル共有のための一手法

大谷敦久[1]　青野寛[1]　外丸浩子[1]　佐竹康司[2]

概　要

本論では、Linux 等を用いたクラスタシステム、及び SAN(Storage Area Network)環境において、高速にディスク／ファイルを共有するための手法を提案する。我々が過去に開発した手法では、サードパーティー転送の機能を有するディスク装置を用いることで、ローカル I/O 性能に近い性能を出し得ることを実証した。しかし、サードパーティー転送のような機能を有しない一般に入手可能なディスク装置であっても適用できる一般性のある方法が望ましいと考えられる。

そこで、カーネルデーモンを使用し、I/O に関する制御情報のみをサーバー／クライアント間において、ネットワーク経由で送受信するというデーモン方式により、ディスク装置の特定の機能に依存せず、且つストライピングにおいてもスケーラビリティーを維持できる高速ファイル共有の手法を開発した。そして、本手法をカーネル 2.4.17 にインプリメントして性能の評価を実施したところ、ローカル I/O 性能の 90%程度の性能が得られることを確認した。また、ストライピングした場合でも、ほぼストライピング数に対してスケールすることがわかった。

１．はじめに

近年のディスク装置の容量の向上は、目を見張るものがある。 一方で、HPC をはじめ、エンタープライズ分野においても動画等の巨大なファイルを扱うことが一般化しており、複数ノードから高速にファイルを共有する必要性が高まっている。Linux で利用できる一般的なファイル共有の手段としては NFS があるが、ネットワーク上を介してデータ転送を行うため、ディスク装置の持つ本来の性能を発揮することができない。このため、サーバーを経由せず、クライアントとディスク装置間で直接データ転送を行うための SAN(Storage Area Network)環境を対象とした、高速ファイル共有のための種々の方法の研究開発が行われている([1]～[5])。

我々が以前に HIPPI のディスクアレイ装置を用いて検証した手法[1][2]では、サードパーティー転送の機能を有するディスク装置を用いることで、クライアントとディスク装置間の直接データ転送を行い、ローカル I/O 性能に匹敵する性能で、ファイル共有機能を実現できることを実証した(同手法を以後 TPT 方式とする)。しかし、この方法は、ディスク装置にサードパーティー転送の機能がサポートされていることが前提になっており、どのようなディスク装置でも使用できる方法ではなかった。ユーザ自身が、価格や性能等の面から好みのディスク装置を選択し、その装置を用いてファイル共有を実現できるような手法の方が、より有用性が高いと言える。また、文献[3][4][5]は、完成度の高い製品であると言えるが、キャッシュ制御やロック制御の仕組みが複雑で、複数クライアントから同一ファイルへのアクセスがあった場合の処理のオーバーヘッドが多少増加する可能性が考えられる。

一方、近年 HPC 関連分野では、SAN におけるファイル共有を前提としたシステム構成におい

[1] 日本電気(株)
[2] 東北日本電気ソフトウェア(株)

ても、単体のディスク装置の性能では不足で、ストライピングによる並列 I/O により、I/O 性能を向上させることが一般化している。このため、ファイル共有機能においてストライピングでの本来の並列 I/O 性能が損なわれることがないように特に留意する必要がある。

そこで、本開発では、1)特定のディスク装置のみがサポートする特殊な機能の使用を前提とせず、Fibre Channel 等による SAN 環境に接続できるディスク装置であれば、どのような装置であっても適用でき、2)少ない工数で開発できるように比較的簡単な構造とし、3)さらに対象ファイルシステムが複数のディスク装置でストライピングされている場合でも、並列 I/O のスケーラビリティーを維持できるように､効率よくデータ転送ができる手法を開発することを目的とする。

そして、開発した手法をカーネル 2.4.17 に対してインプリメントし、その性能について検証する。

なお、本論で述べる手法により実現できるクラスタファイルシステムを GFS(Global File System)と呼称する。

２．対象とするハードウェア構成

図１に、対象とする一般的なハードウェア構成の例を示す。複数ノードからなるクラスタシステムにおいて、各ノードは Fibre Channel 等のスイッチを介して、ディスク装置へ最低１つ以上のパスを持っており、同時に各ノードはネットワーク回線でも接続されている。また、NFS と同様に、対象のファイルシステムをローカルにマウントするノードをサーバーとし、それ以外のノードをクライアントとする。

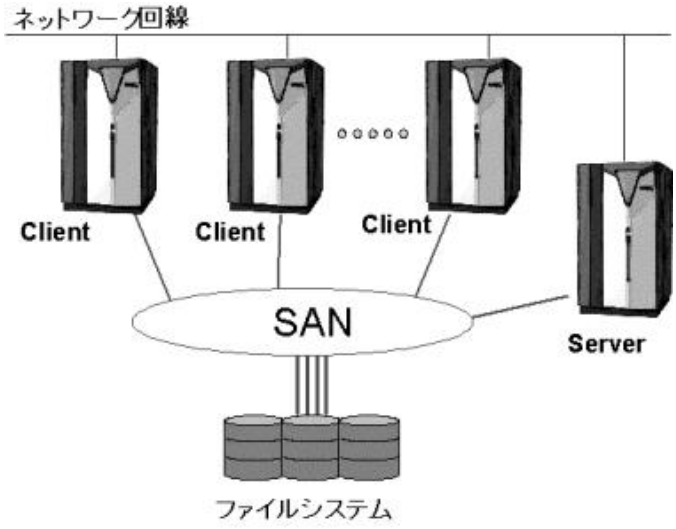

図１：対象とするハードウェア構成

３．高速ファイル共有の基本的な考え方

図２に、NFS の場合のデータ転送経路を示す。NFS では、ネットワークを介してクライアントからサーバー、さらにサーバーからディスク装置と、２度データ転送を行う必要がある上に、１回で転送できるデータサイズが小さいので、ディスク装置が本来持っている性能を出すことはできない。

これに対し､図３に示したように SAN におけるファイル共有の一般的な考え方としては､データの転送自体はサーバーを介さず、クライアントとディスク装置の間で直接データ転送を行うことにより、ディスク装置の性能を損なわないようにすることである。ただし、単にクライアントから I/O を発行するだけでは､ファイルシステムとしての整合性が保持できなくなる｡このため、サーバー／クライアント間で、ファイルシステムの整合性を保つための手段が必要となる。

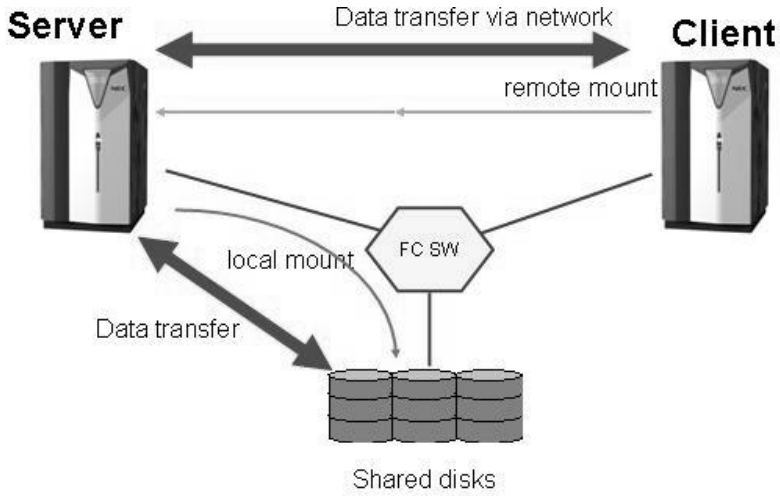

図２：NFS によるデータ転送の経路

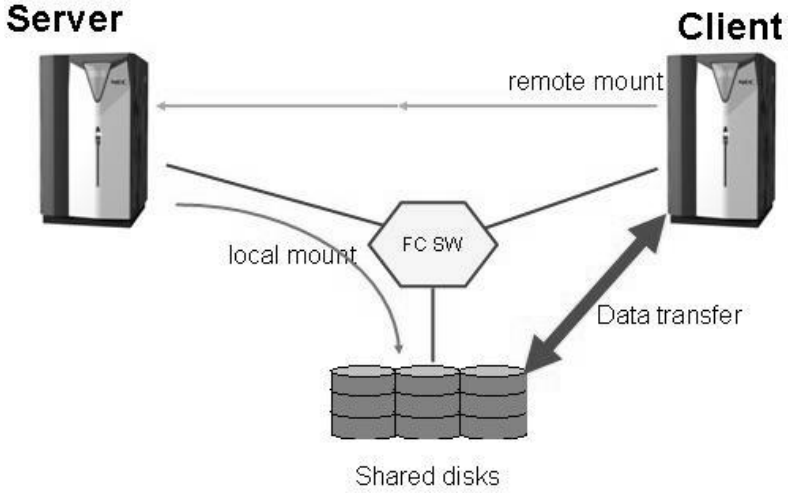

図３：高速ファイル共有のためのデータ転送の経路

４．デーモン方式

サーバー、クライアントの両方から I/O が発行された場合でも、そのファイルやファイルシステムに矛盾を生じさせないようにするため、本手法ではクライアントから要求された I/O も、サーバーのファイルシステムを経由させるという方法をとる。このため、I/O に関する制御情報をサーバー、クライアント間で通信によってやり取りするための４種のデーモン gfsd1, gfsd2, gfsd3, gfsd4 を用いることとする(以後、デーモン方式)。

４．１　処理概要

図４に、本手法の概略図を示す。実線の矢印は、本手法の処理の流れを示しており、濃いグレーと薄いグレーの四角は、新規または変更を加えたカーネル内のコンポーネントを示している。

以下、順を追って説明する。

(1)ユーザプロセス(I/O 発行プロセス)から、システムコール経由で I/O が要求される。この際、一部 NFS の機能を利用し、ファイルハンドラ、I/O サイズ、ファイルオフセット、認証情報等の「制御情報１」を含む制御パケットをサーバーの gfsd1 に送信する。

(2)サーバーのファイルシステムは、gfsd1 から I/O 要求を受け取ると「制御情報２」を作成する。この際、対象ファイルの i-node ロックを取得し、キャッシュをクリアする。その後、サーバードライバを経由して、制御情報２を gfsd2 に渡し、gfsd1 は gfsd4 から I/O 終了通知を受け取

るまで待ち合わせる。なお、制御情報２には、対象ディスク装置のスペシャルファイル名、ディスクの物理的なアドレスとブロック数、I/O 開始メモリアドレスからのオフセット等が含まれる。

(3)gfsd2 は、上記制御情報２を含む制御パケットをクライアントの I/O 発行プロセスに送信する。

(4)I/O 発行プロセスは、クライアント機能において、受信した制御情報２を用いてファイルシステムを介さず直接 SCSI ディスクドライバを呼び出すことにより、対象のディスク装置へ I/O を発行する。

(5)I/O の終了は、割り込みで通知される。

(6)gfsd3 に当該 I/O が終了したことを通知する。

(7)gfsd3 は、gfsd4 に対して、当該 I/O の終了を通知するための「制御情報３」を含む制御パケットを送信する。制御情報３にはエラー情報も含まれる。

(8)gfsd4 は、I/O の終了を待ち合わせている gfsd1 に当該 I/O の終了を通知する。この際、ファイルシステム内で i-node ロックを解除する。

(9)gfsd1 は、クライアントの I/O 発行プロセスに対して、サーバーでの I/O 処理の全てが終了したことを通知する「制御情報４」を含む制御パケットを送信する。制御情報３にエラー情報が含まれていた場合は、これを制御情報４に格納し、クライアント側でエラーを認識できるようにする。そして、クライアント機能においてこの通知を受信した I/O 発行プロセスは、ユーザモードに復帰する。

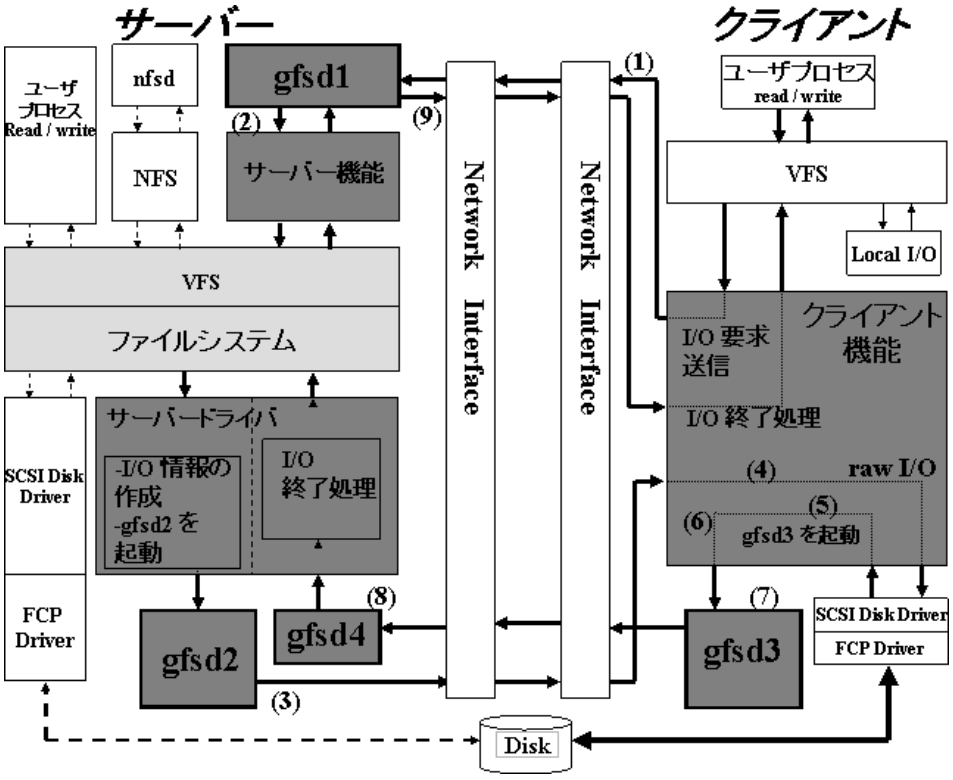

図４：本手法の概略図

以上説明したように、デーモン方式は、TPT 方式のようなディスク装置の特殊な機能を使用せずに、クライアントとディスク装置間の直接データ転送によるファイル共有を実現できる。

４．２　パケットの転送回数と I/O 回数の低減について

I/O 対象のファイルが存在するファイルシステムが、複数台のディスク装置によってストライピングされている場合でも、本手法は適用可能である。しかし、ストライピングサイズの単位毎に図４の(3)～(8)の処理を行うと、サーバー、クライアント間で送受信される制御パケットの数が

膨大になり、ネットワークのトラフィックを増大させる原因になることが考えられる。これは、ストライピングしていない場合は、物理的に連続したブロック毎に制御パケットをサーバーからクライアントへ送ることができるが、ストライピングした場合は、ストライピングサイズ毎に異なるディスク装置へ I/O を発行するので、I/O 発行の単位が細分化されてしまうためである。

図５に、ストライピングサイズの単位毎に番号付けした論理的なファイルのイメージと、これに対応するディスク上の物理的な配置の例を示す。２ストライピングの場合、Disk 1 と Disk 2 へ交互にストライピングサイズ毎の領域が確保されるが、例えば図５の Disk 1 の領域１と３のように、ファイルのイメージとしては離れているが、ディスク上の物理的な位置は連続しているという場合も少なくないと考えられる。このような連続領域については、クライアントからディスク装置へ I/O を発行する際に領域１と３を１つの SCSI コマンドとしてまとめて発行することが可能であるため、サーバーからクライアントへ制御パケット送る際、１と３を１つにまとめて送ることにより、制御パケットの送受信の回数とクライアントでの I/O の回数の低減が可能となる。

さらに、ディスク上の物理的な位置が連続していない場合は、クライアントから発行する I/O 自体はまとめることはできないが、不連続領域であっても制御パケットを１つにまとめることは可能である。つまり、図５の Disk１を例に取ると、領域１，３，５，７に関する４つの制御情報２を１つの制御パケットにまとめ、クライアントにおいて領域１，３、及び領域５，７の２回の I/O 発行により処理できることになる。

上記のように制御パケットをまとめるためには、図４内のサーバードライバが上位層のファイルシステムから制御情報２をまとめて取得できる必要がある。このため、ファイルシステム層では I/O 対象のファイルが使用しているディスクブロックのマップを取得し、各領域の制御情報２をまとめて作成する。

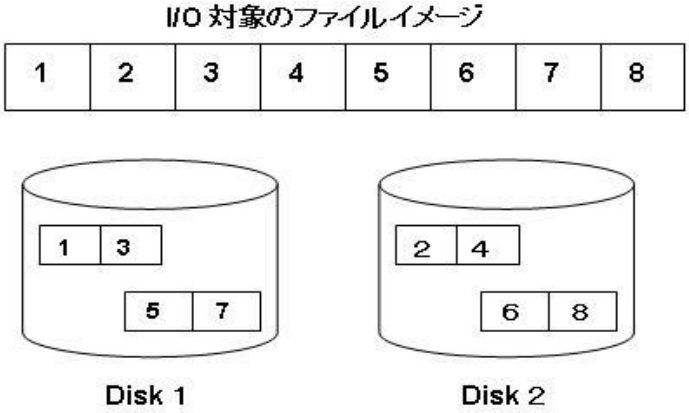

図５：ストライピング時のファイルイメージと
そのディスク上の配置の例。

４．３　クライアントにおける I/O 処理の効率化

図５の Disk １と Disk ２のように、異なるディスク装置に対する制御情報２の送信に関しては、サーバー内のサーバードライバがディスク装置毎に処理を行う関係上、制御パケットを１つにまとめることは難しい。しかし、各装置への I/O を連続的に発行することで、ストライピングとしての並列性能を損なわないようにできる。

具体的には、図４の(3)～(8)の処理は、ディスク装置毎に複数回行う必要がある。この際、実際にディスク装置に対して(4)で I/O を発行する I/O 発行プロセスは、発行した I/O の終了を待ち合わせないので、(3)で gfsd2 から送られて来る制御パケットに即時に対応することができ、受信した I/O 要求を連続的にディスク装置に対して発行することが可能である。また、I/O 終了後の処理(5)～(8)は、I/O 発行の処理(3)～(4)とは、全く独立して行われるため、待ちが発生して次の I/O の発行が遅延されるということがない。

４．４　クライアントのキャッシュ

クライアントにおいてはキャッシュを持たないこととする。クライアントのキャッシュは、比較的小さな I/O サイズで且つ同一クライアントから同じファイルに対して、続けて I/O が発行される場合には効果があるが、I/O サイズが大きい場合にはあまり効果がなく、また複数のクライアントから同じファイルへ I/O が発行されるような場合は、かえって処理が複雑になり、効率が悪くなることも考えられる。さらに、下記４．５節の通り、I/O サイズが小さい場合は、NFS に処理を切り換えるため、NFS のキャッシュ機能が利用できる。このため、I/O サイズが小さい場合の性能低下を避けることができる。

４．５　NFS の利用

前述の通り、本手法はクライアントでキャッシュを持たない。このため、I/O サイズが小さい場合は、サーバー／クライアント間での通信のオーバーヘッドが相対的に大きくなり、かえって NFS より性能が悪くなるという不都合が生じる可能性がある。このため、I/O サイズが 64K バイトに満たない場合は、図４の処理を行わず、NFS の I/O として処理する。この判別は、カーネル内で自動的に行うので、アプリケーションプログラムを変更する必要はない。

また、リモートマウント、ファイルのオープン等の I/O 処理に直接関わらない処理についても、NFS の機能を利用する。

５．適用結果

カーネル 2.4.17 をベースとして、本手法のインプリメントを行った。図４に示したように、サーバー／クライアントの各カーネル機能、及びカーネルデーモンをそれぞれカーネルモジュールとして作成した。また、サーバーのファイルシステムとして、SGI 社が公開しているオープンソースの Linux 版の XFS [6]を採用した。そして、クライアントから渡された制御情報１を処理して図４のサーバードライバへ渡せるように改造を行った。

性能測定に用いたマシンは、NEC の IA-64 マシンである Express 5800/1160Xa を２台(サーバー／クライアント各１ノード)、及びディスク装置として、Fibre Channel のアレイディスク装置(RAID5, 4+P)である。これらを Fibre Channel スイッチに接続して SAN を構築した。

５．１　NFS、ローカル I/O との比較

図６に、GFS、NFS、ローカル I/O（XFS の DIRECT I/O）のリード性能を示す。本測定では、ネットワークは Gigabit Ether を使用しているが、NFS の性能は GFS の性能の高々56%程度である。また 4M バイトの I/O サイズでは GFS の性能の 34%であり、比較にならないことがわかる。

次に、ローカル I/O との比較においては、64K バイトの I/O サイズでは、ローカル性能の 71%になっており多少低下が見られる。これは、I/O サイズが小さい場合、実 I/O の時間が短いため、相対的に制御パケットの転送によるオーバーヘッドが無視できなくなるためであると考えられる。逆に、128K バイト以上の I/O サイズでは、制御パケット転送によるオーバーヘッドはあまり性能に影響していないと考えられ、ローカル I/O の 90%以上の性能が得られている。

また、上記は TPT 方式[1]によって得られる性能とほぼ同等である。

５．２ ストライピング時のスケーラビリティー

図７に、LVM を用いたストライピングにおいて、ストライピングサイズ 64K バイト、I/O サイズ 2M バイトでの GFS のリード性能を示す。図７の理論値とは、ストライピング数１の時の GFS の性能を元に、その値の２～４倍の値を２～４ストライピングの性能として計算した値を意

味する。
４ストライピングにおいて、理論値の 88%の性能となっており、ストライピング数の増加に対する性能の低下は小さく、GFS はストライピング数に対してほぼスケールすると言える。

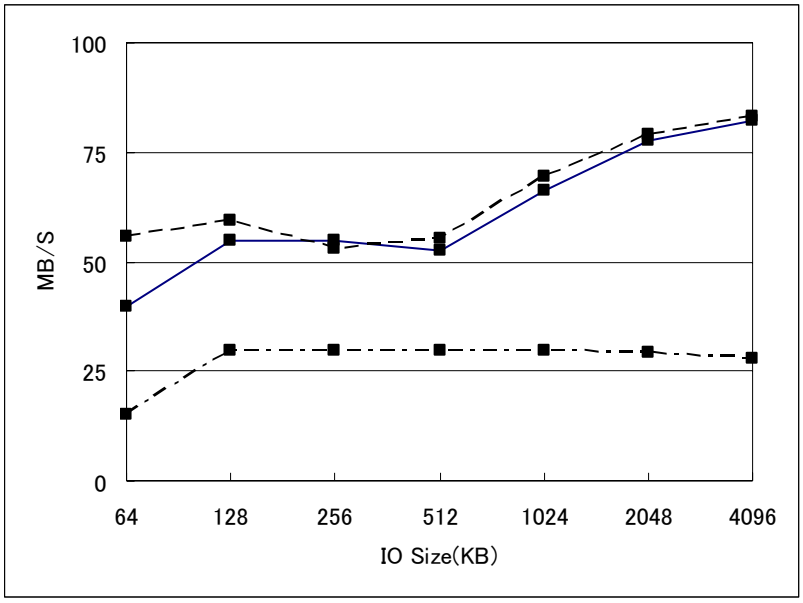

図６：GFS、ローカル I/O、NFS の性能測定結果。
実線、点線、一点鎖線は、それぞれ GFS、ローカル I/O、NFS を示す。

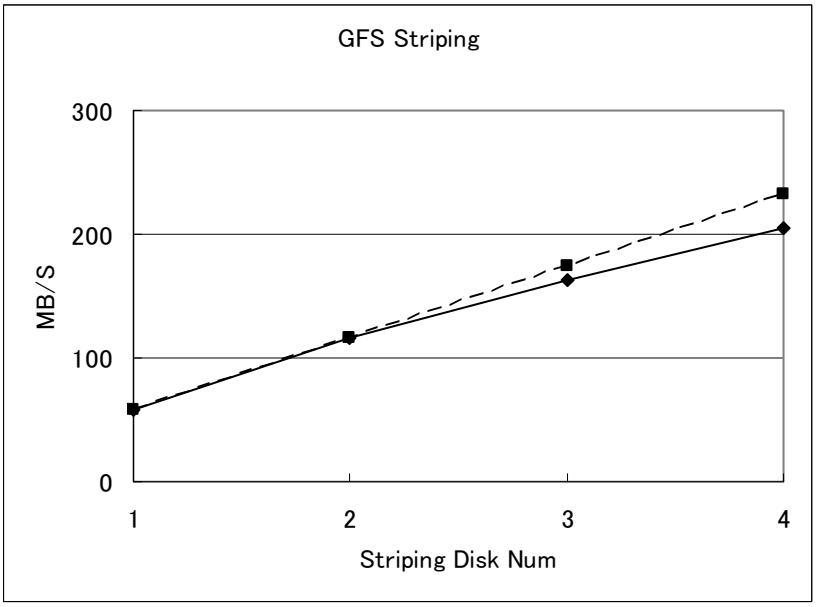

図７：ストライピング使用時における GFS の測定値と理論値の比較。
実線、点線は、それぞれ測定値、理論値を示す。

５．３ ストライピング時のパケットの転送回数

４．３節で述べたパケットの転送回数低減の効果について調べるため、ストライピングサイズ毎に別々に転送した場合(方法 A) と、パケットをまとめて転送した場合(方法 B)について、パケットの転送回数、及び I/O 発行プロセスが消費したシステムタイムの比較を行った。
図８に、4 ストライピングの場合について、各 I/O サイズ毎にリード／ライトを 1000 回繰り返した際の方法 A 及び方法 B のパケットの転送回数を示す。方法 A では、4M バイトでの転送回数

は 256K バイトの約 13 倍になっているのに対して、方法 B ではほとんど増加がないことがわかる。これより、I/O サイズが大きくなってもネットワーク負荷が増加しないことがわかる。

次に、図 9 に I/O 発行プロセスのシステムタイムの変化を示す。方法 A では、I/O サイズが倍になる毎に、システムタイムもほぼ倍になっており、また 256K バイトと 4M バイトでの値を比較するとその差は約 20 倍である。これは、パケット受信回数と I/O 発行回数の増大に伴うオーバーヘッドが大きく反映されていると考えられる。

これに対し、方法 B では 256K バイトと 4M バイトを比較してもわずかに 2 倍程度である。I/O サイズの増加に伴って、I/O 処理の量が多少増大することが反映されたことにより、わずかにオーバーヘッドが出たものと考えられるが、図 9 の結果は方法 B においては、I/O サイズが大きくなっても処理効率の低下がほとんどないと言える。

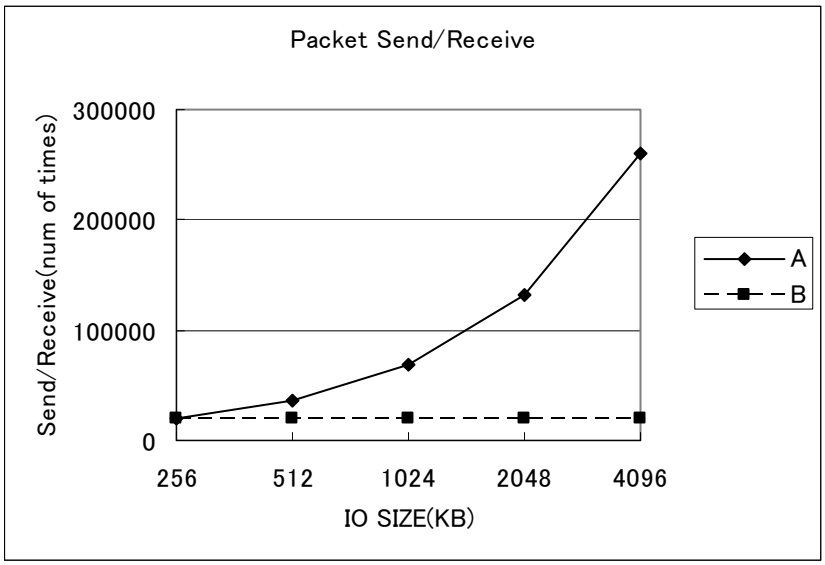

図８：方法 A、方法 B における I/O サイズに対するパケット転送回数の変化。

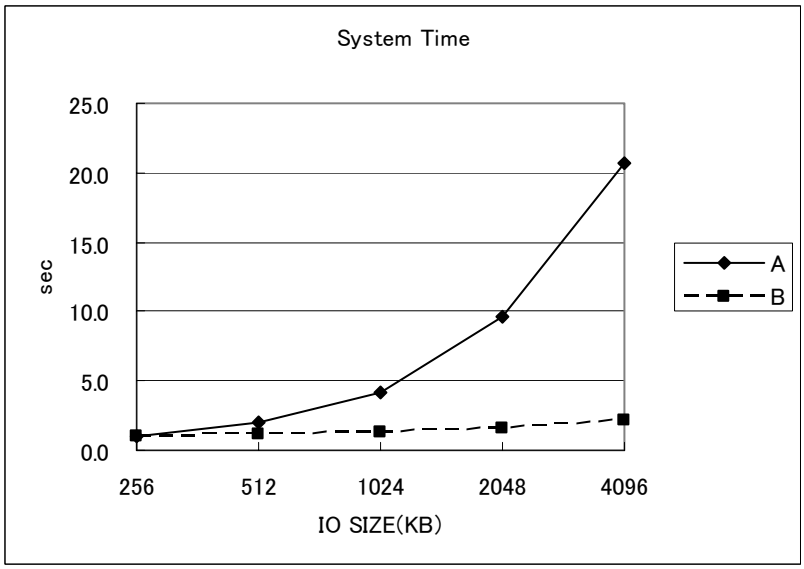

図 9 :方法 A､方法 B における I/O サイズに対する I/O 発行プロセスのシステムタイムの変化。

５．４ 複数プロセスの性能

前節までは、単一の I/O 性能について検証を行ったが、複数の I/O を同時に発行した場合のスケーラビリティーも検証する必要がある。同一ファイルシステムへ複数の I/O を同時に発行した

場合、各プロセスの性能値は、単一の I/O に比べて性能が低下するのは当然である。しかし、理論上は各プロセスの性能値を合計した場合、同時に発行する I/O 数(プロセス数)に依存せず、性能は一定になるはずである。

そこで、同じファイルシステム内の異なるファイルへ、複数のプロセスが同時に I/O を発行した場合のリード性能を測定した。図１０に、４ストライピング、ストライピングサイズ 64K バイト、I/O サイズ 2M バイトにおいて、１～８プロセスについての各プロセスの性能の合計値を示す。

２～８プロセスの場合については、性能の合計値の差は、2.6%以内であり、ほとんど差が見られず、スケーラビリティーの低下は見られない。

なお、１プロセスと、2 プロセス以上で多少性能差が見られることについては、２プロセス以上の場合はディスク装置側の処理とホスト側の処理がオーバーラップして動けるため、１プロセスの場合より処理効率が良くなるためであると考えられる。

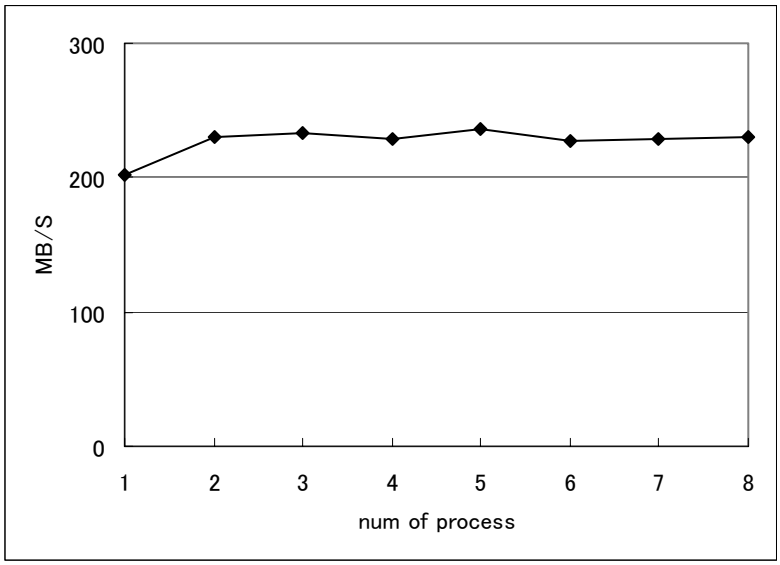

図１０：同一ファイルシステムへ複数の I/O を行った際の各プロセスの性能の合計値。

６．おわりに

本手法を適用した場合、以下のような効果が得られることがわかった。

1. 本手法は、特殊な機能を持たない、一般に入手可能なディスク装置を使用する場合でも、適用可能。
2. 本手法は、I/O のユーザインターフェースとしては、通常の read/write システムコールを用いるので、アプリケーションプログラムを変更する必要がない。
3. I/O サイズが小さい場合は、NFS の I/O として処理するので、クライアントにおいて複雑なキャッシュ制御を行う必要がなく、開発工数を低減できる。
4. GFS の性能は、128KB 以上の比較的大きな I/O サイズではローカル I/O の 90%以上であり、ローカル I/O に近い性能が得られる。
5. 上記のことは、本手法(デーモン方式)によって得られる性能と、TPT 方式によって得られる性能は、ほぼ同等であると言える。
6. NFS の性能は、Gigabit Ether を使用しても GFS の高々56%である。
7. ストライピング時の GFS のスケーラビリティーは、４ストライピングにおいて理論値の 88%である。
8. サーバーにおいて、ディスク装置毎に制御情報を１つの制御パケットにまとめることで、ストライピング時の処理の効率化ができる。
9. 複数の I/O を同時に発行した場合でも、処理効率の低下は見られない。

上記のことから、本手法は、ストライピングした場合でも処理効率の低下がほとんどなく、且つ単一の I/O 性能、及び複数の I/O を同時に発行した場合において、十分な有用性があることがわかった。

７．今後の課題

本論では、４ストライピングまでのストライピングのスケーラビリティーを検証したが、さらに多数のストライピング数でのスケーラビリティーを維持するためには、I/O ロックの細粒度化が必要であると思われる。このため、本手法においても今後カーネル 2.5 における I/O ロックの細粒度化に対応し、その効果を検証する必要がある。

参考文献

[1] Endoh, H., “Performance Improvement of Network File System Using Third-Party Transfer”, NEC Res. & Develop., Vol.39, No. 4, pp.403-406, Oct. 1998.

[2] 大谷、遠藤 ;「ネットワークファイルシステムのデータ転送方法」、公開特許広報、特開平 11-338792, 1999.

[3] Preslan, K. W. et al., “A 64-bit, Shared Disk File System for Linux”, In the seventh NASA Goddard Conference on Mass Storage Systems and Technologies in cooperation with the sixteenth IEEE Symposium on Mass Storage Systems, pp.22-41, San Diego, CA, Mar. 1999.

[4] Preslan, K. W. et al., “Implementing Journaling in a Linux Shard Disk File System”, In the Seventeenth Mass Storage System Symposium held jointly with the Eighth NASA Goddard Conference on Mass Storage Systems and Technologies, New York Mar. 2000.

[5]「 CXFS: クラスタファイルシステム」, White paper, 日本 SGI(株), http://www.sgi.co.jp/product/pdf/sanxfswp.pdf.

[6] “XFS: A High-performance journaling file system”, http://oss.sgi.com/projects/xfs.

著者紹介

大谷　敦久

1991 年日本電気(株)入社。以来、スーパーコンピュータ SX シリーズ のオペレーティングシステム SUPER-UX の開発に従事。現在、コンピュータソフトウェア事業本部第一コンピュータソフトウェア事業部主任。電子情報通信学会、日本リモートセンシング学会各会員。

青野　寛

1992 年日本電気(株)入社。以来、スーパーコンピュータ SX シリーズ のオペレーティングシステム SUPER-UX の開発に従事。現在、コンピュータソフトウェア事業本部第一コンピュータソフトウェア事業部主任。情報処理学会会員。

外丸　浩子

1988 年日本電気(株)入社。以来、スーパーコンピュータ SX シリーズ のオペレーティングシステム SUPER-UX の開発に従事。現在、コンピュータソフトウェア事業本部第一コンピュータソフトウェア事業部主任。

佐竹康司

1990 年東北日本電気ソフトウェア(株)入社。以来、スーパーコンピュータ SX シリーズ のオペレーティングシステム SUPER-UX の開発に従事。現在、ソフト開発事業部基盤ソフト開発部勤務。