HA8000のファイルシステムについて

田浦健次朗
情報理工学系研究科・情報基盤センター (兼務)

1 はじめに

現在, HA8000 システムでは日立ストライピングファイルシステム (Hitachi Striping File System; 以下 HSFS), およびネットワークファイルシステム (Network File System; 以下 NFS) の, 2 種類のファイルシステムが運用されています. また, 2010 年 4 月からは, さらに新しいファイルシステムとして, Lustre ファイルシステム [5] の運用を開始します. 同時に HSFS をアクセスするネットワークも高性能な物に変更されます.

HA8000 システムの導入当時から運用されているファイルシステムは HSFS で, これは複数のサーバに負荷を分散し, クライアント数が増えた際もスケーラビリティを得ることができる一方で, ファイルやディレクトリの作成・消去・open などのメタデータ操作の性能や, 小規模な IO の性能が非常に悪く, 対話的な利用における快適さや, 小さいファイルを多用するアプリケーションの性能に問題があります. HA8000 の運用開始後に, NFS や Lustre ファイルシステムが導入される事になった理由も, まさしくそこにあります.

本稿執筆の目的は HA8000 で運用中, ないし運用予定のファイルシステムについて整理された情報や, ベンチマーク結果を提供し, 使い分けの判断材料を提供する事にあります. ファイルシステムはどれも同じ API でアクセスできるため, ファイルシステム間の乗り換えは容易です. データの移動をすれば良く, プログラムの作り直しなどの手間は発生しませんので, ぜひうまく使い分けて作業効率や性能を向上させていただければ幸いです.

それ以前に, まずは一度各ファイルシステムによって提供されているディレクトリに, アクセスしてみることをお勧めします. フリーソフトをダウンロードして tar ball の展開や, configure, make と言った日常利用をやってみるだけで, 少なくとも対話的な作業 (プログラムの編集やコンパイルなど) には, NFS の方がはるかに良いという事が一目瞭然となると思います. 大きなデータの IO(プログラムのチェックポイントなど) にはどちらが向いているか, その際の注意点などについては本稿の情報を参考にしていただければ幸いです.

2 HA8000のファイルシステム

2.1 ディレクトリとファイルシステム

HA8000 のファイルシステム構成について, 2010 年 4 月に稼働予定の部分を含めて表 1 に要約します. また, 現状の構成 (HSFS および NFS 部分) は, 利用の手引き第 4 章 [9] や, HA8000 FAQ[8] 「システム全般, サービス内容」の節に記述されています.

表 1: ディレクトリと, それを提供するファイルシステム, 運用方針

ディレクトリ	ファイルシステム	運用	quota
/home/ユーザ名	HSFS	運用中	申請による
/short/ユーザ名	HSFS	運用中	5 日間まで自由
/nfs/all/ユーザ名	NFS	運用中	10 GB まで自由
/nfs/{ グループ名,personal}/ユーザ名	NFS	運用中	申請による
/lustre/ユーザ名	Lustre	2010 年 4 月より	(∗)
/tmp	ローカル (ext3)	運用中	1 ジョブ実行中自由

∗: /lustre/ユーザ名 の quota (使用量制限) は, 1 ユーザ当たり 10GB まで自由, それ以上は申請によります. より正確には,「グループごとに, 10 GB× グループのユーザ数」までは自由, です. 20 人のグループであれば, そのグループ全体で 200GB までは自由になります.

HA8000 導入当初から運用されているファイルシステムは HSFS です. 従ってデフォルトの設定では利用者のホームディレクトリは HSFS になっています. 異なるファイルシステムを試しに利用してみたい場合, まずは/nfs/all/ユーザ名 (NFS の場合), もしくは/lustre/ユーザ名 (Lustre の場合. 2010 年 4 月以降) というディレクトリを使って見てください.

```
ha8000-2:% cd /nfs/all/h20000
ha8000-2:% wget http://www.iozone.org/src/current/iozone3_327.tar
ha8000-2:% tar xvf iozone3_327.tar
```

どのディレクトリがどのファイルシステムによって提供されているかは, df コマンドを使って調べることが出来ます

```
ha8000-2:% df
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/sda1             20161172  16634580   3321764  84% /
/dev/sda6            134463688    135328 132962276   1% /tmp
/dev/sda3             10080520   2712024   7266084  28% /var
tmpfs                 16448140         0  16448140   0% /dev/shm
home1-ms             111123983232 26191419648 84932563584  24% /hsfs/home1
home2-ms             111123983232 31417428992 79706554240  29% /hsfs/home2
home3-ms             111123983232 14702180608 96421802624  14% /hsfs/home3
home4-ms             111123983232 25241878976 85882104256  23% /hsfs/home4
short-ms             111126372480 50517453760 60608918720  46% /short
iy050:/export/home   16910278560 203272896 15848012416   2% /nfs/all
iy051:/export/work1  16910278560 2764675520 13286609792  18% /nfs/work1
iy051:/export/work2  16910278560 297513536 15753771776   2% /nfs/work2
iy052:/export/work3  16910278560    132064 16051153216   1% /nfs/work3
iy052:/export/work4  16910278560    131232 16051154080   1% /nfs/work4
/dev/sda2             70557084    435296  69404968   1% /maint
```

/nfs/all/ユーザ名 というディレクトリは申請なしに利用可能で, 各ユーザ 10GB まで利用可能な様に quota (使用量制限) が設定されています. NFS を 10 GB を越えて使いたい場合, これまで自分に (HSFS 領域として) 割り当てられていた領域の一部を移行させることができます. その際割り当てられ

るディレクトリ名は, /nfs/グループ名/ユーザ名 になります. 申請はグループごとに行います. 申請は uketsuke@cc.u-tokyo.ac.jp にメールを送ればよく, 詳しくは上述の FAQ[8] にあります.

Lustre も似た運用となります (本稿掲載の記事「HA8000 クラスタシステムへの Lustre ファイルシステムの導入 (予定) について」もご参照ください) が, ディレクトリは/lustre/ユーザ名 というディレクトリに統一されています. また, 無料で使える quota は課金グループごとにまとめられており, グループ全体で「10GB × そのグループごとのユーザ数」に設定されています.

2.2 ファイルシステムの分類と一般的構成

ファイルシステムは各ノードだけでアクセスできるディレクトリやファイルを提供する, **ローカルなファイルシステム**と, 多数のノードからアクセスできるディレクトリやファイルを提供する, **共有ファイルシステム**とに大別されます. HA8000 では, ローカルなファイルシステムのうち一般ユーザが書き込める領域は/tmp です. /home, /short, /nfs, /lustre などのディレクトリはすべて共有ファイルシステムによって提供されます. 本稿の主な興味は共有ファイルシステムの性能です.

共有ファイルシステムは, ファイルシステムのクライアント (各計算ノード) が, ファイルサーバにネットワークを介してアクセスすることで実現されます. 従ってその性能は,

- ファイルシステムソフトウェア (アーキテクチャ, プロトコル, 設定)
- クライアントの性能 (CPU, メモリ, ネットワークインタフェース (NIC))
- ファイルサーバの性能 (CPU, メモリ, ネットワークインタフェース (NIC))
- ファイルサーバの台数
- クライアントとファイルサーバをつなぐネットワークの性能
- ファイルサーバのディスク (RAID コントローラ) 性能

など, 多数の要素に左右されます.

共有ファイルシステムのうち単純なものは, **単一サーバ型**の構成で, NFS がこれに相当します. 基本的にはサーバのファイルシステムのうち, あるディレクトリから下 (サブツリー) が, そのままクライアントが指定したディレクトリの下に見える, という形態です. このために行う操作をファイルシステムのマウントと言います. 単一サーバのファイルシステムでは, クライアントに公開するサブツリーの一部を別のサーバに持たせて負荷分散をする事はできません. それがしたい場合はクライアントが明示的に複数のサーバをマウントする必要があります. そうすれば, 原理的にはマウントしたサーバ数分のアクセス性能を得ることができますが, その場合でもサブツリー単位の大まかな負荷分散しかできません. 例えば, ディレクトリ構造中の単一のディレクトリを提供しているのは常に一台のサーバとなり, そこへアクセスが集中してもこれを負荷分散する事はできません. また, サーバ上のディレクトリ構造がそのままクライアントに見えるという仕組みなので, 負荷の偏りに合わせてサーバの分担を (ディレクトリ名を変えずに) 変更する事も困難です.

これらの問題を解決するのが**並列ファイルシステム**で, HSFS, Lustre[5], GPFS[2] などがこれに該当します. 要するにディレクトリ構造上の論理的な位置に関係なく, ファイルを複数のサーバ (**IO サーバ**) に配置できるのが並列ファイルシステムです. 設定によっては単一のファイルを複数の IO サーバに分散配置 (ストライピング) する事もあります. ユーザは単一のディレクトリ, 時に単一の巨大なファイル

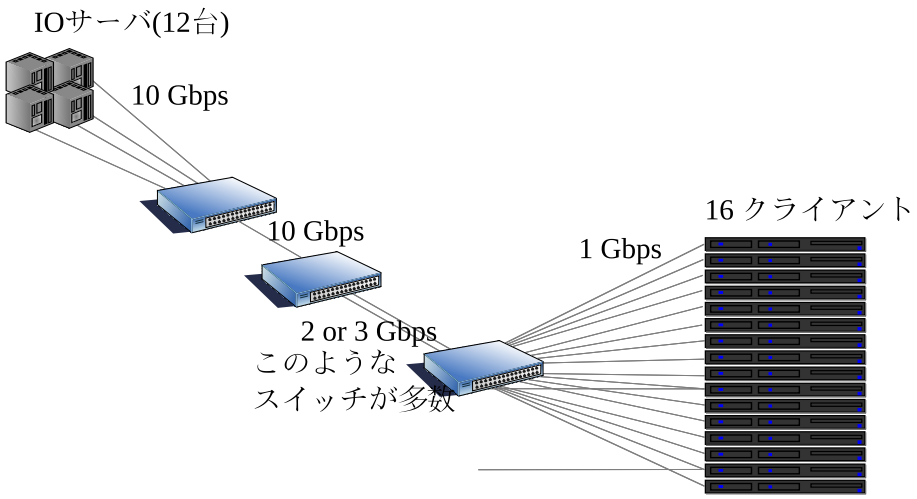

図 1: HSFS クライアント (16 台) とファイルサーバをつなぐネットワークの模式図 (2010 年 4 月まで). 16 台のクライアントが 1 ホップ目のスイッチにつながり, そこからのアップリンクは 2 または 3Gbps のボトルネックになっている.

を使いながらも, アクセス負荷が分散されます. なお, HSFS でもファイルのストライピングが可能ですが, 現在の HA8000 の設定では単一のファイルは一つの IO サーバに割り当てられます.

並列ファイルシステムは, その仕組み上, どのファイル (またはどのファイルのどの部分) が, どの IO サーバに配置されているかを管理する仕組みを必要とします. このために多くの場合一つのサーバが用いられ, そのサーバは**メタデータサーバ (MDS)** と呼ばれます. 並列ファイルシステムにおいてメタデータサーバを経由しなくてはいけない処理 (ファイルの open やディレクトリの作成など. 一般にメタデータ操作と呼ぶ) は, 単一サーバ構成の場合と比べて複雑化し, かつクライアント・サーバ間のメッセージ数も増える傾向にあります. 従ってメタデータ操作の性能は, 並列ファイルシステムだからと言って速くなるわけではないのが普通です.

2.3 HA8000 のストレージとネットワークの構成

HA8000 において, クライアントが各ファイルシステムを用いる際に関わるハードウェアの性能を表 2 に示します.

表 2: HA8000 のファイルシステムに関係するハードウェア諸元

	クライアント NIC	サーバ NIC	ネットワーク	ディスク (write)‡	ディスク (read)‡	IO サーバ並列度
HSFS	1 Gbps Ethernet∗	10 Gbps Ethernet	GigE†	180 MB/sec	370 MB/sec	12
NFS	10 Gbps Myrinet	10 Gbps Myrinet	Myrinet + 10GigE	180 MB/sec	370 MB/sec	1
Lustre	10 Gbps Myrinet	10 Gbps Ethernet	Myrinet + 10GigE	3 GB/sec	3 GB/sec	6

- ∗: 2009 年度末に 10 Gbps IP over Myrinet に移行予定です.
- †: 現在のネットワーク構成では 16 ノードが IO サーバへの 2 Gbps または 3 Gbps のボトルネックリンクを共有しています (図 1). ジョブがどのノードに割り当てられるかにもよりますが, N ノードをクライアントとして用いても, 1 Gbps $\times N$ のバンド幅が出るとは限りません. 他のジョブによる影響を考えないとしても, 同一スイッチに多数のノードが割り当てられると, それら全体で, 2

Gbps または 3 Gbps が上限となります. 2009 年度末に 10 Gbps IP over Myrinet に移行時にこれも解消されます.

- ‡ : 「ディスク」の性能とは実際にはディスクコントローラ, HA8000 の場合はディスクアレイに搭載されている RAID コントローラの性能を意味します.

3 ファイルシステムベンチマーク

ファイルシステムベンチマークは様々な物が作られ, 利用可能です. read/write の IO 性能を中心に測定する物として, IOZone[4], Bonnie++[1], ファイル作成などメタデータを中心とした物としては, Filebench[6] 並列 IO にフォーカスした物としては IOR[3] などがあります. ここでは筆者の共同研究者である頓南氏 (情報理工学系研究科コンピュータ科学専攻) による paramark[7] を用います. Paramark はメタデータ操作の性能, read/write の性能の両方を測定でき, かつ並列クライアントからのアクセスを集計する機能を持つため, 本稿の目的に合致しています.

4 ファイルを作る・開く性能

最初に, ファイルを新規作成する, および既存ファイルを開くのにかかる時間を単独で取り出してみます. どちらもシステムコールとしては open および close システムコールの経過時間を測定する事に相当します.

新規に作成するには,

```
int fd = open(path, O_CREAT|O_WRONLY|O_TRUNC, 0644);
close(fd);
```

を, 既存ファイルを開くには,

```
int fd = open(path, O_RDONLY);
close(fd);
```

を, それぞれ 1024 回繰り返し, 1 秒間の操作回数を測定します. これを 10 回繰り返し, 95%の信頼区間を求めています (以降の実験でも同様). 前者の場合, 作成されるファイルの名前 (path) は, 元々存在していないファイルの名前です.

ファイルの open/close にかかる時間は, 以下の理由から重要です.

1. どんなファイルの作成あるいは読み込みであっても当然のことながら上記の open/close 操作は最低一度ずつ, 必ず行わなくてはなりません. 従って read/write 操作のスループットが高くても, 実際のファイル作成や読み込みに費やす時間は, これに open/close の時間が上乗せされた物になります.

2. 特に HSFS ではこの部分のオーバーヘッドが大きいため, 小さなファイルでは無視できない実効性能の低下をもたらします. open/close を単独で測定する目的の一つは, どのくらい大きなファイルならこのオーバーヘッドが気にならないかを求める事にあります.

3. さらに, Lustre や HSFS などの並列ファイルシステムの場合, read/write は複数のファイルサーバへの負荷分散が行われますが, ファイルの open, close, mkdir などの処理はメタデータサーバという単一のサーバを通り, 負荷分散が行われません. したがって, 少数のクライアントであれば無視できた open/close のオーバーヘッドが, クライアント数を増すに連れて再び顕在化する可能性があります.

4.1 1 クライアント

最初にクライアントが 1 つだけの場合の結果を図 3 に示します. 数字は 1 秒間に実行できた回数で, 逆数が一回あたりの時間となります. NFS では, 読み込み・作成とも, 2700〜2800 回/秒程度 (1 回当たり 0.4ms 程度) に対して, HSFS では読み込みは 140〜200 回/秒程度 (1 回当たり 5〜7ms 程度), 作成は 12〜14 回/秒程度 (1 回当たり 80ms 程度) です. それぞれ NFS の 15 倍, 200 倍程度遅い事になります. なお, 導入予定の Lustre では, 1000 回/秒以上をクリアする予定です.

表 3: 作成と読み込みの速度 (単位: 回/秒)

	HSFS	NFS
作成	13.23 ± 1.10	2745 ± 21
読み込み	170.38 ± 30.44	2862 ± 53

HSFS でソースコードの tar ball を展開したり, コンパイルしたりするのに非常に時間がかかるのは, 一段細かく見るとここに原因があります. 仮に write の転送レートが 100MB/sec だとすると (実測値は 6 節で示します), HSFS で作成のための open/close にかかるのと同じ時間 (80ms) で, 8MB の read/write が行えることになります. この大きさのファイルを作って, 50% が open/close のオーバーヘッドだと言うことです.

一般にファイルの open や close は, 大量のデータを送るわけではないのでネットワーク性能, 特にバンド幅はあまり性能に影響しません. 従ってこの性能の主要因がネットワークにあるとは到底考えられません. クライアントとサーバ間のメッセージの往復時間, およびクライアント, サーバ内で生じる, プロトコル処理やプロセスの切り替えなどの時間などが積み重なって, これだけの時間がかかっていると考えるのが妥当な推論です.

4.2 多クライアント

次にクライアントを増やした場合に総計でどのくらいの open/close を行うことができるかを測定します. これは, ネットワークの遅延やクライアントのオーバーヘッドによらない, ファイルサーバ単独の性能を抽出する事に相当します. つまり, 仮に上述の 80ms のうちの大部分, 例えば 75ms がクライアントやネットワークの時間であった場合, サーバは 1 リクエストを 5ms の負荷で処理していることになり, クライアント数を増やすことで全体のスループットは 1 回/5ms = 200 回/秒まで向上します. 逆に 80ms の大部分, 例えば 70ms がサーバの負荷になっているとすれば, クライアント数を増やしても, 全体のスループット 1 回/70ms = 14.3 回/秒までしか向上しません. 従って, 前節で得た 80ms という数字だけで性能を表せるわけではありません.

図 2 は, 横軸にクライアント数, 縦軸に全クライアント合計での回数をとりプロットした物です. HSFS の場合, 作成は 60 回/秒, 読み込みは 1400〜1600 回当たりで飽和するようです. NFS の場合, 作成, 読み

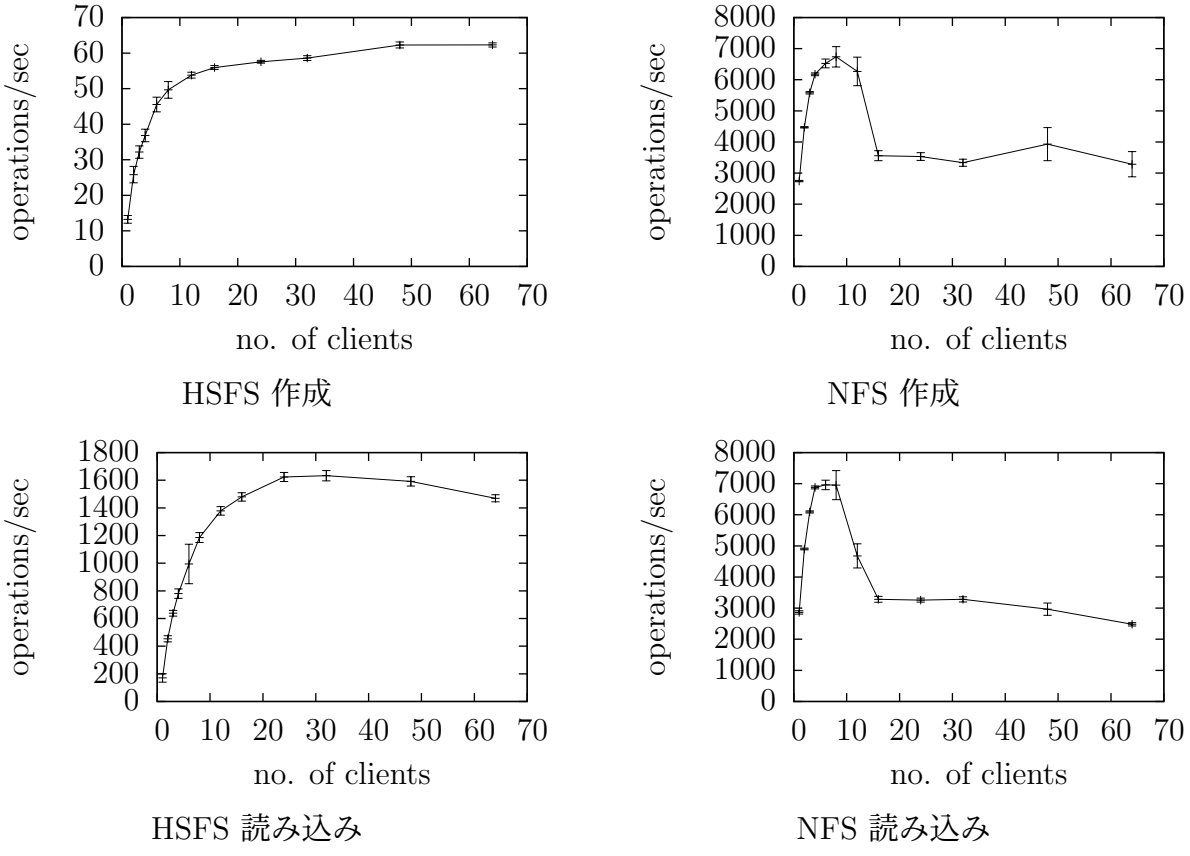

図 2: 作成, 読み込みのための open/close のスループット

込みとも 10 並列度程度で一旦 6000~7000 回/秒くらいのスループットに達しますが, それ以上並列度を上げると 3000 回/秒 ~4000 回に落ち込みます.

導入予定の Lustre では, クライアント数を適切に設定した際に 5000 回/秒以上が仕様上定められた値 (最低ライン) で, 実際には 10000 以上のスループットになる予定です.

5 ネットワークとストレージの性能

次にファイルを開いた後の read/write 性能を理解する事を試みます. どのような状況で性能が何によって律速されるはずなのかを知るために, HA8000 のファイルシステムに関係するネットワークやストレージの構成について, おさらいをしておきます.

5.1 クライアント NIC

NFS の場合, 各クライアントは MPI などの通信に使うネットワーク (Myrinet) を用いてファイルサーバへアクセスします. これは IP over Myrinet という Myrinet 上に IP 通信を行う方式で, バンド幅は最大で 10 Gbps です. [1] Lustre が導入された暁には, Lustre へのアクセスにも同じネットワークを使います. HSFS の場合, 現状では各クライアントは, MPI 用のネットワークとは異なる, 1 Gbps の Ethernet NIC を経由してファイルサーバへアクセスします. 従って現時点では, **1 クライアントから HSFS への**

[1]MPI で通信する場合は, Myrinet が複数 bonding され (束ねられ), 1 ノード当たり 40Gbps です.

アクセスでは 1 Gbps が望める最大値となります. 2010 年 4 月からは構成を変更して NFS, Lustre と同じネットワークを用います.

5.2 ネットワークのボトルネック

クライアントと HSFS を現在つないでいるネットワークは, 別のボトルネックがあります. HA8000 のクライアントは, クライアント 16 ノードごとに一つのスイッチに Ethernet で接続され, そのスイッチからサーバへ向けてのリンクが 2 Gbps もしくは 3 Gbps しかありません (図 1). したがって, 一つのジョブに割り当てられたクライアントノードが同じスイッチに接続されていた場合, **それらのクライアントで合計して 2 Gbps または 3 Gbps の転送レート**しかない事になります. なお, 一つのスイッチにはノード名が連番で割り当てられて (例えば a001∼a016 までが一つのスイッチにつながって) いるため, ホスト名の数字部分 −1 を (a159 であれば 158) を 16 で割った商 (余り切り捨て) が同じなら, 同じスイッチにつながっています. ボトルネックリンクは, ホスト名のアルファベット部分が a なら 2 Gbps, そうでなければ 3 Gbps です. ホスト数としては a が 512 ノード (全体の半数強) あります.

この様にジョブにどのノードが割り当てられるかにより, 全体でファイルサーバまでどのくらいのネットワークバンド幅が得られるかが決まります. 従って, 期待できる性能をノード数だけで確実に予測するのは困難です. もちろん同一スイッチに割り当てられた他のジョブと競合している事も考えれますので, 予測はさらに困難です. 繰り返しますが, 来年度からは構成を変更して NFS, Lustre と同じネットワークを用いますので, このネットワークボトルネックは解消します.

5.3 サーバ NIC

サーバ側は, まず (クライアントから見ると) 入り口である NIC は, NFS ではクライアントと同じ 10 Gbps の IP over Myrinet, HSFS では 10 Gbps の Ethernet を用いています. Lustre も 10 Gbps の Ethernet を用います. 雑に要約すると, サーバ 1 台の NIC 性能はどのファイルシステムも似ています.

5.4 サーバ-RAID コントローラ

一方サーバにつけられたディスクとのバンド幅は以下の様になっています. まず NFS サーバは一つの RAID コントローラと接続されています. HSFS の IO サーバは 2 台がペアになり, 2 台が 2 つの RAID コントローラと接続されています. 1 コントローラあたりのバンド幅は write が 180 MB 程度, read が 370 MB 程度との事です. 従ってサーバの最大転送性能は NIC よりもこちらで律速されます. 実測値は以下で示します.

5.5 まとめ: ボトルネックはどこ?

まず HSFS の場合です.

- 1 クライアントの場合, その 1 Gbps の NIC がボトルネックになります.
- ノード数を増やして, 運悪く一つのスイッチに多数 (3, 4, 5, . . . 程度) のノードが割り当てられると, 前述した「ネットワークのボトルネック」が律速になります.

- 運良くそれが起きないままノード数を増やせると, IO サーバのディスクがボトルネックとなります (write 180 MB/sec, read 370 MB/sec). 各クライアントが別のファイルにアクセスしていれば, それらはある程度均等に IO サーバに分散しますので, 全体としてはこの値のファイルサーバ台数倍 (write 2GB/sec 程度, read 4GB/sec 程度) まで伸びていくことが期待されます.

- 後に示しますが 1 クライアントのスループットの実測値は 70MB/sec 程度のようですので, write の場合 30 ノード, read の場合 60 ノード程度まで, 上記の値に向けてバンド幅が伸びていくというのが期待です. 残念ながら筆者はその台数のノードを使うことが出来ませんので, 今回は 8 ノードまでの結果を示します.

NFS の場合です.

- クライアント・サーバとも NIC が 10 Gbps であり, サーバからディスクへの IO がそれよりも遅いため, 1 クライアントであっても NFS サーバのディスク (RAID コントローラ) のボトルネック (書き込み 200MB/sec, 読み込み 400MB/sec 程度) を観測することになります.

- クライアントの並列度を上げても, 当然の事ながらスループットはすでにほぼ頭打ちになっていて, 向上しない事が予想されます.

導入予定の Lustre では,

- クライアント・サーバとも NIC が 10 Gbps である他, サーバのディスクへのアクセスバンド幅がそれより大きいため, 1 クライアントからのアクセスに対して, 10G Ethernet や, IP over Myrinet の速度を観測することになります. 実際には 650〜750MB/sec 程度の速度を予想・期待しています.

- そして, サーバが 6 台ある事からクライアントの並列度を上げると, 約その 6 倍程度にはスケールする事が予想・期待されます. HA8000 では Myrinet の接続性が非常に高い事から, これは 6 ノード程度のクライアントを用いれば (それらがネットワーク上のどこに割り当てられていても) 達成できると期待されます.

以上を踏まえて実験結果を見ていきます.

6 書き込み性能

6.1 1 クライアント

図 3 は, 一クライアントが 256MB のファイルを書き込む際のスループットを示しています. 横軸はブロックサイズで, write システムコールを一回発行する際に書き込むデータの量 (単位: KB) です. 測定点は 8KB, 32KB, 128KB, . . . , 256MB まで, 4 倍刻みです.

二つの線は, open システムコールにフラグ `O_SYNC` をつけた場合と, つけなかった場合です. 前者はファイルを

```
int fd = open(path, O_CREAT|O_WRONLY|O_TRUNC|O_SYNC, 0644);
```

で開き, 後者は,

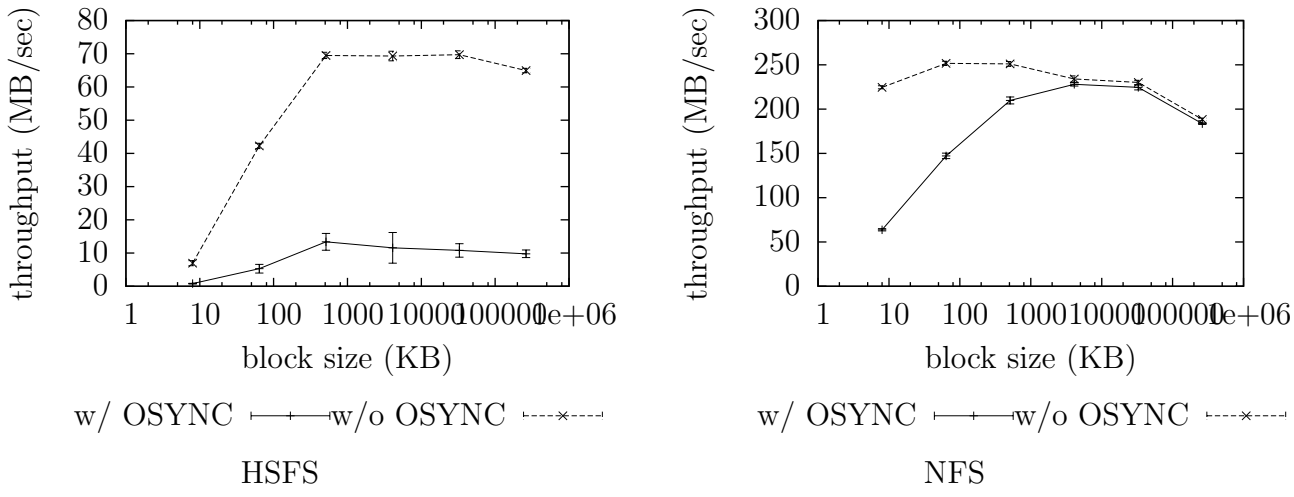

図 3: 1 クライアントによる書き込み

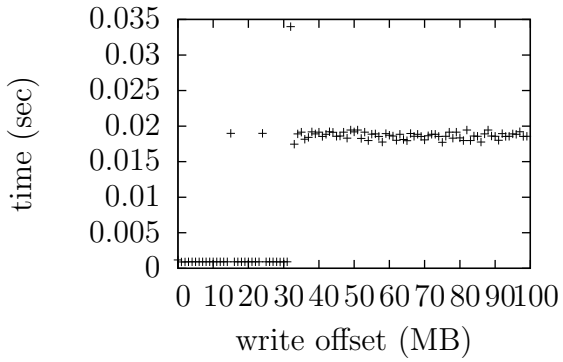

図 4: 先頭のから数えて x MB 目 (オフセット $(x-1)$M〜 xM のデータ) を書くのにかかった (write システムコール内で経過した) 時間. 32MB 付近にあるギャップが, 遅延書き込みのバッファが満杯になったことを示していると考えられる.

```
int fd = open(path, O_CREAT|O_WRONLY|O_TRUNC, 0644);
```

で開きます. open システムコールを直接用いずに, C のストリームライブラリや C++の標準ライブラリなどを用いる場合も, 特に意識していなければ `O_SYNC` はつけずに open している事になります.

両者の違いは実際の IO サーバへの転送が行われるタイミングが, 遅延されるかどうかという所にあります. open 時にフラグ `O_SYNC` をつけると毎回の write 時に書き込みがサーバに転送され, write からリターンした時にはデータはサーバに到着しています. つけないと, write によって書き込まれたデータは, 必ずしもその write からのリターン時にすでにサーバに到着しているわけではなく (**write behind; 遅延書き込み**), 後から非同期に書き込まれます. このため, プロトコル上サーバとのメッセージの往復があっても, クライアントの write システムコール自体はその遅延があたかも 0 であるかのように処理を終える事が出来, スループットが向上します.

もちろん書き込みを遅延するにはバッファが必要で, 従って無条件で write を直ちに終了できるわけではありません. 例えばバッファサイズが 32MB であれば, その 32MB のバッファが既に満杯のときに write システムコールが発行されたら, そのシステムコールはバッファが一部空くまで—つまりバッファの一部がサーバに処理されるまで— ブロックします.

図 4 に, ファイルの先頭からブロックサイズ 128KB でデータを書いて行き, 1 MB 書く度に時計を見て, 各 1 MB を書くのにかかった時間を先頭からプロットした物です. つまり, x 軸が 1 のところは, 最初の 1 MB を書くのにかかった時間, x 軸が 2 のところは, 次の 1 MB のデータを書くのにかかった時間, ... です. ここで, 「書くのにかかった時間」とは単に, クライアントが write システムコール内で経過した時間ということであり, 必ずしもサーバにデータが届いたことを意味しません. グラフから一目見て明らかなように, 32MB 付近に明らかな不連続点があります. これは一度に書き込むサイズをどう変えても, 大体この場所に現れます. つまり現在の HA8000 上では, HSFS の遅延書き込みバッファの大きさが 32 MB 付近であることが推測されます.

経過時間の測定には open にかかる時間を含めておらず, close は含めています. つまり, 上記の後, 以下の処理にかかる時間を測っています.

```
while (256MB 書けるまで) {
   write(fd, buf, ブロックサイズ);
}
close(fd);
```

O_SYNC をつけてもつけなくても, close からリターンした時にはサーバにデータが到着しています. 従って O_SYNC をつけない場合, close 時まで書き込みが遅延されたデータがそこで転送され, close はその終了を待ってリターンします. 実際そこでかなりの時間が費やされます. close 時に実際の転送が行われているのですから, 時間計測に close を含めないのは間違いです.

前置きが長くなりましたが, 図 3 に戻って結果を見ます. まずはどちらも遅延書き込み (O_SYNC なし) をした方が性能が高いのは頷けます. 最大性能は NFS で 250MB/sec 程度でこれは, だいたいサーバディスクの write バンド幅です. HSFS は 70 MB/sec 程度で, これは 1 Gbps のネットワークに律速されていますが, それよりも少し遅いです.

グラフを見ると, O_SYNC ありの場合に, 性能がブロックサイズに敏感な事が分かります. これは, O_SYNC ありの write では, サーバへの到着の確認が出来てから write がリターンします. その間クライアントは待たされており, 小さなブロックサイズではその時間が相対的に大きくなります. HSFS では殊に顕著であり, さらにいくらブロックサイズを大きくしてもスループットは伸びません. 必ずしもユーザが発行したシステムコールのサイズでクライアント-サーバ間の転送が行われるわけではないので, 頭打ちになる事自体は不思議ではありません. [2] そして, おそらく一つの書き込みの処理にかかるサーバ側での遅延が一度に転送されるデータに比べて大きいために, このようなスループットになる物と予想されます.

これに対し O_SYNC なしの場合 (こちらが普通), 性能はブロックサイズによるものの, 割と小さいブロックサイズ (NFS で 64KB, HSFS で 512KB) で, 性能がほぼ最大化しています. 特に NFS では 8KB 程度のブロックサイズでもあまり性能に遜色はありません. HSFS では 512KB 程度のブロックサイズにする事が推奨されます.

6.2 write システムコール以外の書き込み

以上は C で write システムコール, いわば OS のインタフェースを直接用いる IO の場合でしたが, 他の書き方 (例えば C のストリームライブラリ, C++の標準ライブラリ, その他の言語の入出力) をした場合はどうなるでしょうか. それらも結局, ほとんどの場合最終的には write システムコールを呼んでいます. それをどのサイズで呼んでいるかは, strace コマンドで調べることが出来ます. 以下は HA8000 上の

[2] 確認をしていませんが, グラフを見ると 512KB が最大の転送単位なのかもしれません.

gcc, g++コンパイラを用いた場合で, コンパイラ (正確にはライブラリ) の実装で結果は異なるかもしれませんので, 調べて見てください.

C のストリームライブラリ　例えば以下の C プログラム

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
  int i;
  FILE * fp = fopen("hoge.dat", "wb");
  for (i = 0; i < 16 * 1024 * 1024; i++) {
    fprintf(fp, "0123456789abcde\n");
  }
  fclose(fp);
}
```

を gcc でコンパイルして,

```
ha8000-2:h20000% strace ./a.out
```

として, NFS ディレクトリ上で実行すると以下の出力が得られました.

```
execve("./a.out", ["./a.out"], [/* 69 vars */]) = 0
brk(0)                                  = 0xa043000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) \
      = 0xb7fda000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
   ... <途中省略> ...
write(3, "0123456789abcde\n0123456789abcde\n0"..., 4096) = 4096
write(3, "0123456789abcde\n0123456789abcde\n0"..., 4096) = 4096
write(3, "0123456789abcde\n0123456789abcde\n0"..., 4096) = 4096
write(3, "0123456789abcde\n0123456789abcde\n0"..., 4096) = 4096
   ... <以下省略> ...
```

つまりこのプログラムでは write を 4096 バイト単位で出しているらしい, という事がわかります. 一方これを HSFS 上で実行すると, 上記の write の部分が

```
   ... <以上省略> ...
write(3, "0123456789abcde\n0123456789abcde\n"..., 65536) = 65536
write(3, "0123456789abcde\n0123456789abcde\n"..., 65536) = 65536
write(3, "0123456789abcde\n0123456789abcde\n"..., 65536) = 65536
   ... <以下省略> ...
```

となり, 64KB 単位で出しているらしい, という事が分かります.

setvbuf というライブラリを関数を用いる事でこれを変更できます (説明用のためエラーチェックなどを省略しています).

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
  int i;
  FILE * fp = fopen("hoge.dat", "wb");
  size_t bufsiz = 512 * 1024;
  setvbuf(fp, malloc(bufsiz), _IOFBF, bufsiz);
  for (i = 0; i < 16 * 1024 * 1024; i++) {
    fprintf(fp, "0123456789abcde\n");
  }
  fclose(fp);
}
```

すると先ほど 4096 バイトや 64KB ごとに行われていた write システムコールが,

```
   ... <以上省略> ...
write(3, "0123456789abcde\n0123456789abcde\n0"..., 524288) = 524288
write(3, "0123456789abcde\n0123456789abcde\n0"..., 524288) = 524288
write(3, "0123456789abcde\n0123456789abcde\n0"..., 524288) = 524288
   ... <以下省略> ...
```

のように 512KB 単位で行われるようになります.

C++の標準ライブラリ　C++ (g++) ではどうでしょうか.

```
using namespace std;
#include <iostream>
#include <fstream>

int main()
{
  int i;
  ofstream ofs("hoge.dat");
  for (i = 0; i < 16 * 1024 * 1024; i++) {
    ofs << "0123456789abcde\n");
  }
  ofs.close();
}
```

のようなプログラムを g++でコンパイルして strace で実行すると,

```
   ... <以上省略> ...
writev(3, [{"0123456789abcde\n0123456789abcde\n0"..., 8176}, \
          {"0123456789abcde\n"..., 16}], 2) = 8192
writev(3, [{"0123456789abcde\n0123456789abcde\n0"..., 8176}, \
          {"0123456789abcde\n"..., 16}], 2) = 8192
writev(3, [{"0123456789abcde\n0123456789abcde\n0"..., 8176}, \
          {"0123456789abcde\n"..., 16}], 2) = 8192
   ... <以下省略> ...
```

となり, デフォルトのバッファサイズは 8192 バイトのようです. そして g++の場合, HSFS にアクセスしたからといって自動的にバッファサイズが大きくなることはありません. さらなる注意としては, 改行として文字列に改行文字 (\n) を含めるのと, endl を送るのは違うという点があります. これはライブラリ自身の仕様で, endl を送るとそこで write システムコールが発行されます.

つまり,

```
    ofs << "0123456789abcde\n";
```

のところを,

```
    ofs << "0123456789abcde" << endl;
```

とすると, 16 バイト単位で write システムコールが発行され, 大打撃となります.

C++の標準ライブラリでバッファサイズを増やす (以下では 512KB) には, 以下の様にします.

```
using namespace std;
#include <iostream>
#include <fstream>

int main()
{
  int i;
  ofstream ofs;
  int bufsiz = 512 * 1024;
  ofs.rdbuf()->pubsetbuf(new char[bufsiz], bufsiz);
  ofs.open("hoge.dat");
  for (i = 0; i < 16 * 1024 * 1024; i++) {
    ofs << "0123456789abcde\n";
  }
  ofs.close();
}
```

実質的には,

```
ofs.rdbuf()->pubsetbuf(new char[bufsiz], bufsiz);
```

の 1 行ですが, open に先立ってこれを呼ぶようにしないと変更が有効になりません. つまり以下は意味がありませんので注意してください.

```
ofstream ofs("hoge.dat");
ofs.rdbuf()->pubsetbuf(new char[bufsiz], bufsiz);
```

次に注意が必要な点は, C でも C++でも, 書式付き出力を行うと, たちまちその変換 (つまり CPU) が律速となり, ファイルシステムの性能とは無縁の (低い) 性能になるということです. 試しに上記の,

```
ofs << "0123456789abcde\n";
```

を,

```
ofs << "x = " << 1.2 << " y = " << 3.4 << "\n";
```

へ,

```
fprintf(fp, "0123456789abcde\n");
```

を

```
fprintf(fp, "x = %f y = %f\n", 1.2, 3.4);
```

へ, それぞれ変更すると, C++でバッファを明示的に大きくせずに HSFS に書き込むという場合を除き, CPU 律速になります.

実験結果　以上を踏まえて, 上記のプログラムで 256MB のファイルを作ったときの書き込み速度をまとめます.

表 4: gcc/g++のライブラリでの書き込み性能

処理系	バッファサイズ	NFS	HSFS
gcc	default	136.93	43.73
gcc	512k	138.53	59.83
g++	default	131.13	4.00
g++	512k	132.82	42.96

書式なし文字列 ("0123456789abcde\n")

処理系	バッファサイズ	NFS	HSFS
gcc	default	17.78	11.44
gcc	512k	17.55	13.73
g++	default	10.07	2.73
g++	512k	10.09	8.70

文字列と書式付き double (本文参照)

まずは double を文字列に直して書くと非常に遅いという事. また, g++で HSFS に書いた場合は際立って遅く, これはバッファサイズが 8KB である事が原因です.

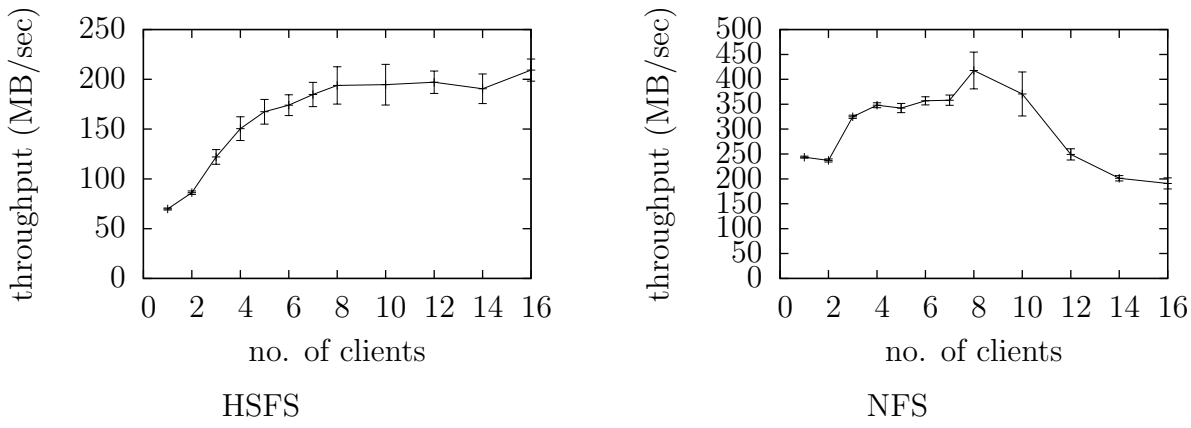

図 5: 多クライアントによる書き込み

6.3 多クライアント

5 節で述べたような構成・状況であるため, 現状運用されている NFS, HSFS においては大規模な read/write 性能はクライアント数を上げてもあまり向上しません. NFS ではサーバが 1 台でディスクのボトルネックにすぐに当たり, HSFS ではネットワークのボトルネックに多くの場合当たります.

図 5 にファイルサイズ 256MB でクライアント数を 1 から 16 まで変えた際のスループットを示します. ブロックサイズは先の実験でよい値を示すことが分かった, 512KB を使います.

予想通り, NFS は 1 台でそこそこ良い性能 (250MB/sec; disk の書き込み速度に近い値) を示すものの, クライアントノード数を増やしても性能は伸びません. open の際と同様, 並列度を上げて高負荷をかけると, むしろスループットが下がります. 一方 HSFS は 1 台の性能は 1 Gbps (120 MB/sec 程度) に律速されており, 実際にはそれ以下 (70 MB/sec 程度) しか出ませんが, 台数を増やすとスケールし, 例の 2 Gbps のボトルネックリンクで飽和している様子です.

なお, この実験で割り当てられたホストは, a194,a195,a196,a197,a201,a202,a204,a208 の 8 つで, 数字 −1 を 16 で割ると, すべて 12 となり, 見事にすべて同じスイッチになっていました. これは不運なケースといえますが, 一般に連続した (固まった) 番号のホストが割り当てられることが多いようです.

7 読み込み性能

7.1 1 クライアント

図 6 は, 一クライアントが 256MB のファイルを読み込む際のスループットを示しています. グラフの意味と見方は write の際と同様です. つまり,

```
int fd = open(path, O_CREAT|O_WRONLY|O_TRUNC, 0644);
```

とした後, 以下の処理にかかる時間を測っています.

```
while (256MB 読めるまで) {
   read(fd, buf, ブロックサイズ);
```

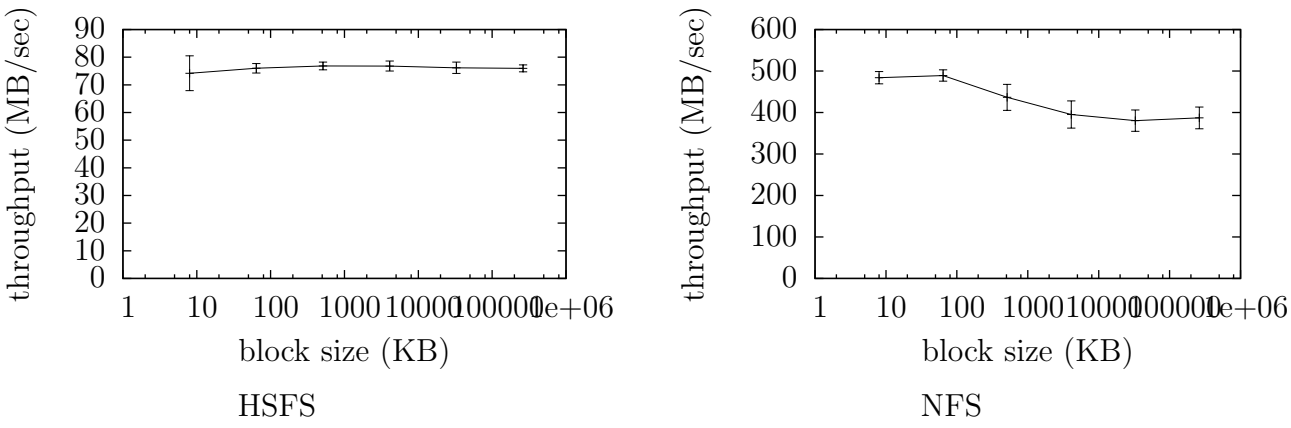

図 6: 1 クライアントによる読み込み

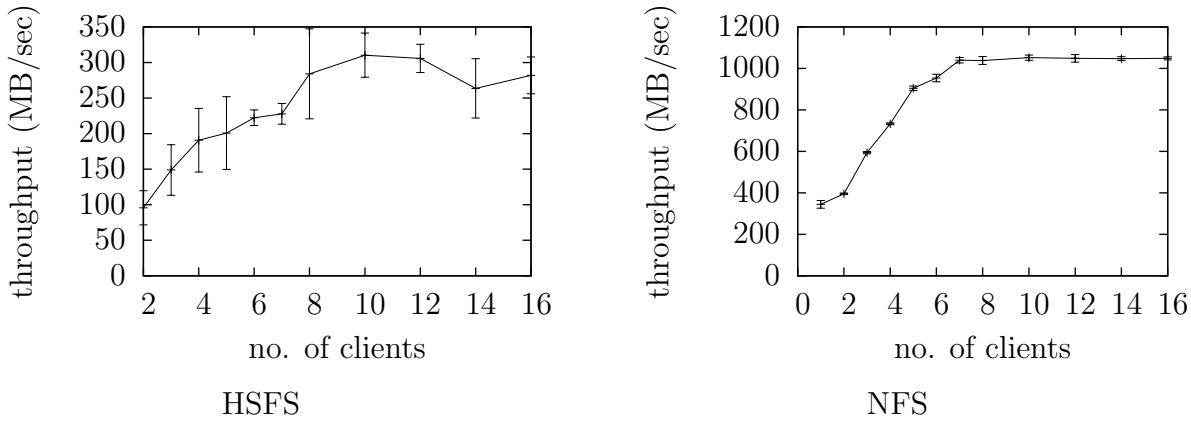

図 7: 多クライアントによる読み込み

```
}
close(fd);
```

なお, 測定の際はクライアントのキャッシュにファイルがある状態での測定を避けるため, あるノードでファイルを作りその直後に別のノードで読むようにします.

やはり結果は頷けるもので, NFS はサーバディスクの速度である 400~500MB/sec 程度になり, HSFS では書き込み時と同様の速度 (70MB/sec 程度) になります. ブロックサイズによらずだいたい一定の性能が出ているのは, ファイルシステムがファイルの逐次読み出しを検知して, 先読み (read ahead) を発行しているからだと思われます.

7.2 多クライアント

図 7 に, ファイルサイズ 256MB でクライアント数を 1 から 16 まで変えた際のスループットを示します. やはりブロックサイズは 512KB を使います.

NFS については, 1.2 GB/sec 程度で飽和しており, ディスクからの読み出し速度 (370 MB/sec) をはるかに上回ります. これは NFS サーバの NIC (10 GigE) の性能です. ディスクの性能を上回っているのは, paramark がベンチマークを行う際, ファイルを作成した直後にそれらを読み出すため, ファイルがサーバのキャッシュにある状態で測定が行われているからだと考えられます. HSFS についても同様で

すが, 飽和した際の性能を決めているのは, NIC ではなく, ネットワークのボトルネック (約 2 Gbps = 250 MB/sec) です.

なお, あるクライアントが作ったファイルは, 別のクライアントが読み出すようにしています.

8 まとめ

HA8000 上のファイルシステムに関して, 性能を理解するための情報と, 実験を通した確認を行いました. 最後に, 知っておくと良いと思われる事をまとめます.

- メタデータ操作の性能は NFS に比べ HSFS が圧倒的に遅い. 特にファイルの作成には 80ms 程度の時間がかかる. ファイルを作ったら, このオーバーヘッドが十分小さくなる程度の書き込みを行うように, アプリケーションを書く必要があります. 対話的操作や小さなファイルの IO は今日から NFS で行うようにすると, 快適になります.

- NFS の read/write の性能は, 1 サーバしかない事によるボトルネックがあり, どれだけのクライアントを用いても write 180 MB/sec, read 370 MB/sec 程度が限界です. しかしその転送レートを出すのに何も工夫はいりません (1 クライアントでも, 少なめのブロックサイズでも自然に出ます). 逆に言うと, 並列に IO を行うことの性能上の利点はあまりないどころか, 並列度を上げると高負荷になり, スループットが低下します.

- HSFS の read/write の性能は, 現状クライアント NIC の限界 (1 Gbps ≈ 120MB/sec) があります. 実測値は 60∼70MB/sec 程度です. クライアント数を増やすと, クライアントがどう割り当てられるかにより, ネットワークのボトルネック (16 ノードで共有された 2 or 3 Gbps のリンク) に部分的に律速されます. 運が良ければ IO サーバのディスクバンド幅 × IO サーバの台数までスケールしますが, ジョブの割り当てにも左右されます. 今回の実験ではそこまでの性能は確認出来ていません.

- HSFS と NFS を全体で比較すると, メタデータ操作のみならず read/write の転送レートという面に置いても, HSFS が NFS を上回る状況は, 現状では少ない事がわかります. ただし多数のユーザが同時にファイルシステムを利用すると, NFS では 1 台のサーバへ全ユーザがアクセスするのに対し, HSFS ではそれが 12 台のサーバへ分散されるため, 高負荷でもより安定したバンド幅が得られることはあり得ます.

- HSFS の read/write 性能は, ブロックサイズ 8KB 程度では得られず, 512KB 程度まで増やすことが推奨されます. gcc のストリームライブラリや g++の標準ライブラリは小さいバッファを用いるので注意が必要です. gcc で HSFS をアクセスする場合 64KB (NFS の場合 4KB だがそれでも問題はない), g++で HSFS をアクセスする場合 8KB となるので, g++の場合に特に注意が必要です. これらは 6.2 節で述べた手法でバッファサイズを変えれば, 回避できます.

- HSFS へのアクセスに使われるネットワークが 2010 年 4 月より, Myrinet に変更されるため, この後は状況が多少変わり, 少ないクライアントで高いバンド幅が得られるようになると期待されます. その際は IO サーバのディスク (write 180 MB/sec ×12, read 370 MB/sec ×12) が律速となり, 負荷を IO サーバに分散させれば, より多くの状況で write 2 GB/sec, read 4 GB/sec 程度のバンド幅を得ることが可能と期待されます. 一方, HSFS メタデータ操作の性能がネットワークのバンド幅に起因するものとは考え難いため, メタデータ操作の性能はあまり改善しないでしょう. 逆に

言うと, open+close の時間が相対的に目立つことになり, それを隠すのに必要なファイルサイズは増えることになります.

- そして, 同時期に導入される Lustre を用いれば, NFS に近い逐次メタデータ性能, NFS を上回るメタデータスループット, NFS/HSFS を上回る read/write 性能を得ることが出来る予定です.

Lustre については安定性や, ソフトウェアのバグによるデータのロスなどが今もまれに発生しているという懸念もあり, 実際に運用をしているサイトでも安定稼働までに時間がかかったという話も聞きます. 我々は導入後, できるだけ多く知見を集積し安定運用に取り組んで行くとともに, なるべく多くの情報をユーザに提供していく所存です.

本稿がファイルシステムの性能理解, ひいては HA8000 環境の有効・快適な利用の一助となれば幸いです.

謝辞

Paramark ベンチマークを開発した頓南氏に感謝致します.

参考文献

[1] bonnie++. http://www.coker.com.au/bonnie++/.

[2] IBM General Parallel File System. http://www-03.ibm.com/systems/clusters/software/gpfs/index.html.

[3] IOR HPC Benchmark. http://ior-sio.sourceforge.net/.

[4] IOZone Filesystem Benchmark. http://www.iozone.org/.

[5] Lustre. http://wiki.lustre.org/.

[6] Richard McDougall, Joshua Crase, and Shawn Debnath. Filebench: File System Benchmark. http://hub.opensolaris.org/bin/view/Community+Group+performance/filebench.

[7] paramark High Fidelity Parallel File System Benchmark. http://code.google.com/p/paramark/.

[8] 東京大学情報基盤センタースーパーコンピューティング部門. HA8000 クラスタシステム FAQ. http://www.cc.u-tokyo.ac.jp/service/faq/ha8000_faq.html.

[9] 東京大学情報基盤センタースーパーコンピューティング部門. HA8000 クラスタシステム利用の手引き. http://www.cc.u-tokyo.ac.jp/service/ha8000/ha8000-tebiki/.