2.2.2. MPI 性能検証

富士通株式会社　　志田直之

ここでは，Open MPI および富士通 MPI を用いて，MPI 性能の評価結果について報告する。

1. 性能評価のポイント

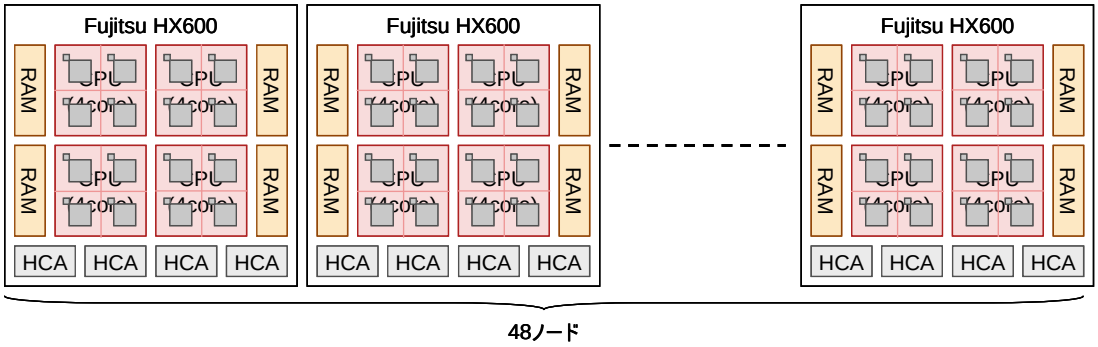
MPI の性能評価は，大きく 3 つに分けて評価を行った。

- プロセス数増加に向けた検証
- ノード内通信とノード間通信の検証
- MPI-IO 性能検証
  - 連続データ転送
  - ストライド転送

2. プロセス数増加に向けた検証

評価に用いたシステムを以下に示す。

CPU	Quad-Core AMD Opteron™ Processor 8354 2.2GHz 4CPU (16core) / node
RAM	16GB /node
Interconnect	ConnectX DDR HCA * 4
OS	Linux version 2.6.18-92.1.22.el5
Topology	FAT Tree



今回の評価に用いたコードと MPI を以下に示す。

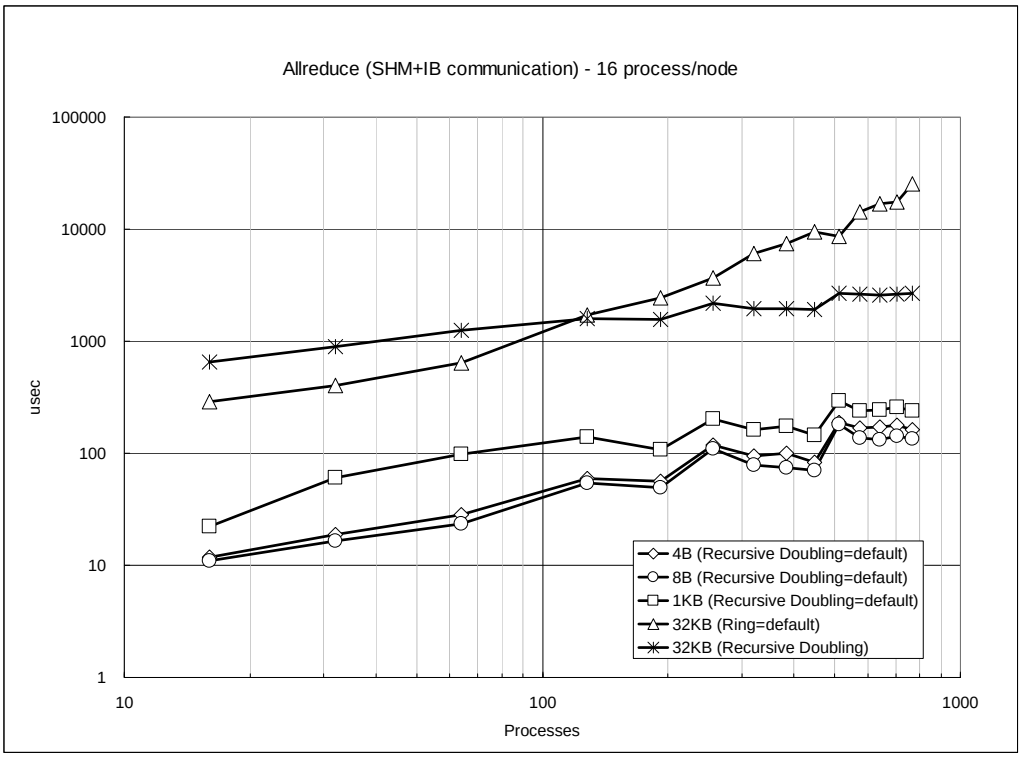
コード	Intel® MPI Benchmarks 3.1
MPI	Open MPI 1.3.3

評価は以下の方法で行った。

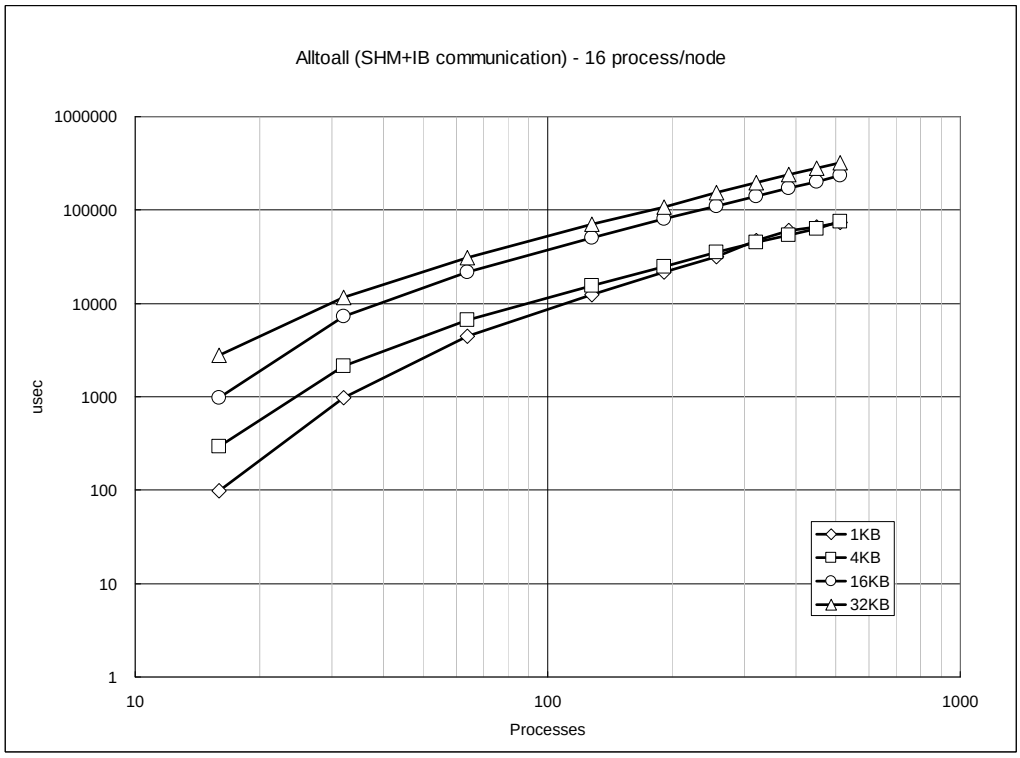
- ノード内のコアを全て使用（1 ノード 16 プロセス× 1 ノード）
- ノード内は共有メモリ通信，ノード間 InfiniBand 通信
- 現状のユーザーアプリケーションを想定し，測定関数と測定範囲を以下のように設定した
  - Allreduce (16～768 プロセス)：4B, 8B, 1KB, 32KB
  - Alltoall (16～512 プロセス)：1KB, 4KB, 16KB, 32KB

検証結果を以下に示す。

- Allreduce (16 プロセス/ 1 ノード)



- Alltoall (16 プロセス/ 1 ノード)



まとめ

- Allreduce

  – 128 プロセス・256 プロセス・512 プロセスの性能値が悪くなるが，相対的にプロセス数とともに実行時間が増えていく。

  ⇒ 一般的に使用される 4 バイトや 8 バイトの Allreduce は，並列度を上げていくと実行コストが見えてくる。このデータ長は一般的にチェックサムに使用されるため，プロセス数増加しても通信量を減らすことができない。このため，スケーラビリティ低下の要因の一つとなる。

  – 32KB では Ring アルゴリズムが選択されるが，実際は 128 プロセスまでは，Recursive Doubling のアルゴリズムの方が効果的である。

- Alltoall
  – プロセス数にスケールして，確実に実行時間が増えていく。
  ⇒ Alltoall を使用したプログラムのスケーラビリティを確保するには，メッセージ長を短くすることが重要となる。

3. ノード間通信とノード内通信

評価に用いたシステムを以下に示す。

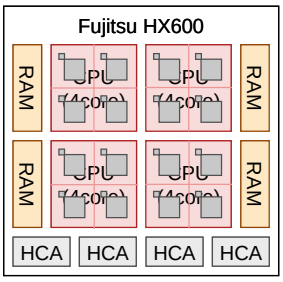
CPU	Quad-Core AMD Opteron™ Processor 8354 2.2GHz 4CPU (16core) / node
RAM	16GB /node
Interconnect	ConnectX DDR HCA * 4
OS	Linux version 2.6.18-92.1.22.el5
Topology	FAT Tree

今回の評価に用いたコードと MPI を以下に示す。

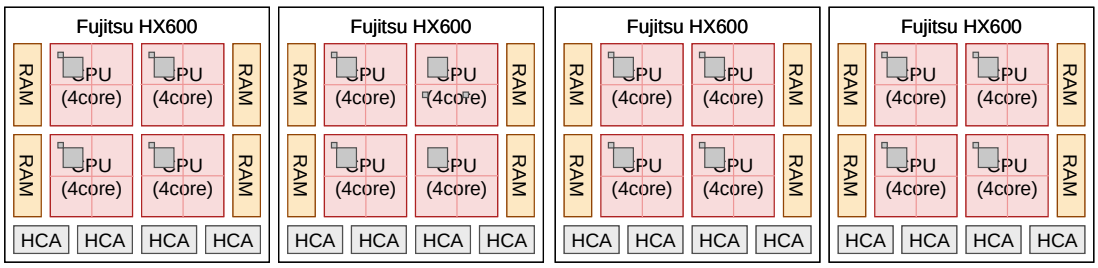
	IB + SHM 通信	IB 通信
実行時 オプション	--mca btl_openib_warn_default_gid_prefix 0 --mca btl self,sm,openib	--mca btl_openib_warn_default_gid_prefix 0 --mca btl self,openib
実行時付加	numactl --interleave=all	numactl --interleave=all

テストパターンと評価システム上のプロセス構成を以下に示す。

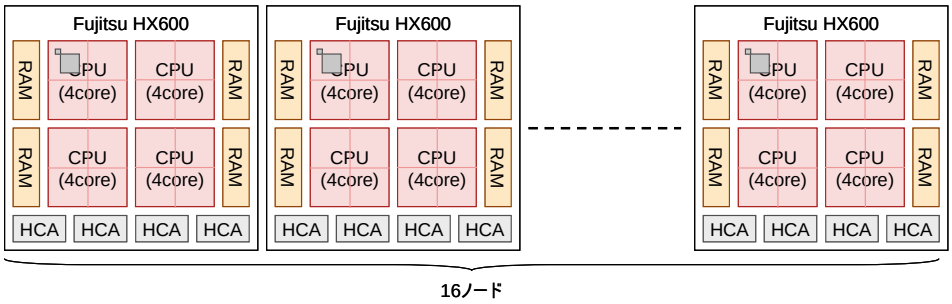
- ノード内のコアを全部使用（1 ノード 16 プロセス× 1 ノード）
  (1) **全ての通信を SHM 通信**
  (2) **全ての通信を IB Loopback 通信**



- ノード内の CPU を 1 コアだけ全部使用（4 プロセス/ 1 ノード× 4 ノード）
  (3) **ノード内は SHM 通信，ノード間は IB 通信**
  (4) **ノード内は IB Loopback 通信，ノード間は IB 通信**

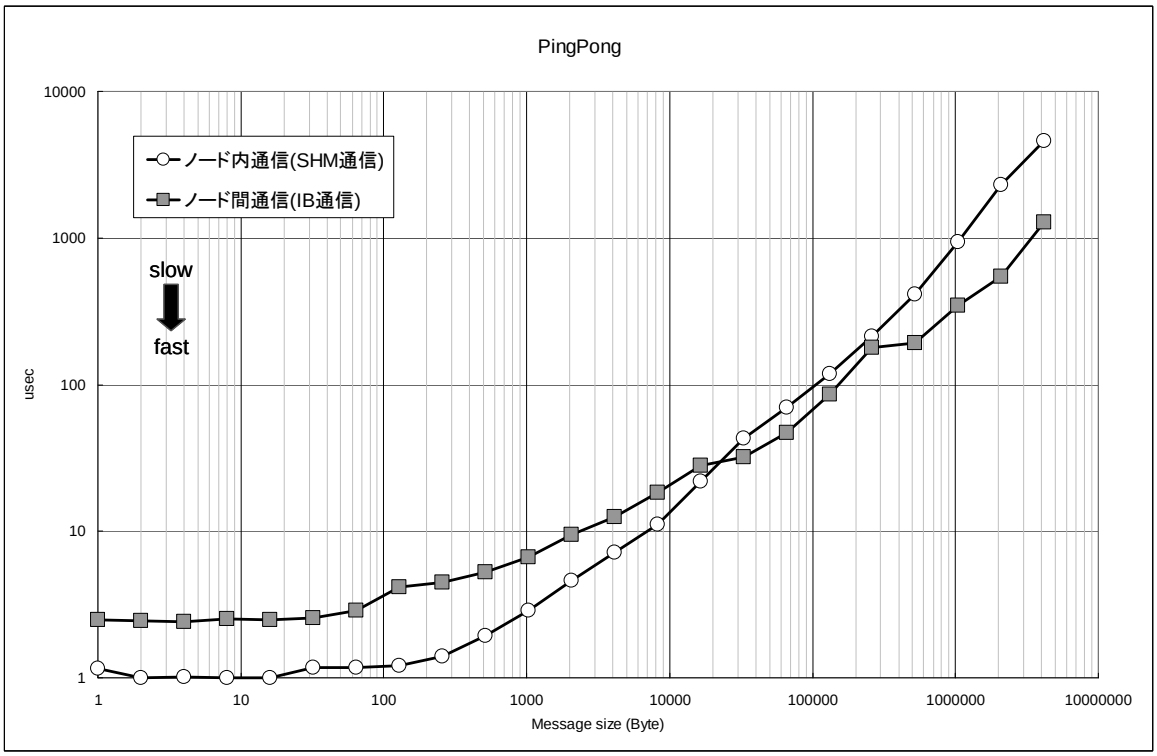


- ノード内の CPU を 1 つだけを使用（1 プロセス/ 1 ノード× 16 ノード）
  (5) **全ての通信を IB 通信**

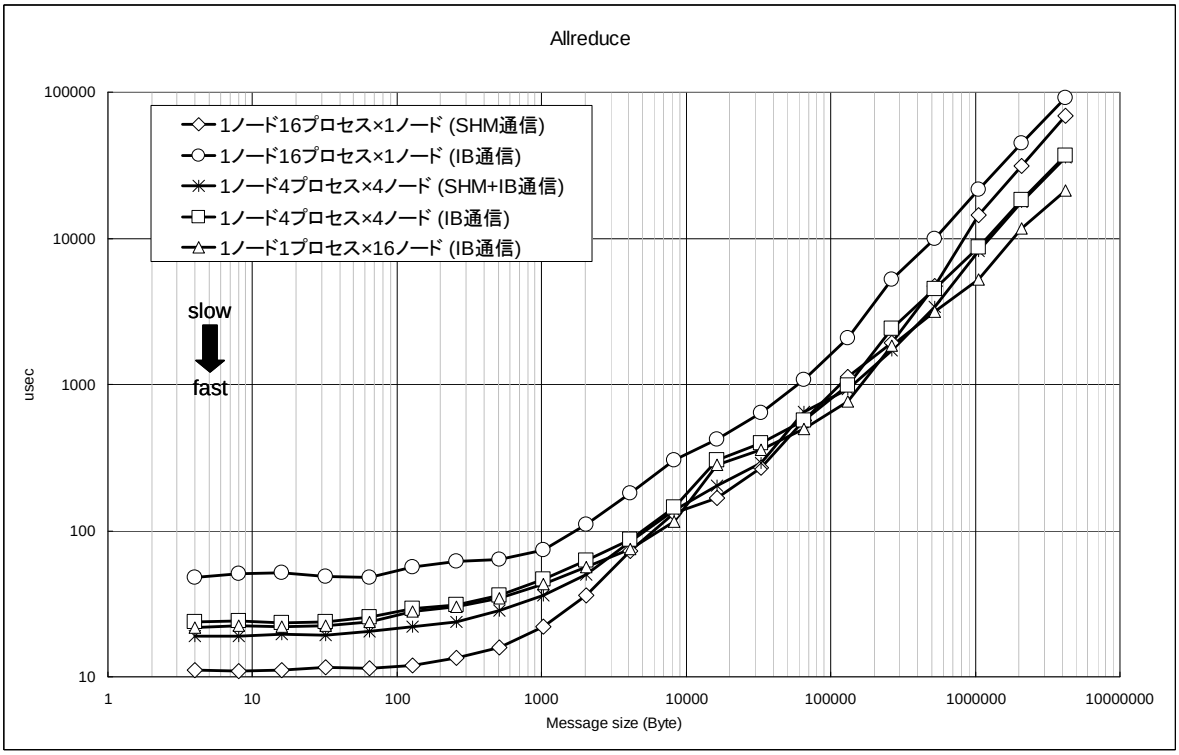


検証結果を以下に示す。

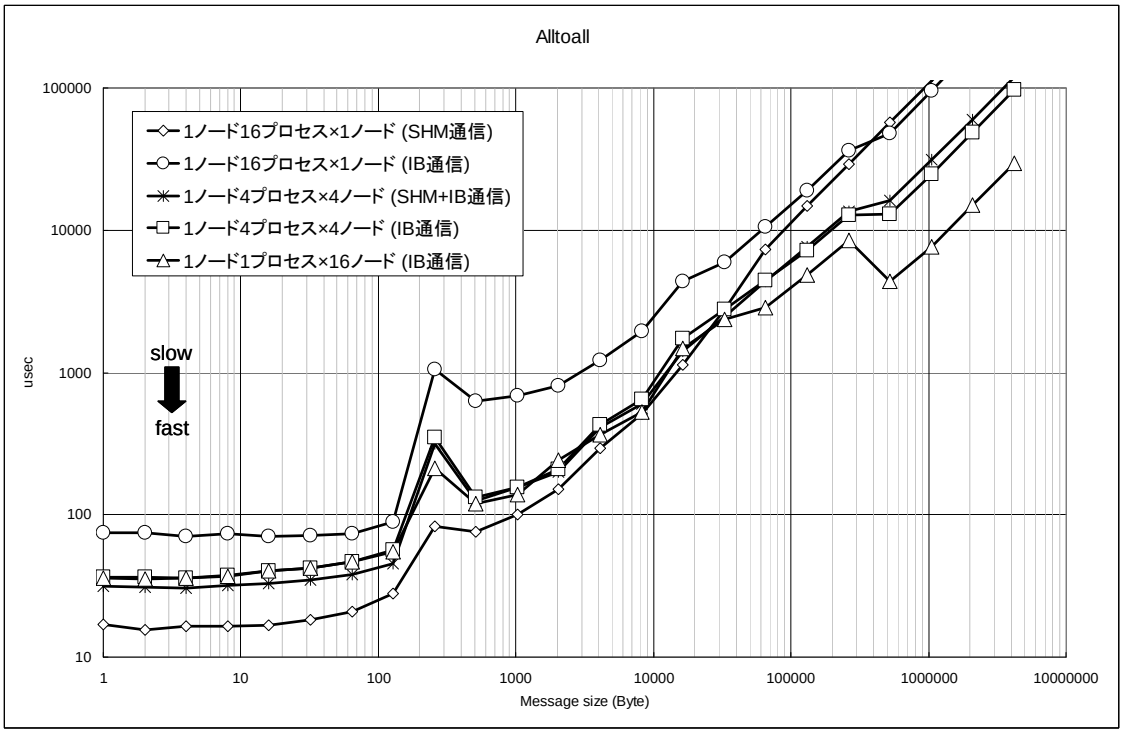
- ノード内通信とノード間通信の比較（PingPong）



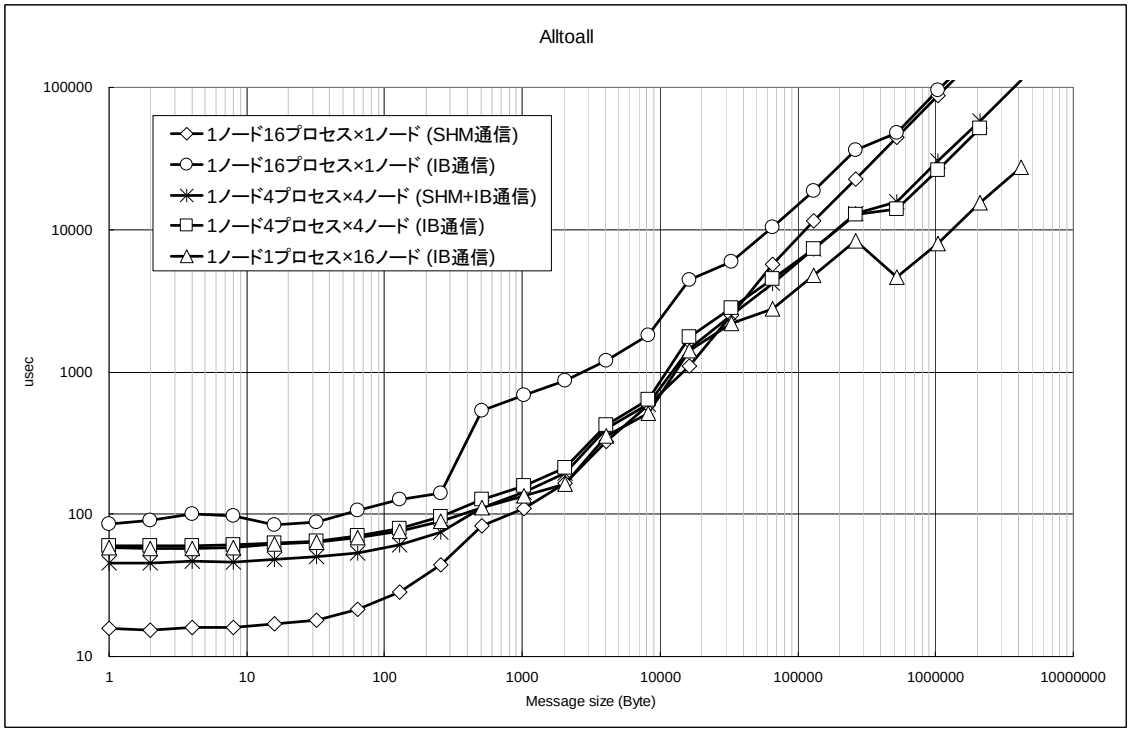
- ノード内通信とノード間通信の比較（Allreduce）



- ノード内通信とノード間通信の比較（Alltoall）



- ノード内通信とノード間通信の比較（Alltoall<一部アルゴリズム変更版>）



まとめ

- 転送長が短い通信では共有メモリ通信が有利となる。本システム(富士通 HX600)では 20KB がしきい値となり, 20KB 以下では共有メモリ通信, 20KB 以上がノード間通信の方が速くなる。
- 必要なメモリやメモリバンド幅が気にならないアプリケーションの場合, アプリケーションの選択は以下のように考える。

	メッセージ長	
	全体的に短い	全体的に長い
ノード内プロセス	ノード内に多くのプロセスを詰め込む	ノード内のプロセスを減らす
ノード内通信	SHM 通信を選択する	IB Loopback 通信を選択する

4. MPI-IO 性能

評価に用いたシステムを以下に示す。

- 計算ノード（富士通 FX1）

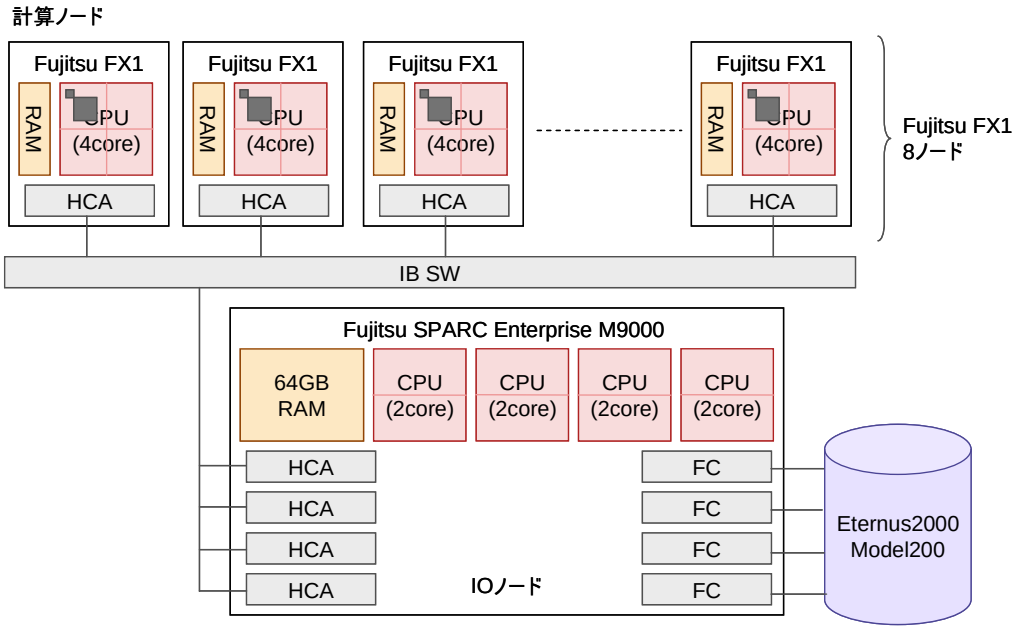
CPU	SPARC64 VII 2.5GHz * 1 / node (4-cores)
RAM	32GB /node
Interconnect	InfiniBand DDR * 1
OS	OpenSolaris Build 79
Middleware	Parallelnavi Base Package 3.1
File system	FUJITSU Parallelnavi SRFS 3.0.0-03(B30000-02G)

- IO ノード（富士通 SPARC Enterprise M9000）

CPU	SPARC64 VI * 4
RAM	64GB /node
Interconnect	InfiniBand DDR * 4
Fibre Channel	4G-FC * 4
File system	Sun StorageTek™ QFS

- ディスクシステム（富士通 ETERNUS2000 Model200）

コントローラ数	2
キャッシュ容量	4GB
Interconnect	InfiniBand DDR * 4
RAID	RAID6(NL-SAS 4D+2P) *
実 WRITE 性能	約 400MB/s



富士通の Parallelnavi Language Package V3 に含まれる MPI を使用した。1 ノードあたり 1 プロセスを生成し，以下の観点でテストを行った。

- 連続データ転送性能
- ストライド転送性能

計測パターンは 2 種類を用意し，以下のように定義する。

- IO キャッシュ性能： IO サーバーへデータを転送した時間とする。
- ディスク込み IO 性能： 実際にディスクに書き込まれるまでの時間とする。

以下に，連続データの転送性能について評価する。
連続データの転送性能は，以下の API 間の消費時間を計測した。

呼び出すAPI一覧

MPI-IO	Split Files	IOマスター	
MPI_File_open	open	open	
MPI_File_set_view		MPI_Gather	IOキャッシュ性能
MPI_File_write	write	write	
MPI_File_sync	fsync	fsync	ディスク込みIO性能
MPI_File_close	close	close	

		I/O 方式		
		MPI-IO	Split Files	IO マスター
ディスク込み IO 性能	IO キャッシュ性能	MPI_File_set_view MPI_File_write	write	MPI_Gather write (rank0 のみ)
		MPI_File_sync	fsync	fsync (rank0 のみ)

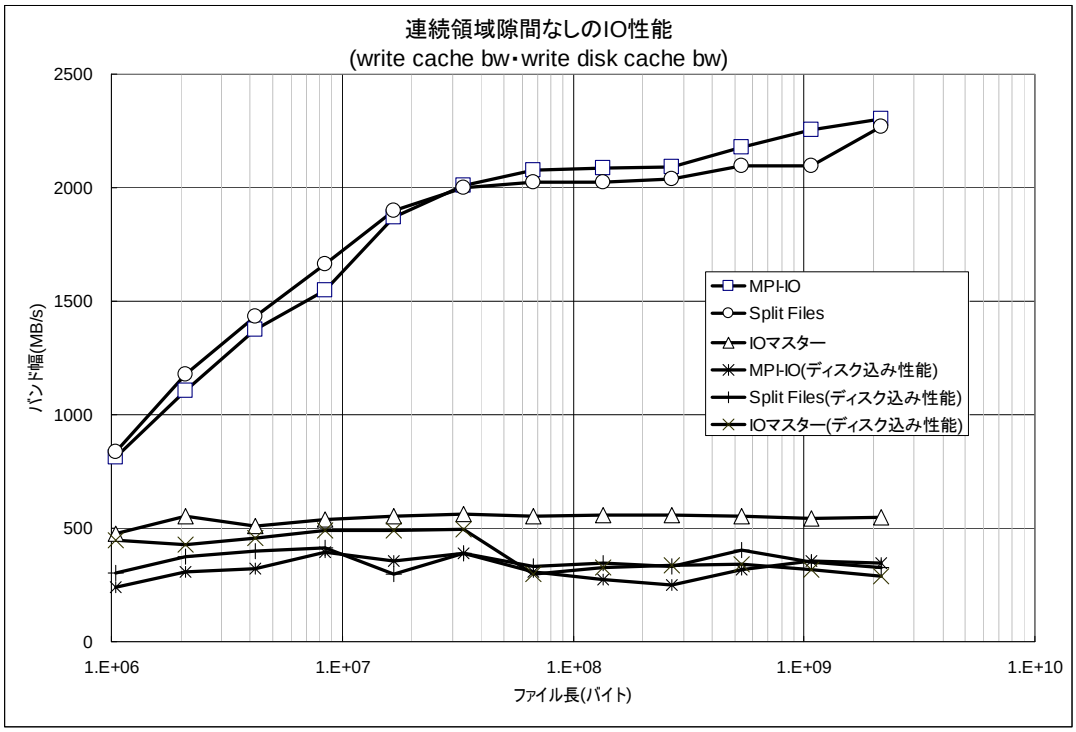
IO 性能は性能ブレが激しく，数回の試行に対する平均値を性能値と定義することが難しい。例えば，IO キャッシュの吐き出しタイミングによって大きく性能ブレが発生するが，ユーザープログラムから見るとランダムに発生するため想定できない。

計測は，1 つの IO 長に対し 20 回試行し，明らかに性能ブレしたデータを目視で排除して集計した。今回提示するデータは，各方式を比較するために「システムが安定しているときの理想的な IO 性能」と考える。

連続データ転送性能は，以下の 3 つのパターンについて計測を行った。

Split Files 方式	各プロセスがファイルを作成する。
MPI_IO 方式	MPI-IO のブロッキング入出力を用いて，一つのファイルを作成する。
IO マスター方式	MPI_Gather によって，全プロセスのデータをルートプロセスに集めてから，一つのファイルを作成する。

検証結果を以下に示す。



MPI-IO 性能と Split Files 性能はほとんど変わらずに，ファイル長が長くなるにつれて傾きは悪くなるがスケールしていく。IO マスター方式は，500MB/s 程度で一定となっている。これは，データの再構成に時間がかかるため，性能面で劣ると言える。

今回評価したシステムでは，ディスク性能が低いため，ディスク込み性能は全ての方式でほぼ同じ性能になっている。MPI-IO 方式・Split Files 方式ともに，IO キャッシュ効果を有効に利用していると言える。

以下に，ストライドデータの転送性能について評価する。

ストライドデータ転送性能は，以下の 3 つのパターンについて計測を行った。

MPI-IO 方式	MPI-IO のブロッキング入出力を用いて，一つのファイルを作成する。
MPI_IO(COLL)方式	集団的 MPI-IO を用いて，一つのファイルを作成する。
IO マスター方式	MPI_Gather によって，全プロセスのデータをルートプロセスに集めてから，一つのファイルを作成する。

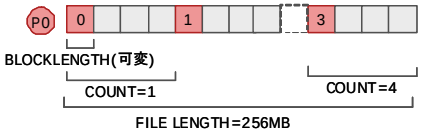
計測は，IO キャッシュ性能を測定し，以下の API 間の消費時間を計測した。

呼び出すAPI一覧

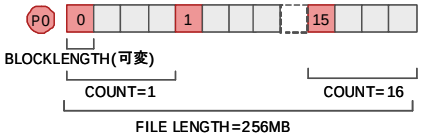
MPI-IO	MPI-IO(COLL)	IOマスター	
MPI_File_open	MPI_File_open	open	
MPI_File_set_view MPI_File_write	MPI_File_set_view MPI_File_write_all	MPI_Gather write	IOキャッシュ性能
MPI_File_sync MPI_File_close	MPI_File_sync MPI_File_close	fsync close	

【ストライド転送性能検証パターン】

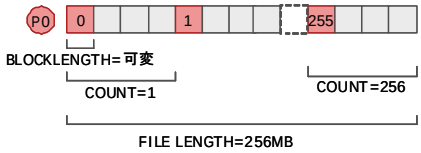
1. ファイル長256MB、ブロック粒度(ファイル長/繰り返し数/プロセス数)、繰り返し数 (4回)



2. ファイル長256MB、ブロック粒度(ファイル長/繰り返し数/プロセス数)、繰り返し数(16回)

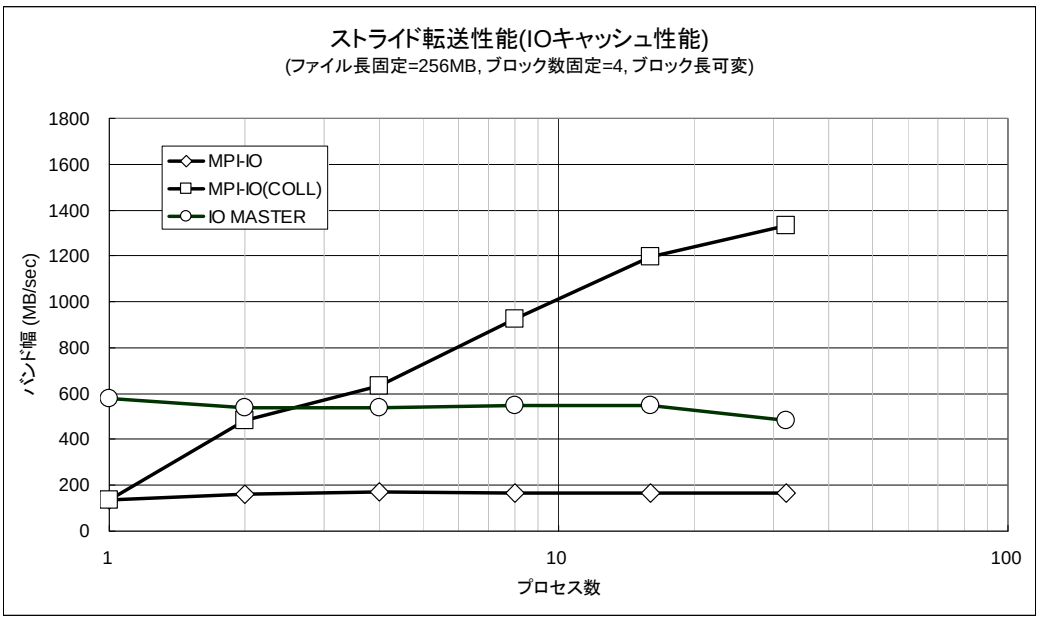


3. ファイル長256MB、ブロック粒度(ファイル長/繰り返し数/プロセス数)、繰り返し数(256回)

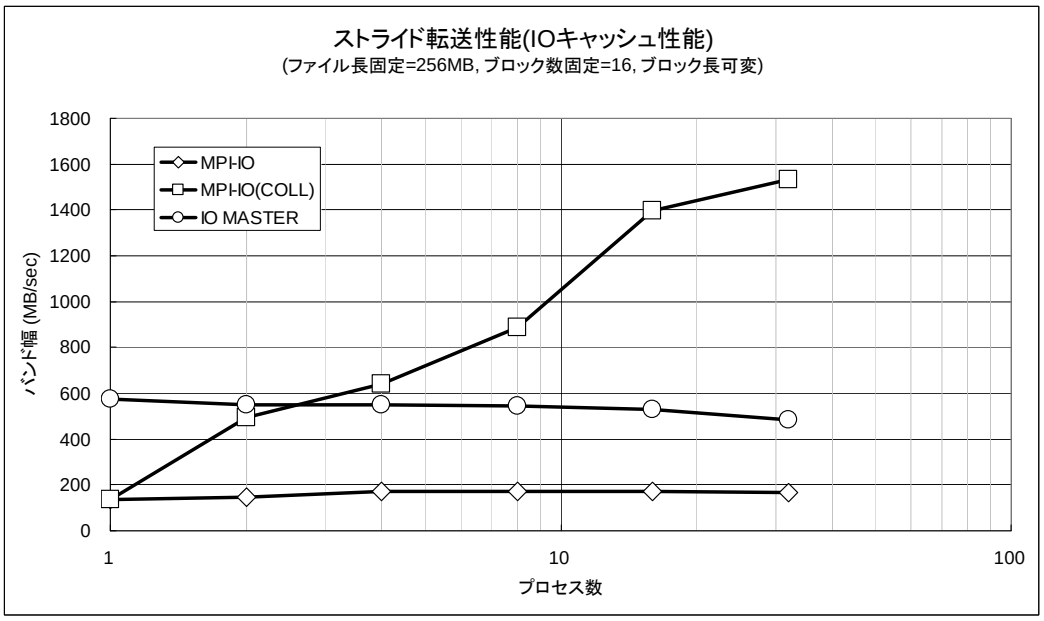


検証結果を以下に示す。

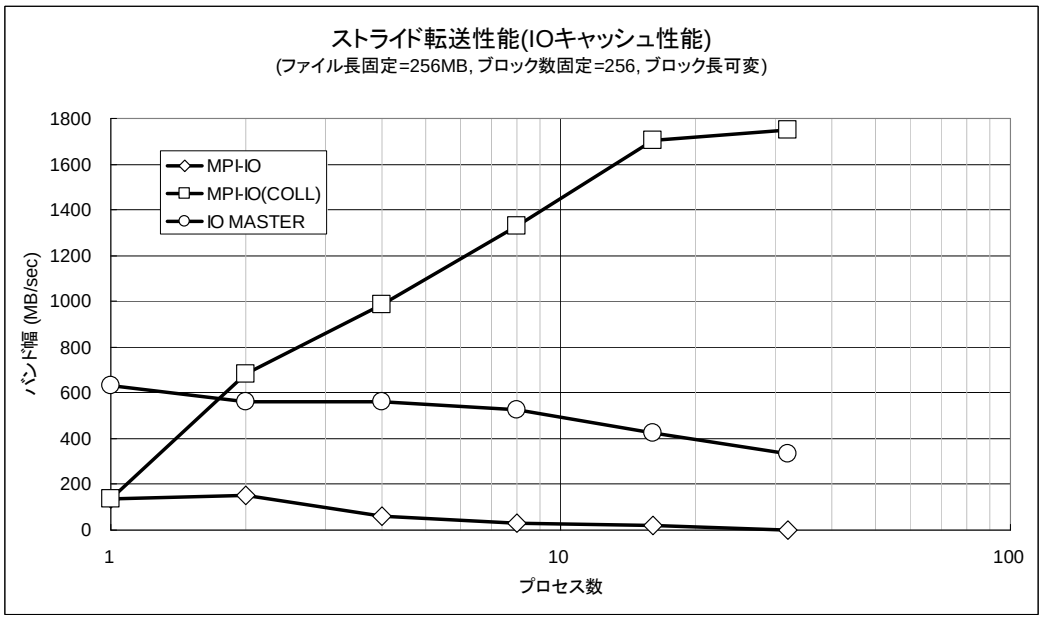
＜ストライド転送性能（ブロック数=4）＞



＜ストライド転送性能（ブロック数=16）＞

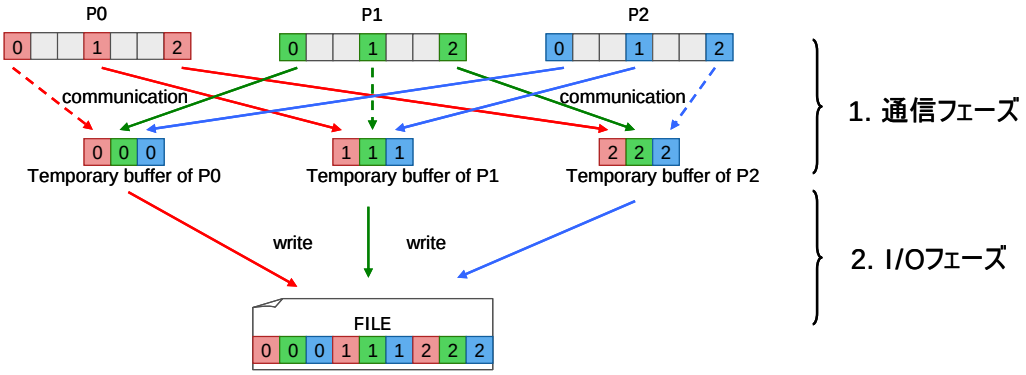


＜ストライド転送性能（ブロック数=256）＞



ストライドデータのファイル転送では，集団的 MPI-IO の性能が効果的である。16 プロセス実行の比較的小さな環境ではあるが，転送性能はスケールしている。これは，ROMIO における集団的 MPI-IO の実装に，Data Sieving の技術が利用されていることが効果的に働いていると考えられる。Data Sieving とは，ストライドデータの入出力処理において，各プロセス内にバッファリングを行い，連続データでファイルを入出力する方法である。2 つのフェーズにわかれていて，出力の場合は通信フェーズから I/O フェーズ，入力の場合は IO フェーズから出力フェーズの順に処理を行う。Data Sieving を利用することで，一つ一つの小さなブロックに対するファイル要求ではなく，大きなブロックに対するファイル要求が行われるため，ブロック数が多い場合に効果的な性能が期待できる。

Data Sieving のファイル出力処理を以下に示す。



逆に，非集団的 MPI-IO の性能は，ブロック数を増やすにつれて性能の低下が大きい。各プロセスが独立して IO を実行する上で，細かい領域に対してのファイルアクセスに対する排他処理の影響が大きいと考えられる。

以上