Vol.2013-ARC-205 No.1
Vol.2013-OS-125 No.1
2013/4/25

# メニーコア向け NUMA 最適並列分散 I/O の予備検証

小田和友仁[†1]　住元真司[†1]　堀敦史[†2]　石川裕[†3†2]

概要：将来の High Performance Computing (HPC)システムではメニーコア化・NUMA 化が進み，これらを考慮したシステムソフトウェアが重要になる．本稿では NUMA 型のメニーコアシステムにおける並列分散 I/O の 実現に向けた予備検証の結果と考察について述べる．予備検証では，NUMA 環境での性能律速要素と律速時の性能影響を確認するため，既存の NUMA 環境でキャッシュ I/O における read/write の I/O 性能を測定した．本稿では予備検証で得た結果をもとに，将来 HPC での最適化された並列分散 I/O を実現する上での最適な分散ポリシーについて議論する．

# Preliminary Validation of NUMA aware Parallelized I/O for Manycore

TOMOHITO OTAWA[†1]　SUMIMOTO SHINJI[†1]
ATSUSHI HORI[†2]　YUTAKA ISHIKAWA[†3†2]

Abstract: Future High Performance Computing (HPC) systems will have more manycore processors with NUMA. This paper shows results of preliminary validation of NUMA aware Parallelized I/O for Manycore and delivers consideration for it. In the preliminary validation, we measured I/O bandwidth of read /write using cache I/O on existing NUMA machine to check rate-limiting factors and impact of the effect. This paper discusses I/O distribution policies to optimize parallelized I/O for future HPC system.

## 1. はじめに

スーパーコンピュータのランキング TOP500 では近年スカラ型のシステムが上位を占めている[1]．スカラ型システムでは演算性能と低消費電力の両立のために低クロックかつメニーコア化が進んでおり，その傾向は今後も変わらないと予想される[2].

一方，入出力装置も並列化することで高速化が図られてきた[3][4]．今後，更なるインターコネクトの高速化，入出力装置の高速化が進むと，計算ノード側の I/O 処理がネックとなる可能性も考えられるため，高速化が望まれる．並列演算性能の向上に対して，単コアでの I/O 処理では性能向上は望めずボトルネックとなる可能性が高い．改善策のひとつとして I/O 処理の並列化による性能向上が研究されてきた[5].

将来 HPC では以下の理由から，Non-Uniform Memory Access (NUMA)構成がより複雑になると予想される．

- メニーコア化に伴いコア間，メモリ間のバスが増え，CPU と Memory 間の距離が不均等になる
- キャッシュコヒーレンシ保障のための制御通信がノード間バスに及ぼす影響が増大
- キャッシュコヒーレンシをハードウェアではなくソフトウェアが保証するアーキテクチャも考えられる
- 高速低容量メモリと低速大容量メモリの組み合わせによるメモリ階層化なども研究されており，データの局所性がより重視される

NUMA 構成のマシンでは CPU とメモリ間の距離が均等ではなく，近距離と遠距離のメモリアクセスレイテンシに差が生じる．そのため並列演算では各スレッドのデータ局所性を高め，なるべく近いメモリに配置するような最適が図られてきた．将来 HPC での並列分散 I/O においても，NUMA を意識した最適が必要になる．

本研究では将来 HPC での並列分散 I/O の課題として，NUMA 最適な分散アルゴリズムを検討していく．本検証ではその第一段階として，NUMA 環境での性能律速要素と律速時の性能影響を確認するため，既存の NUMA 環境でキャッシュ read/write 性能を測定した．

本稿では予備検証で得た結果をもとに，メニーコア向けの並列分散 I/O を実現する上での最適な分散ポリシーについての考察結果を述べる．2 章で本研究が実現しようとする並列分散 I/O について述べ，3 章でそのための課題を述べ，課題に対する解決策検討の予備検証として，既存 NUMA 環境で課題の影響を検証することを述べる．4 章で検証用の並列分散 I/O 実装を，5 章でベンチマーク実装の詳細を述べ，6 章で使用する NUMA 環境を紹介，7 章と 8 章で各検証方法と検証結果を述べる．

## 2. 並列分散 I/O

将来の HPC システムでは，メニーコア資源を最大利用しアプリの使用可能メモリ量を増やすため，スレッド並列，ハイブリッド並列（プロセス並列とスレッド並列の組み合わせ）が主流になる．スレッド並列ではデータ読み込み後

†1 富士通株式会社
Fujitsu Limited.
†2 理化学研究所計算科学研究機構
RIKEN AICS
†3 東京大学
University of Tokyo

に,“並列演算と同期,ファイルへの結果出力“を繰り返す．HPC 向けの大規模演算では演算結果も膨大となる傾向にあり，結果出力では単一の巨大な I/O 要求が発生する．この I/O 処理がボトルネックになると全体の演算効率は低下してしまう．

HPC システムでは，一般的に計算ノードは高速ネットワークを介してストレージシステムと接続される．よって計算ノードの演算プロセスの write システムコールは，以下の手順で処理される．

(1) 計算ノードのカーネルがユーザ空間のユーザバッファからカーネル空間のキャッシュへコピーし，システムコールから復帰

(2) 計算ノードのファイルシステムクライアントがネットワークを介してキャッシュをストレージサーバに転送

(3) ストレージサーバがストレージに書き出し

上記の(1)の処理に対し，(2)，(3)は非同期に処理されるため，(1)の完了後はユーザの演算処理に復帰可能である．よって(1)の高速化が演算処理にとっての I/O 処理の応答性の向上に繋がる．また(2)，(3)の処理は(1)の完了に依存するため，(1)の処理が(2)，(3)の処理よりも遅いと，処理待ちが生じボトルネックとなってしまう．現状の HPC システムではインターコネクト経由でのストレージ書き込み性能に比べてメモリアクセス性能は十分に高いため問題とならないかもしれない．しかし将来的にインターコネクト性能やストレージ性能が向上した場合に，表面化する可能性がある．そこで本研究では(1)のキャッシュ I/O の高速化のため，並列処理化を図る．

図 1 にスレッド並列での単一 I/O 要求に対する一般的なカーネルでの逐次 I/O 処理のイメージを示す．例えば演算処理を OpenMP でスレッド並列化し各ユーザスレッドが配列 a の担当するオフセット位置に結果を書き込む場合を仮定する．ユーザはメモリ書き込み速度を最適にするよう，各ユーザスレッドが使用するメモリはそのユーザスレッドが動作するコアと同じ NUMA ノード（ローカル）に配置するのが一般的である．並列演算終了後にスレッドマスタが配列 a のファイル書き込み要求を実施する．この要求に対し，一般的なカーネルの共通 I/O 処理では 1 コアでキャッシュに書き出そうとする．そのため 1 コアのコピー性能に律速してしまうという問題がある．また NUMA システムでは，他の NUMA ノードをまたぐメモリアクセスが発生し，レイテンシに影響するという問題も発生する．

本研究ではこのようなスレッド並列演算からの単一の I/O 要求に対して,並列にキャッシュ I/O を実施することで高速化を図る．図 2 にスレッド並列での単一 I/O 要求に対する並列 I/O 処理のイメージを示す．配列 a のファイル書き込み要求を複数の I/O スレッドで分担し書き出す．このとき各 I/O スレッドがローカルの領域を担当し，ローカル

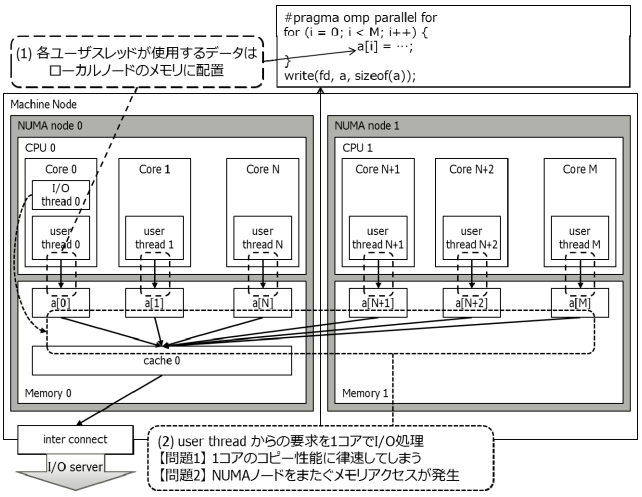


図 1 スレッド並列での逐次 I/O 処理イメージ

Figure 1 Image of serial I/O processing for parallel threads

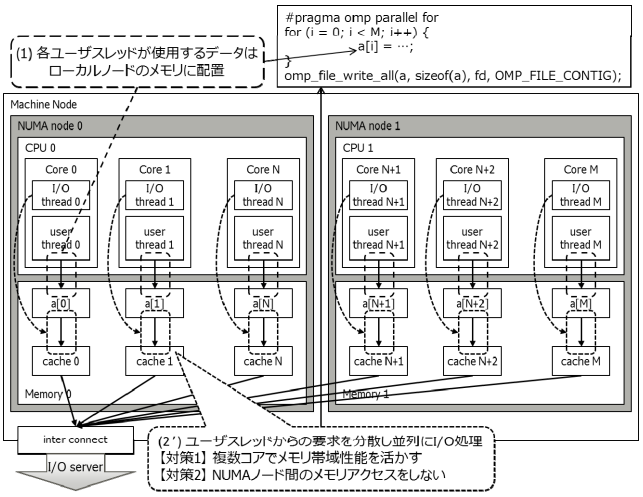


図 2 スレッド並列での並列 I/O 処理イメージ

Figure 2 Image of parallel I/O processing for parallel threads

のキャッシュに書き出すようにすることで, NUMA ノード間のデータ移動を最小にする．これにより全コアの処理性能を活かし，各 NUMA ノードのメモリ帯域を最大限に活用することが可能となる．

## 3. 並列分散 I/O の課題と検証

本研究では将来 HPC で課題となる NUMA 影響を最小限に抑えるための分散方式を検討する．並列分散 I/O の完了は全 I/O スレッド処理が完了した時点であり, I/O スレッド毎の処理時間に偏りがあると並列効果は低下する．よって各 I/O スレッドで処理完了時間が均等になるよう分散することが望ましい．NUMA 環境ではユーザバッファと I/O スレッド，キャッシュの NUMA 配置の組み合わせによって転送速度に差が生じるため，分散時にこれらを意識する必要がある．このうちユーザバッファの NUMA ノード配置はユーザが決めるものであり, I/O 処理では制御できない．よってユーザバッファの NUMA ノード配置情報をもとに,

各 I/O スレッドへ分配する仕事量を調整する必要がある．

この場合ユーザバッファが各 NUMA ノードに均等に分散されるか，偏りが生じる（不均等）かにより対処が異なる．

ユーザバッファが各 NUMA ノードに均等に分散する場合（以降，on-each-node）には，NUMA ノード毎にユーザバッファのうちローカルの領域をローカルの I/O スレッド群でサイズ均等に分散し処理することで，NUMA ノード間のメモリアクセス無しに，全 I/O スレッドでの処理時間を均等にすることが期待できる．さらに各 NUMA ノードに分散しているため，各 NUMA ノードのメモリ帯域を最大限に使用することができる．よって並列 I/O で最大の効果を得られる最良のケースといえる．

一方，ユーザバッファが各 NUMA ノードに不均等に配置されている場合には，前記と同じようにローカルに閉じて分散させようとすると，NUMA ノード毎に I/O スレッドの処理時間が不均一になってしまう．よって NUMA ノード間をまたいで負荷分散する必要がある．特にユーザバッファが 1 ノードに配置される場合（以降，on-one-node）が最悪のケースである．ノードを跨いで負荷分散してもユーザバッファの配置されたノードへのメモリアクセスが集中するため，メモリ帯域に律速するか，NUMA ノード間のバス帯域幅に律速する可能性がある．その場合メモリ帯域に律速するならば並列数を上げずに，より近い NUMA ノードの I/O スレッドのみで分散する方が効率的である可能性がある．

以上のようにユーザバッファの NUMA ノード配置の偏りによって，I/O スレッドへの分散の配分は動的に決定する必要がある．その入力値としては，メモリ帯域幅，NUMA ノード間の帯域幅，NUMA ノード間のホップ数によるレイテンシの差などが考えられる．これらはシステムによって異なり，そのバランスによって最適な分散方針は異なる．本来的には将来システムでの性能バランスを元に検討する必要があるが，現状で不定である．よって本研究では，まず既存のシステムを例題として検討する．

分散方式を検討にあたり，まず性能改善の見込み幅と最適な並列数を確認する必要がある．さらに配分決定のための入力情報としてローカルのメモリアクセスと NUMA ノード間をまたぐメモリアクセスの性能差を把握する必要がある．そこで本検証では，まずサイズ均等に分散する並列分散 I/O のプロトタイプを実装し，既存の NUMA 環境を使用しキャッシュに対する write 性能を測定することで課題の影響を検証する．最良ケース（on-each-node）と最悪ケース（on-one-node）での I/O 性能を測定し，スケーラビリティと並列化による改善幅を確認する（検証 1）．次にさらなる NUMA 最適を検討するための基礎性能として，ユーザバッファと I/O スレッド，キャッシュの NUMA ノードの組み合わせ（本稿では経路と呼ぶ）毎の帯域幅を測定する（検証 2）．この結果をもとに，NUMA 最適な分散方針を検討していく．

## 4. 並列分散 I/O の検証用実装

本節では，並列分散 I/O の検証用実装の概要について述べる．

本検証では I/O 要求分散後の並列キャッシュ read/write 性能を重点とする．このため I/O スレッドとキャッシュの NUMA 配置を制御しやすいようにユーザ空間で分散する方式を採用し，並列分散 I/O ライブラリのプロトタイプを作成した．ライブラリは初期化関数と並列分散 I/O 関数，終了関数から構成される．図 3 にベンチマークから各関数を呼ばれた時の処理の流れを示す．

初期化関数は引数に指定された“NUMA ノードのビットマップ”と“ノード毎のスレッド数”に従い，I/O スレッドを作成し，各ノードにバインドする．各スレッドは要求があるまで待機する．

並列分散 I/O 関数は引数に指定された I/O 要求を分割し，各 I/O スレッドに指示する．各 I/O スレッドは指示に従って read/write システムコールを呼び出し，実施する指示がなくなると，再び待機状態に移行する．分散方法として，サイズ均等に分割し各 I/O スレッドに分配するようにした．I/O スレッドへの分配順序は一定にしたため，ベンチマーク側でユーザバッファを用意する際に I/O スレッドが担当する領域のノード配置を順番に合わせて任意に指定することができる．

終了関数は，各スレッドに終了を指示し，スレッド終了を待ち合わせる．各 I/O スレッドは終了指示を受けると，スレッドを終了する．

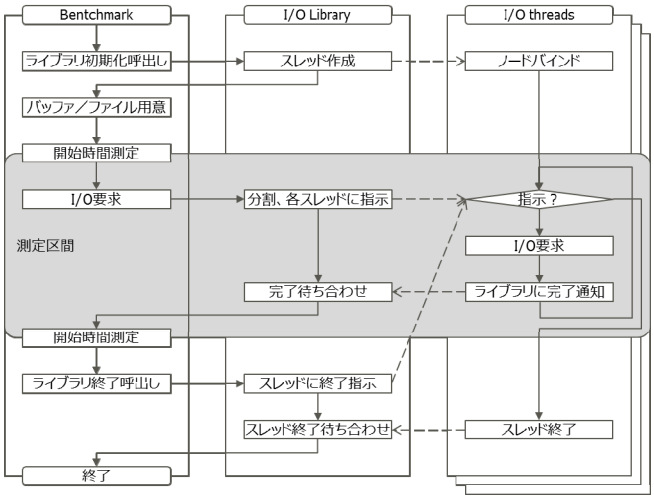


図 3　並列分散 I/O プロトタイプ実装
Figure 3 Parallelization in user space

## 5. ベンチマーク実装

前述の並列 I/O ライブラリを使用するベンチマークを作成した．ベンチマークでは libnuma を使用してユーザバッファ，キャッシュのメモリポリシーを指定し，指定の範囲で並列分散 I/O スレッド数を変更しながら繰り返しファイル I/O 要求に対する応答時間を測定する．

キャッシュ read/write 性能を測定するためファイルは ramfs に配置した．Linux の共通 write ルーチンでは，同一ファイルへの I/O の競合を防ぐためにロック（i_mutex）を獲得してしまう．そのため並列で write を呼び出してもロックにより逐次処理化してしまう．そこで本検証では共通 write ルーチンを流用し，ロックを獲得しないよう改造した write ルーチンを用意した．また Linux の ramfs ファイルシステムを流用し，我々の write ルーチンを呼び出すよう，ファイル操作関数ポインタを差し替えた並列 I/O 用 ramfs を作成しこれを利用した．

本検証では，純粋な I/O 性能の検証を目的としているためページ割り当て他の影響は排除する必要がある．そこでユーザバッファ，ページキャッシュともに，ベンチマークプロセスにて測定のための実行の前に，ウォームアップ実行で全領域にアクセスしておくことでページ割当て済みとした．また以下のオーバーヘッドの影響は最小にする必要がある．

1. ベンチマークからの I/O 指示から各 I/O スレッドでの実行開始までの時間
2. システムコールのオーバーヘッド
3. 最後の I/O スレッド処理完了からベンチマーク処理復帰までの時間

これらのオーバーヘッドのレイテンシを事前に測定した結果，すべて合わせても 1 ミリ秒程度であった．このオーバーヘッド影響を 1%以下に抑えるため，キャッシュ I/O の処理時間が 100 ミリ以上となるように，I/O 要求の総データサイズは 4GiB とした．4GiB としたことで L1～L3 キャッシュのサイズも大きく上回るため，キャッシュメモリの影響も誤差の範囲となる．

## 6. 検証に使用するシステム

本検証では，既存の NUMA 構成マシンとして PRIMERGY RX500S7（以降，RX500）を使用した．スペックを表 1 に，構成図を図 4 に示す．

表 1 測定環境
Table 1 Measurement environment

| | |
|---|---|
| CPU | Intel Xeon E5-4620 2.2 GHz (8 core) x 4 |
| MEM | DDR3-1333 8 GB x 8 |
| Bus | 7.20 GT/s Intel® QPI |
| OS | Cent OS 6.3 |

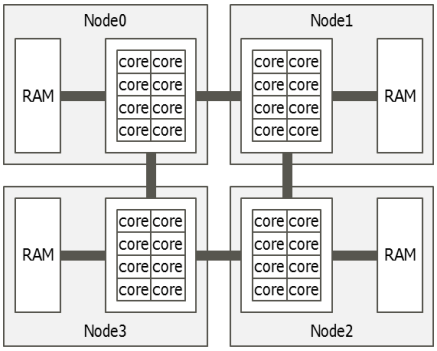


図 4 RX500S7 の NUMA 構成
Figure 4 NUMA topology of RX500

## 7. 検証 1：スケーラビリティ

### 7.1 測定方法

最良ケース（on-each-node）と最悪ケース（on-one-node）で I/O スレッド数を変化させた I/O 性能を測定し，スケーラビリティと並列化による改善幅を確認する．I/O スレッド数は 4～32 までの 4 の倍数とし，逐次処理性能として I/O スレッド数 1 の場合も測定する．I/O スレッド数 4～32 の場合，I/O スレッドは各ノードに均等に分散して配置し固定した．I/O スレッド数 1 の場合は node0 で動作するように固定する．

図 5 に on-each-node の場合のユーザバッファ配置と write 経路を示す. on-each-node では 4GB のユーザバッファを 1GB ずつ各 NUMA ノードに均等に配置し，これらを一つのユーザバッファとして検証用の ramfs 上の 1 ファイルに対して並列 write 要求を実施する．I/O スレッド数 1 の場合，I/O スレッドは各ノードに分散したユーザバッファを逐次的に読み出し，ローカル（この場合 node0）のキャッシュに書き出す．I/O スレッド数 4～32 の場合，各 I/O スレッドはユーザバッファのうちローカルのメモリ上に存在する領域を読み出し，ローカルのキャッシュに書き出す．

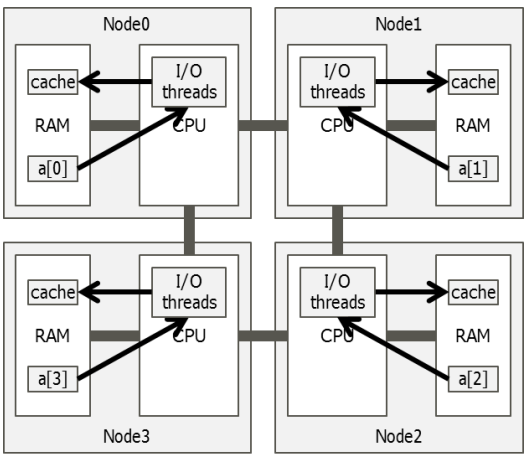


図 5 on-each-node のユーザバッファ配置と write 経路
Figure 5 positions of on-each-node user buffer and route of write data movement

図 6 に on-each-node の場合のユーザバッファ配置と write 経路を示す．on-one-node では 4GB のユーザバッファをまとめて node0 に配置する．I/O スレッド数 1 の場合，I/O スレッドは node0 のユーザバッファを読み出し，ローカル（この場合 node0）のキャッシュに書き出す．I/O スレッド数 4～32 の場合，各 I/O スレッドは node0 のユーザバッファを読み出し，ローカルのキャッシュに書き出す．

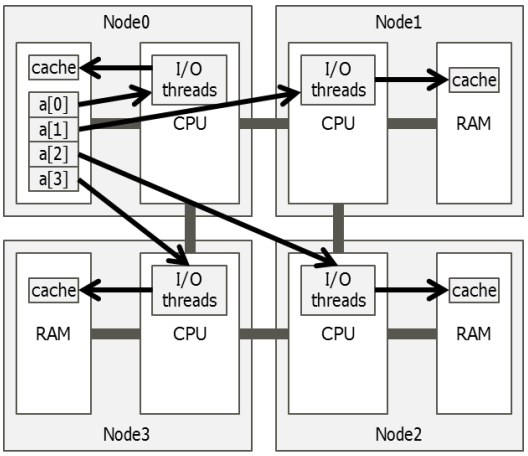


図 6　on-one-node のユーザバッファ配置と write 経路
Figure 6 on-one-node position of user buffer and route of write data movement

### 7.2 測定結果

図 7 に on-each-node と on-one-node でのキャッシュ write の測定結果を示す．

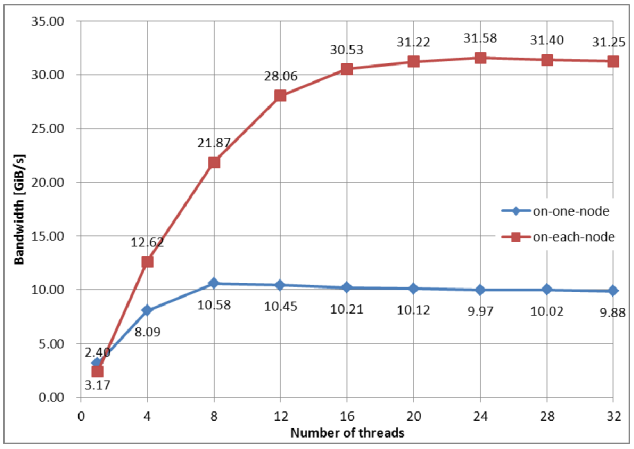


図 7　キャッシュ write 帯域幅
Figure 7 Bandwidth of cache write access

I/O スレッド数 1，すなわち逐次 I/O では，on-one-node の帯域幅が on-each-node を上回った．これは on-one-node がローカルのメモリアクセスに閉じるのに対し，on-each-node では各 NUMA ノードに配置されたユーザバッファを読みだす必要があり，NUMA ノード間転送のオーバーヘッドが発生するためである．

一方，I/O スレッド数 4 以上の並列分散 I/O においては，on-each-node の帯域幅はすべての I/O スレッド数において on-one-node を上回った．

on-each-node では増加に伴い性能も増加するが徐々に傾きは緩やかになり，I/O スレッド数 24 で最高性能（31.58GiB/s）を記録した．それ以上では緩やかに減少し続けた．この律速は NUMA ノード毎のメモリ帯域によるものと考えられる．on-each-node の並列分散 write では各 I/O スレッドでのデータ転送はローカルメモリアクセスに閉じ，QPI への通信が発生しないためである．

on-one-node では I/O スレッド数 8 まではスケールしたが，それ以上では緩やかに減少し続けた．これは node0 からの読み出しにおいてメモリ帯域幅に律速していると考えられる．on-one-node の並列分散 write では，全 I/O スレッドは node0 のユーザバッファから読みだすため，アクセスが集中する．メモリ帯域に律速するのならば，ローカルの I/O スレッドでのキャッシュ書き込みを抑えるため，隣接する I/O スレッドに分散する方が効率的である可能性がある．

以上の結果から，on-each-node と on-one-node ともに，逐次 I/O 処理（I/O スレッド数 1）に対して並列分散 I/O はある程度のノード数まではスケールし，効果があることを確認した．それ以上ではメモリバンド幅に律速してしまうため，割り当てる I/O スレッド数は一定に抑える方が効率的である．RX500 では on-one-node で I/O スレッド数 8，on-each-node で 24 がピークであったので，ユーザバッファの配置されるノード数に比例して I/O スレッド数を 16 前後で増減させるのが効率的であると考える．

on-each-node が最良ケースであり on-one-node の最悪ケースであることから，ユーザバッファが各 NUMA ノードに不均等に配置されている場合は，この間の性能を示すと考えられる．これらの性能を最良ケースの性能に近づけることが，NUNA 最適化の目標となる．

## 8. 検証 2：データ経路に対する NUMA 影響

### 8.1 測定方法

I/O スレッドを Node1 に固定して配置し，ユーザバッファとキャッシュの NUMA ノード配置について，node0～3 の全組み合わせで，キャッシュ read/write の帯域幅を計測する．I/O スレッド数 1 ではメモリ帯域幅，QPI の律速点が見えにくいため，4 スレッド並列で測定する．

### 8.2 測定結果

図 8 に単経路のキャッシュ write の帯域幅測定結果を示す．read の結果は write の結果とほぼ同じ傾向を示したため割愛する．表 2 にデータ転送経路のタイプ分けを示す．図 8 の結果をこのタイプで分けると，RX500 では全体的な傾向として，local タイプ，pull タイプ，push/relay タイプ，

round-trip タイプの順に性能が高かった．local では最高性能を記録した．local の結果はメモリアクセスのみに閉じるため，ノード毎のメモリアクセス性能に依存すると考えられる．1 ノード内の memcpy 性能を実測したところ，同程度の性能に収まることを確認した．pull では local に比べて 6 割程度の性能に抑えられた．リモート NUMA ノードからのメモリ読み込みアクセスがレイテンシを上げる原因である．push や relay，round-trip のようにリモートノードに書き込むパターンがさらに遅くなる傾向にあった．push と relay とで同程度の性能を示すことから，リモートノードからの読み出しよりは，リモートノードへの書き込みが支配的であることが分かった．さらに NUMA ノード間の距離も影響しており，node0 に対してホップ数 2 となる node3 へのアクセスはどのタイプでも遅くなる傾向にあった．

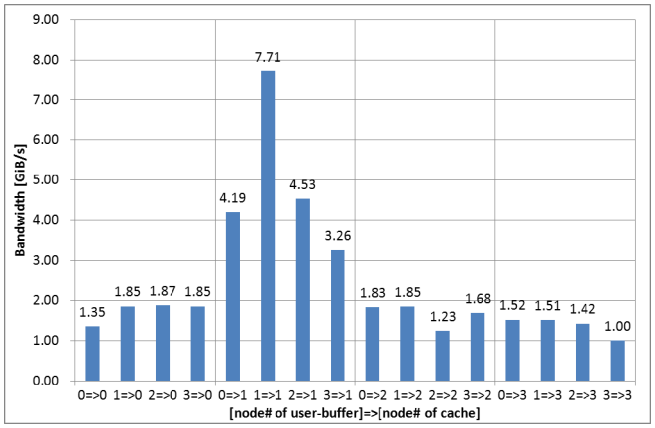


図 8　データ転送経路別のキャッシュ I/O 帯域幅(Write)
Figure 8 Cache I/O Bandwidth on each data movement route(Write)

表 2 データ転送経路タイプ
Table 2 Types of data movement route

| タイプ | 説明 | 例：図 8 |
|---|---|---|
| local- | NUMA ノード内に閉じる経路 | 1=>1 |
| pull | I/O スレッドがリモートノードから読み出しローカルノードに書き込む経路 | 0=>1,<br>2=>1,<br>3=>1 |
| push | I/O スレッドがローカルノードから読み出し，別のリモートノードに書き出す経路 | 1=>0,<br>1=>2,<br>1=>3 |
| relay | I/O スレッドがリモートノードから読み出し，別のリモートノードに書き出す経路 | 2=>0, 0=>2,<br>3=>0, 0=>3,<br>3=>2, 2=>3, |
| round-trip | I/O スレッドがリモートノードから読み出し，同じリモートノードに書き出す経路 | 0=>0,<br>2=>2,<br>3=>3 |

以上の結果から，以下の分散方針を導いた．

A)　全スレッドの完了時間が同程度となるよう，以下を考慮して I/O 処理を分散する．

B)　可能な限りユーザバッファと I/O スレッド，ファイルキャッシュが同じ NUMA ノードに配置されるようにする（local 型）

C)　NUMA をまたぐ場合にはユーザバッファの NUMA ノードに可能な限り近い NUMA ノードにキャッシュを配置し，メモリ書き込みが起こるノードに I/O スレッドを配置する（pull 型）

## 9. 関連研究

Kshitij Mehta 等は OpenMP での並列 I/O のインターフェイスを提案し，プロトタイプ実装について評価を実施している[6]．この文献では OpenMP 向けのインターフェイスの提案に主眼を置いており，プロトタイプ実装については NUMA 最適を対象としていない．ただし将来の展開として NUMA 考慮の必要性を示唆している．本研究は NUMA を考慮した最適な分散方針を研究対象とする点が異なる．

また Kwangho Cha 等はマルチコアシステムでの MPI-IO の two-phase I/O では I/O アグリゲータ（他プロセスからの I/O 要求を集積するプロセス）の配置によって他プロセスとの通信コストが異なることを指摘し，この通信コストを削減するための I/O アグリゲータの配置方法を提案している[7]．本研究とは並列動作する I/O 処理主体をどのコアに配置するかで I/O 要求コアとの通信コストを最適にしようとする点が似ている．しかし本研究はスレッド並列を対象とし，単一 I/O の性能向上を目指す点が異なる．

## 10. おわりに

本稿ではスレッド分散型の単一 I/O 要求を高速化する手法として並列分散 I/O を紹介し，将来の HPC では NUMA 配置を意識した分散方式が必要であることを述べた．並列分散方式のひとつとして，サイズ均等に分散する方式を試作し，既存の NUMA マシン上でベストケースとワーストケースの性能を測定した．その結果，サイズ分散に分散する方法でも，メモリ帯域幅に律速しない程度の並列数に抑えることで並列効果が期待できることを確認した．またユーザバッファの配置されるノード数に比例して I/O スレッド数を増減させるのが効率的であるとの結論を導いた．更なる最適化のためには，I/O スレッドの NUMA ノード配置を制御する必要があることを述べ，その方針決定のための基礎性能として，データアクセス経路毎の帯域幅を測定した．その結果から可能な限り local 型に配置し，NUMA ノードを跨ぐ場合は pull 型に配置する方が良いとの分散方針を導き出した．

本検証で得た方針は RX500 での測定に基づく結果であるが，以下のような RX500 と同様の構成のシステムでは同

様の分散方針により効果が得られると期待できる.

- 最長ホップ数が小さい
- NUMA ノード間バスよりもメモリの帯域が大きい.
- NUMA ノード内のコアからの並列アクセスでノード内のメモリ帯域を最大限まで使用できる

また異なる性能バランスのシステムについても，同様にベンチマークによる実測値に基づき最適な分散方針を導き出す必要がある. 将来 HPC システムの性能バランスは現状では不定であるため，対応するには分散を柔軟に設定できるような実装とする必要がある.

今後は, 将来 HPC システムでの実現可能性を追求するため,「レイテンシコアの高度化・高効率化による将来の HPCI システムに関する調査研究」の研究課題のひとつとして進める「次世代高性能並列計算機のためのシステムソフトウェアスタック」[8]検討のためのメニーコア向けカーネルのプロトタイプに実装し，他アーキテクチャでも評価していく.

**謝辞** 本研究は，文部科学省「将来の HPCI システムのあり方の調査研究」の研究課題「レイテンシコアの高度化・高効率化による将来の HPCI システムに関する調査研究」によるものである．研究にあたり議論に参加いただいた参加メンバ各位にこの場を借りて謝辞を述べる.

## 参考文献

1) Super Computer TOP500, http://www.top500.org/
2) Dongarra, J., et al.: The international exasccale software project roadmap, The international journal of high performance computing applications, 25(1), pp. 3-60 (2011).
3) Kenichiro Sakai, Shinji Sumimoto, Motoyoshi Kurosawa: High-Performance and Highly Reliable File System for the K computer, FUJITSU Sci. Tech. J., Vol.48, No.3 (July 2012). http://www.fujitsu.com/global/news/publications/periodicals/fstj/archives/vol48-3.html
4) 酒井憲一郎, 住元真司, 黒川原佳: スーパーコンピュータ「京」の高性能・高信頼ファイルシステム, Magazine FUJITSU, Vol.63, No. 3 (May 2012). http://jp.fujitsu.com/about/magazine/backnumber/vol63-3.html
5) ROMIO: A High-Performance, Portable MPI-IO Implementation, http://www.mcs.anl.gov/research/projects/romio/
6) Kshitij Mehta, Edgar Gabriel, Barbara Chapman: Specification and Performance Evaluation of Parallel I/O Interfaces for OpenMP, OpenMP in a Heterogeneous World, Lecture Notes in Computer Science, Volume 7312, 2012, pp 1-14
7) Kwangho Cha and Seungryoul Maeng: An Efficient I/O Aggregator Assignment Scheme for Collective I/O Considering Processor Affinity,
8) 石川裕, 堀敦史, Gerofi Balazs, 高木将通, 島田明男, 清水正明, 佐伯裕治, 白沢智輝, 中村豪, 住元真司, 小田和友仁: 次世代高性能並列計算機のためのシステムソフトウェアスタック，情報処理学会第 124 回システムソフトウェアとオペレーティング・システム研究会(2013).