Vol.2015-HPC-152 No.9
2015/12/16

Many Core プロセッサ向け MPI ライブラリのマルチスレッド高速化

伏見 政晃[1]　川島 崇裕[1]　野瀬 貴史[1]　安達 知也[1]　志田 直之[1]　住元 真司[1]

概要：Many Core プロセッサでは消費電力を抑えて演算性能を高めるため，動作周波数を抑え，多数のプロセッサ Core を集積している．しかし，最先端のインタコネクトにおいて，単一の Core 性能がボトルネックになり，MPI ライブラリ処理性能が低下する問題がある．通信処理性能を最大限に引き出せる MPI ライブラリの処理方式が必須である．本論文では，Many Core プロセッサ向けの MPI ライブラリにおいて高速化すべき MPI ライブラリ処理を整理し，これらをスレッド並列化することにより高速化する手法について述べる．高速化すべき MPI ライブラリ処理の中で，メモリコピー性能の高速化を取り上げ，Pack 処理と Unpack 処理に伴うメモリコピー処理と集団通信である MPI_Bcast について，OpenMP を用いてスレッド並列化する変更を加え，並列化の効果検証を行った．評価の結果，メモリコピー処理の割合が大きな場合については，スレッド並列化による高速化は効果が見込めるということがわかった．一方，マルチスレッド高速化の効果については，並列化するメッセージサイズとスレッド数により効果が違うことがわかった．

1. はじめに

近年の HPC システムにおいては，消費電力を抑えて演算性能を高めるため，動作周波数を抑えて多数のプロセッサ Core を CPU に集積する Many Core プロセッサが主流となっている．Core 数を増やすことでシステムとしての性能は向上し続けているが，単一 Core 自体の性能向上は頭打ちにある．

このため，性能向上が著しい最先端のインタコネクトにおいて，単一 Core の処理性能がボトルネックとなり，MPI[1] ライブラリの処理性能向上の妨げになる例が報告されている [8]．単一の Core の性能向上が見込めないことを想定し，通信性能を最大限に引き出せる MPI ライブラリの処理方式が必要である．

Many Core プロセッサにおける MPI ライブラリの処理性能向上の一つの手段として，プロセス内で並列化可能な処理をスレッド並列化する方法がある．本論文では，Many Core プロセッサ向けの MPI ライブラリにおいて，スレッド並列化による高速化が見込める処理を整理し，その中でメモリコピー処理に着目したスレッド並列化の効果について検証した結果を示す．

[1] 富士通株式会社
Fujitsu Ltd.

2. Many Core CPU と高性能インタコネクト

本章では，近年の Many Core CPU と高性能インタコネクトの性能の傾向について述べ，その課題について述べる．

2.1 インタコネクト性能

本節では，PC クラスタにおいて広く利用されている InfiniBand[2] と Tofu インタコネクト [3], [4] の通信バンド幅性能の傾向について述べる．

InfiniBand: 2015 年 10 月現在，TOP 500 で上位に位置する HPC システムにおいて，最も採用されているインタコネクトは InfiniBand である．InfiniBand Trade Association(IBTA)[2] により 2000 年に規格が策定された InfiniBand SDR 4X は，1GB/s のバンド幅性能であったが，2GB/s の DDR 4X，4GB/s の QDR 4X，7GB/s の FDR 4X，そして 12.5GB/s の EDR 4X と継続的な性能向上を続けている．QDR から FDR，EDR への性能向上率については少し陰りが見えているが，引き続き向上を続けている．しかし，計算ノードあたりの性能向上率に比べて性能向上率が高くないため，性能を補うために，複数の InfiniBand を束ねて利用する Multirail と呼ばれる手法が用いられている．

Tofu，Tofu2: 「京」ならびに，富士通の FUJITSU Supercomputer PRIMEHPC FX10 (以下，FX10) においては，Tofu インタコネクト [3], [4] が採用されて

いる．Tofu インタコネクトでは，双方向 10GB/s のネットワークインターフェイスを 4 つ搭載し，高いシステム演算性能を実現している．後継のインタコネクトとして FUJITSU Supercomputer PRIMEHPC FX100 (以下，FX100) で採用された Tofu インタコネクト 2(Tofu2) があり，双方向 25GB/s のネットワークインターフェイスを 4 つ搭載している．Tofu，Tofu2 上の MPI ライブラリでは，複数のネットワークインターフェイスを活用した集団通信が実装されている．

上記の通り，インタコネクトのバンド幅は，単一リンクの通信バンド幅性能の向上率に少し陰りが見えてきている状況だが，通信バンド幅の向上には Multirail やリンク数の増強 (8X，12X) など，技術的に改善余地が残されており，この傾向は今後もまだ続くと推測される．

2.2 Many Core プロセッサの単一 Core 性能

Many Core CPU の代表例として挙げられるのは Xeon Phi である．ここでは，Xeon Phi に加え，「京」で採用されている SPARC64 VIIIfx，および，FX10 で採用されている，SPARC64 IXfx，および，FX100 で採用されている SPARC64 XIfx の CPU 周波数とメモリバンド幅を示し，単一 Core 性能について議論する．

2.2.1 CPU 周波数

Many Core CPU は，CPU 全体の消費電力対処理性能を高めるために，単一 Core の周波数と動作電圧を下げ，さらに単一 Core の CPU チップ内での面積を小さくすることにより，一定の消費電力と CPU チップ面積で多くの CPU Core を搭載している．例えば，Xeon Phi の CPU 周波数は 1.238 GHz(7110P)，SPARC64 IXfx は 1.848GHz(FX10)，SPARC64 XIfx は 2.2GHz 程度 (FX100) と，同世代の Xeon の 3.2GHz(E7-8893 v3) に比べ，CPU 周波数は高くない．

2.2.2 Core あたりのメモリバンド幅性能

Many Core CPU は，CPU 全体の演算性能が高いため，相対的に Core あたりのメモリバンド幅性能が低くなる傾向にある．

Xeon Phi (7110P) 1.2TFLOPS の演算性能である Xeon Phi (7110P) は 61Core を搭載し，GDDR5 メモリで 352GB/s のメモリバンド幅性能である．これは Core あたり 5.77GB/s のメモリバンド幅性能となる．

SPARC64 VIIIfx 128GFLOPS の演算性能を持つ SPARC64 VIIIfx では，8Core を搭載し，DDR3-1066 DIMM を 8 チャネル採用することで，64GB/s のメモリバンド幅性能を実現している．これは Core あたり 8GB/s のメモリバンド幅性能となる．

SPARC64 IXfx 236.5GFLOPS の演算性能を持つ SPARC64 IXfx では，16Core を搭載し，DDR3-1333 DIMM を 8 チャネル採用することで，85GB/s のメモリバンド幅性能を実現している．これは Core あたり 5.31GB/s のメモリバンド幅性能となる．この Core あたりのメモリバンド幅は，Tofu のネットワークインターフェイス 1 つ分の性能と同等であるが，複数のネットワークインターフェイスを使用したメモリコピーを伴う場合，メモリコピー性能がボトルネックとなる．

SPARC64 XIfx 1.1TFLOPS の演算性能を持つ SPARC64 XIfx は 32 個の Core と 2 つのアシスタント Core と呼ばれる Core を搭載している．高い演算性能を支えるため，高バンド幅積層メモリである HMC (Hybrid Memory Cube) を 8 個搭載することで，480GB/s (240GB/s 双方向) を達成している．これは Core あたり 15GB/s のメモリバンド幅性能になる．この Core あたりのメモリバンド幅は，Tofu2 のネットワークインターフェイス 1 つ分の性能とほぼ同等であるが，複数のネットワークインターフェイス使用したメモリコピーを伴う場合，メモリコピー性能がボトルネックとなる．

同世代の Xeon の 3.2GHz (E7-8893 v3，4Core) の Core あたりのメモリバンド幅性能は 25.5GB/s であり，Many Core CPU の Core あたりのメモリバンド幅性能は，一般の Multi Core CPU の Core あたりメモリバンド幅性能に比べて低いことがわかる．

以上の通り，Many Core プロセッサの場合，CPU 周波数は抑えられ，単一 Core あたりのメモリコピーバンド幅も CPU の演算性能と比較して近年あまり伸びていないと言える．

2.3 ハードウェア性能の動向から見える MPI ライブラリの課題

2.1 節，2.2 節で示したように，近年，インタコネクトの性能が継続して向上しているのに対し，Many Core プロセッサの CPU 周波数は抑えられているため，単一 Core あたりの処理性能は伸びていない．また，単一 Core あたりのメモリコピー性能も，インタコネクトの通信性能に比べて伸びていないことがわかる．

既存の MPI ライブラリ [5], [6] は，単一スレッドによる処理を前提に実装されている．このため，Many Core CPU 上で MPI ライブラリを実行すると，単一 Core 性能がボトルネックとなって通信性能が上がらない場合がある．この問題は，単一 Core の命令実行性能とメモリコピーバンド幅の 2 つの点が課題となる．

2.4 MPI ライブラリ処理高速化の関連研究

本節では，前節の内容を踏まえたうえで，MPI ライブラ

リやアプリケーションの処理を高速化するという観点で，これまでに行われてきた工夫と，そこで見えた課題について述べる．

2.4.1 単一 Core 命令実行性能高速化と課題

「京」，および，FX10，FX100 では，ハードウェアの構成を直接意識した専用の集団通信アルゴリズムを考案することにより，高速化が行われた [8]．MPI_Allreduce における Reduce 演算の高速化のために，コンパイラの SIMD 化を活用できるような修正を追加し，SIMD 演算を利用しない場合と比べて 1.54 倍の性能向上を実現している．

しかし，前節で述べたように，高性能なインタコネクトを複数制御するには CPU 命令実行性能に限界があるため，複数の Core による処理性能向上，もしくは，ネットワークインターフェイスハードウェアでの工夫が必要である．

2.4.2 メモリコピー性能高速化と課題

本項では，メモリコピー性能高速化としてノード内通信の高速化と Derived Datatype を用いた通信の高速化について述べる．

Derived Datatype を用いた通信の高速化については，Pack/Unpack 処理の高速化の研究が見られる．高速化の手法として，スレッド並列化することで高速化した研究がある [13]．また，アプリケーション側で行われた高速化としてステンシル計算における袖通信の高速化について，OpenMP を用いて Pack/Unpack 処理を高速化 [16]，また，ネットワークインターフェイスの Scatter/Gather 転送機構を用いた高速化 [14], [15] がある．

ノード内通信の高速化については，これまで様々な研究が行われてきている．多く取り組まれてきた手法は， 通常，共有メモリを用いてノード内通信を行う場合，2 回のコピーが必要になるが，ノード内通信では相互にメモリ空間をマップし，1 コピー通信にすることで高速化できる．関連研究としては，KNEM[9]，XPMEM[10], [11]，CMA[12] などの関連研究がある．ただし，1 コピー性能以上は出せないのが問題である．より性能向上するためには，マルチスレッドによる並列コピーでの性能向上を考える必要がある．さらには，ノード内通信をハードウェアにより高速化する研究としては， MVAPICH2-MIC[7] では，Xeon-Phi の DMA 転送ハードウェアを利用することにより高速化している．

2.4.3 Many Core CPU 向けの MPI ライブラリ高速化の方向性

以上述べたように，Many Core CPU 上で用いる MPI ライブラリの高速化を考えるあたり，単一 Core 実行性能とメモリコピー性能の高速化に対応する案として複数 Core による並列処理が挙げられる．そこで，マルチスレッド処理による MPI ライブラリ処理の高速化を考え，その利点や課題について次章で整理する．

3. MPI ライブラリ処理のマルチスレッド並列化における課題

本章では，前章で述べた Many Core CPU における MPI ライブラリ処理の課題から，高速化すべき MPI ライブラリ処理を定義する．そして，これらのマルチスレッド並列化により高速化する場合の課題について述べる．

3.1 高速化すべき MPI ライブラリ処理

前章で述べたように，マルチスレッド処理による MPI ライブラリの高速化の対象として，単一 Core 命令実行性能高速化，メモリコピー性能高速化を対象に実現を考える．

3.2 マルチスレッドによる並列化実現における課題

マルチスレッドを用いて MPI ライブラリ処理を実現する場合，2 つの課題がある．

一つは，マルチスレッド環境を元々シングルスレッドを前提に実装されている MPI ライブラリにどのように導入するかの課題である．元々 MPI ライブラリは HPC アプリケーションと共に利用されている．HPC アプリケーションは，シングルスレッドで書かれる場合，そして，OpenMP や並列化コンパイラなどマルチスレッドで書かれる場合がある．前者のシングルスレッドで書かれたアプリケーションの場合で他に余剰の Core が存在しない場合は一般にマルチスレッド化による性能向上は困難である．一方，後者のマルチスレッド環境で書かれた場合においても，利用可能なスレッド数など OpenMP や並列化コンパイラと MPI ライブラリ間で実行コンテキストである Core 資源を効果的に交換する仕組みが必要である．

もう一つは，MPI ライブラリ処理をどのように並列化するかである．MPI ライブラリ利用において，マルチスレッド処理を想定すると MPI_THREAD_MULTIPLE 環境の導入が自然である．MPI 内部処理を含めてマルチスレッド化し，その中で MPI 全体処理をマルチスレッド並列化を進めることが可能である．しかし, MPI_THREAD_MULTIPLE 環境自体は内部で綿密な排他制御機構を導入する必要があるため，排他制御導入による実行性能の低下が懸念される．また，複数スレッドを意識した実装は容易ではない．他の並列化の方式としては OpenMP を利用する方式がある．MPI ライブラリ自体はシングルスレッドで実行されるが，並列化に効果がある部分のみを OpenMP により実装するのである．MPI ライブラリのコンパイルに OpenMP を用いたスレッド並列化に対応したコンパイラ，実行に OpenMP ランタイムが必要であるが，比較的容易に実装可能な点がメリットである．本方式は論文 [13] においても使用されている．

4. MPI ライブラリ処理のマルチスレッド処理高速化の試作評価

4.1 試作評価目的

これまでに述べてきたように，特定の MPI ライブラリ処理の高速化の手段として，スレッド並列化が考えられる．スレッド並列化による高速化が期待できる処理として挙げられるのは，主にメモリコピー処理と，複数ネットワークインターフェース活用の 2 点である．今回は，メモリコピー処理に着目し，以下の 3 点を目的としてスレッド並列化の検証を行う．

Pack/Unpack 処理のスレッド並列化の評価
Derived Datatype の Pack 処理と Unpack 処理をスレッド並列化し，その効果と課題を検証することを 1 点目の目的とする．Derived Datatype の合計データサイズは，ブロックサイズとブロック数の積で定まる値のため，ブロックサイズ，ブロック数，およびスレッド並列数に着目した実験により，粒度の細かい検証を行う．

Derived Datatype を用いた 1 対 1 通信の評価
スレッド並列化した Pack/Unpack 処理を含む通信の評価を 2 点目の目的とする．基本的な通信で課題を見出すために，Derived Datatype を用いた 1 対 1 通信に着目した検証を行う．

MPI_Bcast におけるスレッド並列化の評価　集団通信におけるスレッド並列化による効果の検証の一環として，今回はまず集団通信の中でも比較的通信が単純な MPI_Bcast 関数において，共有メモリを扱う際のノード内コピー処理のスレッド並列化を行い，その効果と課題を明らかにすることを 3 点目の目的とする．

4.2 実験方法

本節では，MPI ライブラリ処理におけるスレッド並列化の効果を検証するために行った実験方法について述べる．以下では，Pack 処理と Unpack 処理を次のように定義して用いる．

Pack 処理　メモリ上に不連続に並んだデータを，連続に並ぶように別のバッファにコピーする処理.

Unpack 処理　Pack 処理されたデータを，元の不連続な並びのデータになるように別のバッファへコピーする処理.

また，測定環境は以下である．

測定環境　FUJITSU Supercomputer PRIMEHPC FX100
- CPU: SPARC64 XIfx
- インタコネクト：Tofu インタコネクト 2

なお，MPI ライブラリとしては，FUJITSU Software Technical Computing Suite に含まれる MPI ライブラリ（以降、富士通 MPI と呼ぶ）を用いた．

表 1　Derived Datatype に関する検証に用いたパラメーター

パラメーター	範囲
ブロックサイズ	1 B 以上 1 MiB 以下
ブロック数	1 以上 1024*1024 以下
ストライドサイズ	1 B 以上 1 MiB 以下
スレッド並列数	1, 8

4.2.1 MPI_Pack 関数と MPI_Unpack 関数の内部処理のスレッド並列化

MPI_Pack 関数では，データの Pack 処理をする際に，メモリ上に不連続に並んだデータの個数（ブロック数）分だけメモリコピーを行う．このメモリコピーを行う処理について，OpenMP の parallel 構文で括ることで，スレッド並列化した．MPI_Unpack 関数では，MPI_Unpack 関数とは逆に，Pack 処理されているデータを，元の不連続な配置に戻すために，ブロック数分だけメモリコピーが発生する．したがって，そのメモリコピー処理を OpenMP の parallel 構文でスレッド並列化した．これらの修正の効果を検証するために，データがメモリ上に一定の間隔で並んだ Derived Datatype を用いて，表 1 に示すパラメーターを変化させ，MPI_Pack 関数と MPI_Unpack 関数それぞれの実行時間を測定した．ただし，測定は表 1 において，（ブロックサイズ）＜（ストライドサイズ）を満たす範囲でのみ行った．

4.2.2 Derived Datatype を用いた 1 対 1 通信の内部処理のスレッド並列化

Derived Datatype のデータを用いて通信を行う場合，MPI ライブラリ内部の 1 対 1 通信部では，送信側で一時バッファへの Pack 処理を行ってから送信する．逆に，受信側では，Pack 処理されたデータを，元の不連続な並びのデータになるように Unpack 処理を行う．したがって，このときの Pack 処理，Unpack 処理についても，4.2.1 項と同様に OpenMP の parallel 構文によるスレッド並列化を行った．

効果の検証のために，表 1 に示すパラメーターを用いて，次の通信レイテンシを測定した．
- 2 プロセスでのノード間 PingPong
- 2 プロセスでのノード内 PingPong

4.2.3 MPI_Bcast における共有メモリ通信のスレッド並列化

MPI ライブラリの各集団通信では，複数のアルゴリズムが実装されている．異なるアルゴリズムでの比較は，課題を明らかにすることが困難になる可能性がある．そこで，今回は，富士通 MPI の MPI_Bcast 関数で実装されている Trinaryx3 アルゴリズムに着目した検証を行う．Trinaryx3 アルゴリズムは，Tofu インタコネクト向けに最適化された集団通信アルゴリズムであり，メッセージをセグメント分割してパイプライン転送を行うという特徴がある．同一ノード内のプロセス間では共有メモリを使ったメモリコピーによる通信が行われる．そこで，この共有メモリ通

表 2　MPI_Bcast の検証に用いたパラメーター

パラメーター	範囲
ノード数	8
形状	2x2x2
ノード内プロセス数	2
メッセージサイズ	16 KiB 以上 4 MiB 以下
セグメントサイズ	64 KiB, 128 KiB, 256 KiB
スレッド並列数	1, 2, 4, 8

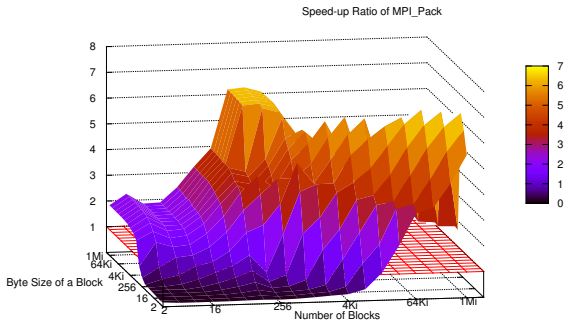

図 1　MPI_Pack 関数を 8 スレッド並列で実行した場合の，シングルスレッド実行に対する性能向上率

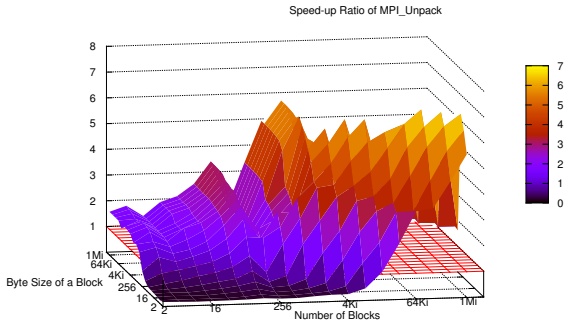

図 2　MPI_Unpack 関数を 8 スレッド並列で実行した場合の，シングルスレッド実行に対する性能向上率

信におけるメモリコピー処理について，4.2.1 項と同様に OpenMP の parallel 構文を用いてスレッド並列化を行った．なお，Trinaryx3 アルゴリズムの詳細は，論文 [8] を参照されたい．

効果の検証のために，表 2 に示すパラメーターを用いて測定を行った．

4.3　実験結果

初めに，MPI_Pack 関数と MPI_Unpack 関数の内部処理をスレッド並列化したときの，それぞれの関数の実行時間を図 1，2 に示す．

図 1，2 より，等間隔に並んだ単純な Derived Datatype のデータを扱う場合，ブロックサイズとブロック数の積で

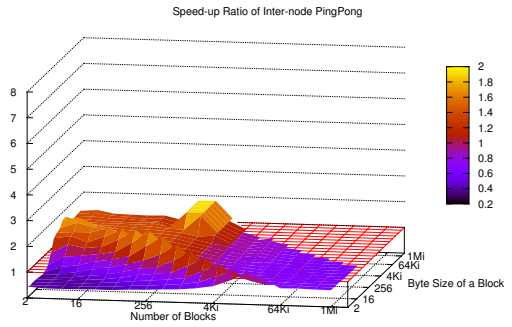

図 3　Derived Datatype を用いたノード間 PingPong を 8 スレッドで実行した場合の，シングルスレッド実行に対する性能向上率

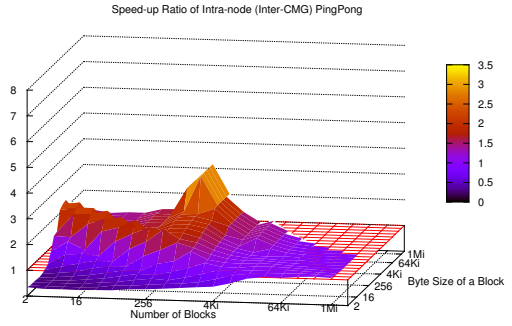

図 4　Derived Datatype を用いたノード内 PingPong を 8 スレッドで実行した場合の，シングルスレッド実行に対する性能向上率

表される合計のデータサイズが大きくなるほど，シングルスレッド実行の場合と比較して，スレッド並列化した場合の性能向上率が高くなっている．特に，合計データサイズが 1 MiB 以上では，約 6 倍の性能向上率になっている．したがって，MPI_Pack 関数や MPI_Unpack 関数の場合，単純なパターンでは，スレッド並列化の効果は十分にあると言える．

次に，MPI ライブラリ内部の 1 対 1 通信部での Pack/Unpack 処理をスレッド並列化した場合の，PingPong レイテンシの性能向上率について，図 3，図 4 に示す．

図 3 より，単純な Derived Datatype を用いたノード間 PingPong を 8 スレッドで実行した場合は，通信レイテンシがスレッド並列化する前と比べて最大で約 2 倍向上している．また，ノード内 PingPong でも，8 スレッドで実行した場合の方が，最大で約 3 倍向上している．MPI_Pack 関数や，MPI_Unpack 関数の実行時間に比べて性能向上率が低いことについては，考察で触れる．

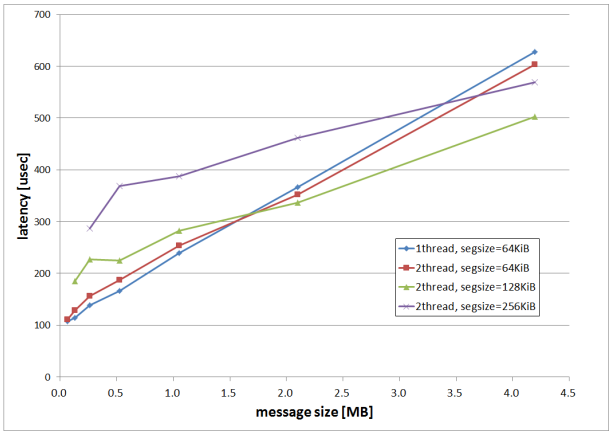

図 5 2 スレッド実行におけるセグメントサイズと性能の関係

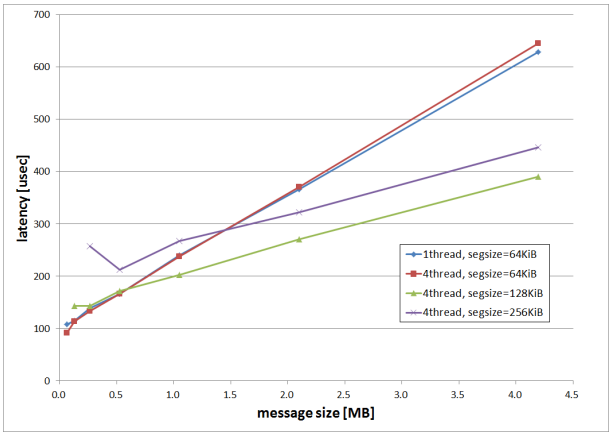

図 6 4 スレッド実行におけるセグメントサイズと性能の関係

最後に，MPI_Bcast 関数における，ノード内コピー処理のスレッド並列化の効果について述べる．図 5，図 6，図 7 は，MPI_Bcast の Trinaryx3 アルゴリズムにおいて，セグメントサイズとスレッド並列数を変えたときの性能を示している．1 スレッド実行でセグメントサイズを固定する場合，セグメントサイズ 64KiB 以上では，セグメントサイズ 64KiB の測定結果がメッセージサイズに関係なく最も良い性能であったため，各スレッド数における結果との比較用に表示している．また，メッセージサイズよりもセグメントサイズが大きな範囲の値については，グラフから除外している．

まず，スレッド並列数ごとに結果を確認する．図 5 より，2 スレッドの場合は，セグメントサイズが 128KiB のときが最もスレッド並列効果が高く，メッセージサイズがおよそ 1.8MiB 以上で 1 スレッドの結果を上回り，メッセージサイズが 4MiB の場合で 25%の高速化が確認できる．図 6 より，4 スレッドの場合は，2 スレッドの場合と同様にセグメントサイズが 128KiB のときに，最もスレッド並列化の効果があり，最大 61%の高速化が確認できる．図 7 より，8 スレッドの場合は，セグメントサイズが 64KiB のときに，最もスレッド並列化による高速化が確認できるが，メッセージサイズが長くなるにつれて，セグメントサイズ

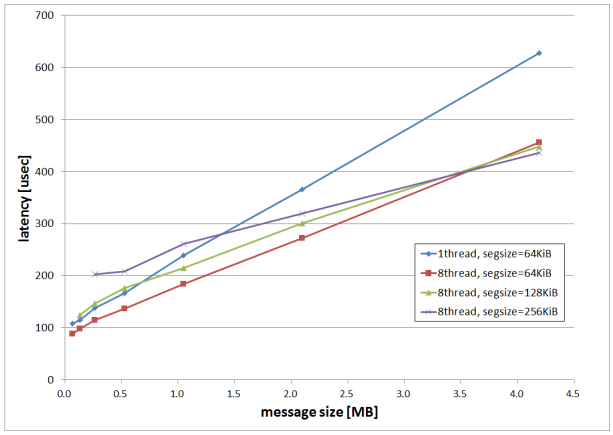

図 7 8 スレッド実行におけるセグメントサイズと性能の関係

ごとの性能差はほとんど見られなくなっている．

次に，セグメントサイズが 128KiB のときのスレッド並列数に着目する．4 スレッドの並列効果が最も高く，次に 8 スレッド，2 スレッドの順となっている．このことから，共有メモリ通信のスレッド並列化は，ある程度のメッセージ長があれば効果を確認することはできるが，スレッド並列数に応じた改善効果は期待できないことがわかる．

5. 考察

5.1 Pack/Unpack 処理に関する考察

今回スレッド並列化による性能向上を試みたのは，全体の処理の中のメモリコピー処理部分だけである．したがって，メモリコピー処理にかかる時間がどれだけの割合を占めているかで，スレッド並列化しない場合に対する性能向上率が変わるはずである．まず，図 1 と図 2 に示される MPI_Pack 関数と MPI_Unpack 関数におけるスレッド並列化の効果については，全体としてブロックサイズとブロック数の積で表される合計のデータサイズが大きくなるほど性能向上率も大きいという傾向にあることがわかる．Derived Datatype を用いた PingPong では，処理が単純な Pack 処理，Unpack 処理，送受信処理に分かれているならば，性能向上率のグラフは図 1 と図 2 と同様の傾向を示すはずである．しかし，図 3，図 4 に示されるように，性能向上率の傾向は MPI_Pack/MPI_Unpack の場合と異なっている．図 3，図 4 には，次の 3 つの特徴がある．

特徴 1 ある一定の値までは，MPI_Pack 関数や MPI_Unpack 関数と同様に，ブロックサイズとブロック数の積が大きくなるほど性能向上率も大きくなる．

特徴 2 ブロックサイズとブロック数の積がある一定の値を超えると，性能向上率は低くなる，もしくはスレッド並列化前よりも悪くなる．

特徴 3 ブロックサイズとブロック数の積が，特徴 1 と特徴 2 の閾値より大きい場合でも，特定の値付近だけは性能向上率が大きくなっている．

特徴 1 と特徴 2 については，通信時のアルゴリズムの違

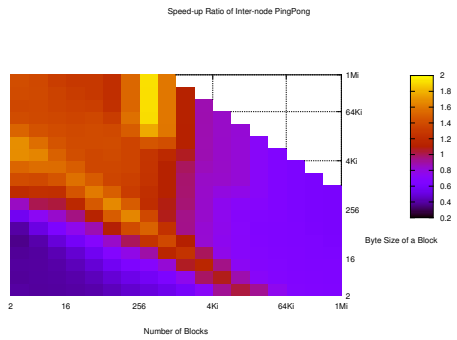

図 8 Derived Datatype を用いたノード間 PingPong の 8 スレッドで実行による性能向上率（2 次元）

いによるものであると考えられる．これについて説明するために，3 次元グラフである図 3 において，横軸をブロック数，縦軸をブロックサイズにとった場合の性能向上率の 2 次元グラフを，図 8 に示す．

図 8 より，特徴 1 と特徴 2 を分けるブロックサイズとブロック数の積の値は，約 40 KiB であることがわかる．約 40 KiB は，PRIMEHPC FX100 で用いる富士通 MPI において，通信アルゴリズムが切り替わる閾値となるデータサイズである．

富士通 MPI では，メモリ上に不連続に配置されたデータを送信する場合，ブロックサイズとブロック数の積で表される合計のデータサイズが約 40 KiB 未満の場合は，それらのデータを全て MPI ライブラリが管理する送信用バッファに Pack してから送信する．受信側では，MPI ライブラリが管理する受信用バッファに 1 つの連続データとして受信した後，元の不連続な配置になるように Unpack する．したがって，データの送受信が完了するまでに，Pack 処理と Unpack 処理はそれぞれ 1 回ずつしか行われないため，スレッド並列化による性能向上率は，MPI_Pack 関数や MPI_Unpack 関数と同様の傾向を示していると考えられる．

一方，合計データサイズが約 40 KiB 以上となるメモリ上の不連続データを送信する場合，送信側ではあらかじめ決められたセグメントサイズごとにデータの Pack 処理と送信処理を，パイプライン形式で行う．受信側では，Pack されたデータを受信する度に Unpack 処理を行う．このときのセグメントサイズは約 12 KiB に設定されているため，全てのデータの転送時に得られるスレッド並列化による性能向上率は，約 12 KiB の Pack/Unpack 処理における性能向上率で律速される．約 12 KiB の Pack/Unpack 処理での性能向上率は低いため，合計データサイズが約 40 KiB 以上の Derived Datatype の PingPong では，MPI_Pack 関数や MPI_Unpack 関数よりも性能向上率が伸びていないと考えられる．

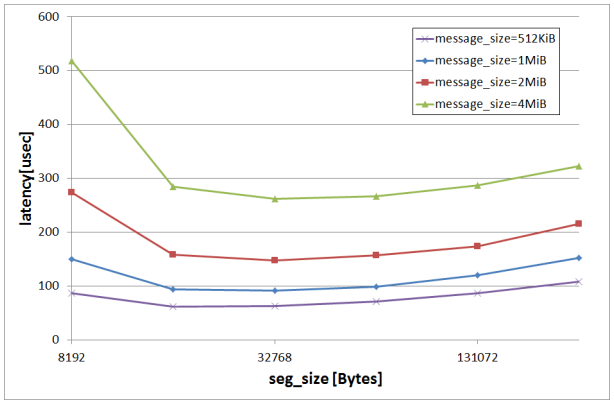

図 9 MPI_Bcast の Trinaryx3 アルゴリズムにおけるセグメントサイズごとのノード間通信性能傾向

上記を踏まえると，Pack/Unpack 処理自体は，対象のデータサイズが大きくなるほどスレッド並列化の効果が大きくなる傾向にあることから，約 40 KiB に設定していた通信アルゴリズム切り替えの閾値を大きくすることで，Derived Datatype を用いた 1 対 1 通信の，スレッド並列化による性能向上率を上げることができる可能性がある．また，パイプラインアルゴリズムの場合も，通信レイテンシの増加とのトレードオフを考慮し，セグメントサイズを調整することにより，スレッド並列化の効果を高めることができる可能性がある．これらの検証は今後行っていきたい．

また，特徴 3 については原因がまだ推測できていないため，今後さらに検証することとする．

5.2 MPI_Bcast 関数に関する考察

Pack/Unpack 処理のスレッド並列化では，メッセージサイズを大きくするほど性能向上率が高くなる傾向にあったが，MPI_Bcast では，Trinaryx3 アルゴリズムにおけるセグメントサイズを変えていっても，セグメントサイズが 64KiB から 128KiB 程度で性能向上率が頭打ちになるという結果であった．理由としては，セグメントサイズが大きくなると，メモリコピー処理のスレッド並列化の効果は高くなるが，通信レイテンシが大きくなることでパイプライン転送の効果が得にくくなるというトレードオフが考えられる．これについて考えるために，同一環境でノード内 1 プロセスでメッセージサイズごとにセグメントサイズを変えながら Trinaryx3 アルゴリズムを実行した際のレイテンシの傾向を図 9 に示す．ノード内 1 プロセスのため，ノード間通信のみが行われることに注意されたい．

図 9 より，1MiB 程度のメッセージサイズでは，セグメントサイズ 16KiB 付近まではレイテンシが短くなりパイプライン転送が有効に作用しているのに対し，セグメントサイズが 32KiB を越えるとレイテンシが大きくなることがわかる．したがって，セグメントサイズ 32KiB 以上では，確かにメモリコピー処理のスレッド並列化による高速化とセグメントサイズ増加に伴う通信レイテンシ増加のトレー

ドオフが発生する．よって，今回の結果は，そのトレードオフを加味した最適なセグメントサイズが，64KiB から 128KiB であることを示していると考えることができる．

また，MPI_Bcast における実験では，メッセージサイズの大きな範囲であっても，スレッド数が多いほど性能が向上するわけではないという結果も示していた．今回の条件，環境では，1 プロセスあたり 8 スレッド実行でも Core 数を上回るスレッド数にはならない．4 スレッド並列において最も性能が向上した理由については，今後さらに検証を行う．

6. 関連研究

MPI ライブラリ処理のスレッド並列化に関する研究として，メモリコピー処理のスレッド並列化を試みた論文 [13] がある．論文 [13] では，Derived Datatype を扱う際の Pack/Unpack 処理のスレッド並列化と，ノード内の共有メモリ通信におけるメモリコピー処理のスレッド並列化が行われている．Pack/Unpack 処理のみに着目した評価としては，Derived Datatype の合計データサイズとスレッド並列数に着目した評価が行われているが，その合計データサイズをブロックサイズとブロック数に分けた，より細かい粒度での評価は行われていない．また，スレッド並列化した Pack/Unpack 処理と通信が混在する場合の評価も行われている．その評価には，多数のプロセスが Derived Datatype を用いた通信を行うベンチマークプログラムが用いられている．共有メモリ通信におけるメモリコピー処理のスレッド並列化については，ノード内の 1 対 1 通信での評価が行われている．

本論文では，MPI_Pack 関数と MPI_Unpack 関数のスレッド並列化の効果について，Derived Datatype のブロックサイズ，ブロック数，スレッド並列数に着目して検証を行った．Derived Datatype を用いた通信については，1 対 1 通信に着目した効果と課題の分析を行った．また，スレッド並列化による集団通信の高速化の試みとして，ノード内での共有メモリ通信におけるメモリコピー処理のスレッド並列化を，MPI_Bcast 関数に適用して検証を行った．

7. まとめ

本論文では，Many Core プロセッサ向けの MPI ライブラリにおいて高速化すべき MPI ライブラリ処理を整理し，これらをスレッド並列化することにより高速化する手法を検討した．

検討の結果，単一 Core 命令実行性能とメモリコピー性能について高速化すべきであることを示した．また，スレッド並列化による高速化の効果を検証するため，メモリコピー性能の高速化を取り上げ，Pack 処理と Unpack 処理に伴うメモリコピー処理と集団通信である MPI_Bcast について，OpenMP を用いてスレッド並列化する変更を加え，検証を行った．検証の結果，メモリコピー処理の割合が大きな場合については，スレッド並列化による高速化は効果が見込めるということがわかった．一方，マルチスレッド高速化の効果については，並列化するメッセージサイズとスレッド数により効果が違うことがわかった．これは，MPI 内部の実装アルゴリズムにより異なるため，個々の場合について検討していく必要があると考えている．

また，今回は行わなかったが，MPI_Reduce 関数など演算を伴う集団通信への適用，単一 Core の命令実行性能の高速化としての複数のネットワークインターフェイスのハンドリング処理についても，スレッド並列化の効果があると考えられるので，今後試作し，その効果について検証したい．

次期システムの実現に向け，引き続き課題解決に取り組む．

謝辞 本論文の一部は，文部科学省「特定先端大型研究施設運営費等補助金（次世代超高速電子計算機システムの開発・整備等 ）」で実施された内容に基づくものである．

参考文献

[1] "The Message Passing Interface (MPI): standard: http://www.mpi-forum.org"
[2] InfiniBand Trade Association: http://www.infinibandta.org/.
[3] Yuichiro Ajima, Shinji Sumimoto, and Toshiyuki Shimizu.: "Tofu: A 6d mesh/torus interconnect for exascale computers", In *IEEE Computer*, Vol.42, No.11, pp.36–40 (2009).
[4] Yuichiro Ajima, Yuzo Takagi, Tomohiro Inoue, Shinya Hiramoto, and Toshiyuki Shimizu.: The Tofu Interconnect, In *Hot Interconnects*, pp.87–94 (2011).
[5] Open MPI: http://www.open-mpi.org/.
[6] MPICH: http://www.mpich.org/.
[7] Potluri, S., Hamidouche, K., Bureddy, D., Panda, D.K.: "MVAPICH2-MIC: A High Performance MPI Library for Xeon Phi Clusters with InfiniBand", *Proc. XSW '13 Proceedings of the 2013 Extreme Scaling Workshop (XSW 2013)*, IEEE Computer Society, pp.25–32 (2013).
[8] 松本幸, 安達知也, 住元真司, 南里豪志, 曽我武史, 宇野篤也, 黒川原佳, 庄司文由, 横川三津夫：MPI_Allreduce の「京」上での実装と評価, 情報処理学会論文誌, コンピューティングシステム, Vol.5, No.5, pp.152–162 (2012).
[9] Brice Goglin, Stphanie Moreaud.: "KNEM: a Generic and Scalable Kernel-Assisted Intra-node MPI Communication Framework", Journal of Parallel and Distributed Computing, Vol.73, No.2, pp.176–188 (2013).
[10] XPMEM, cross-process memory mapping (online), available from ⟨http://code.google.com/p/xpmem/⟩
[11] https://www.nersc.gov/assets/NUG-Meetings/2012/HowardP-MPI-NUG2012.pdf
[12] Jerome Vienne. 2014. Benefits of Cross Memory Attach for MPI libraries on HPC Clusters. In Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment (XSEDE '14). ACM, New York, NY, USA, Article 33 , 6 pages.
[13] Min Si, Antonio J. Peña, Pavan Balaji, Masamichi Takagi, Yutaka Ishikawa: "MT-MPI: Multithreaded MPI for

Many-Core Environments", ICS '14, Proceedings of the 28th ACM international conference on Supercomputing, pp. 125–134 (2014)
[14] M. Li, H. Subramoni, K. Hamidouche, X. Lu, and D. K. Panda: "High Performance MPI Datatype Support with User-mode Memory Registration: Challenges, Designs and Benefits", IEEE Cluster 2015, pp.226–235 (2015)
[15] Santhanaraman, Gopalakrishnan and Wu, Jiesheng and Panda, DhabaleswarK.: "Zero-Copy MPI Derived Datatype Communication over InfiniBand" Lecture Notes in Computer Science, Recent Advances in Parallel Virtual Machine and Message Passing Interface, Vol.3241, pp47–56 (2004).
[16] 村井均，佐藤三久：並列プログラミング言語 XcalableMP におけるステンシル通信の効率的な実装，情報処理学会研究報告，Vol.2013–HPC–140，No.8，pp.1–9 (2013).