Vol.2017-HPC-162 No.17
2017/12/19

非ブロッキング集団通信の通信隠蔽効果に関する調査

南里 豪志[1,a)]　大島 聡史[1,b)]　小野 謙二[1,c)]

概要：本稿では，非ブロッキング集団通信による通信隠蔽技術について、特にプログレススレッドを用いた場合の実用上の効果を計測し、評価した。従来の通信隠蔽率のみを計測するベンチマークプログラムでは、プログレススレッドを利用した場合の計算性能の低下による影響が計測結果に反映されないため、実用性の検証が困難である。そこで本稿では、計算と通信を含む総合的な性能評価を行うため、スレッド並列とプロセス並列によるハイブリッド並列のベンチマークプログラムを作成した。このプログラムは、通信と計算の量をそれぞれ明示的に指定するため、プログレススレッドへの CPU コアの割り当て方法やスレッドのスケジューリングポリシーなどの実行時パラメータを変化させた場合の、計測結果の相互比較も可能となった。
このプログラムを、Fujitsu PRIMERGY CX400 および Fujitsu PRIMEHPC FX100 上で実行し、性能を計測した。その結果、Alltoall では、適切な実行時パラメータを選択することにより、プログラム全体としての性能向上が見込めることが分かった。一方、Allreduce では、特にノード内で複数のプロセスを起動した場合に、性能が低下する場合があることが分かった。これらの結果から、非ブロッキング集団通信の利用にあたっては、使用する集団通信の種類やメッセージサイズ、計算量等に応じて、効果を事前に調査することが重要であることを確認した。
また、非ブロッキング集団通信を推進するもう一つの手段であるオフロード機能について、Mellanox 社の SHArP 機能を用いた場合の通信隠蔽効果を予備評価し、通信隠蔽による性能向上が見込めることを確認した。

Study on the Effect of Communication Overlapping with Non-Blocking Collectives

NANRI Takeshi[1,a)]　OHSHIMA Satoshi[1,b)]　ONO Kenji[1,c)]

1. 背景

プロセス並列プログラムにおける通信の標準規格である Message Passing Interface (MPI) で定義されている非ブロッキング通信は、あるプロセス間の通信と、それらのプロセス上での計算を並行して進行させることにより、見かけ上、通信に要する時間の一部を計算時間で隠蔽することを可能とする。このうち、MPI-3.0 規格において採用された非ブロッキング集団通信は、多くの並列プログラムで頻出する、複数のプロセス間でのデータ複製や集約のような定型通信である集団通信について、通信時間の隠ぺいを図るものである。集団通信は、計算機の大規模化に伴って所要時間が増大するため、並列プログラムのスケーラビリティ向上の手段として、この非ブロッキング集団通信が期待されている。

非ブロッキング集団通信は、その集団通信に参加する各プロセスが計算を行っている間に、集団通信を完了させるための通信アルゴリズムを推進することで、通信時間を隠蔽するため、この、集団通信アルゴリズムの推進手法が、隠蔽効果に大きく影響する。現在、多くの MPI ライブラリで採用されている集団通信アルゴリズム推進手法のうち、特にプログレススレッドを用いた推進手法は、プログ

[1] 九州大学
Research Institute for Information Technology, Kyushu University
[a)] nanri.takeshi.995@m.kyushu-u.ac.jp
[b)] ohshima@cc.kyushu-u.ac.jp
[c)] keno@cc.kyushu-u.ac.jp

ラム中に MPI_Test 関数のようなアルゴリズム推進のための MPI 関数を挿入しなくても高い通信隠蔽率が期待できる上、インターコネクトネットワークに特殊なハードウェアを必要としないという特徴があるため、特に注目されている。一方、この手法は、プロセス毎に集団通信アルゴリズム推進専用のスレッドに CPU コアを割り当てる必要があるため、その分、計算性能が低下する。特に、スレッド並列とプロセス並列によるハイブリッド並列プログラムを利用する場合や、一つのノードの中で複数のプロセスを動作させる場合、この推進手法による非ブロッキング集団通信の効果は、計算性能低下による計算時間の増加と、通信隠蔽による通信時間の減少のトレードオフとなる。しかし、Intel MPI Benchmarks [1] や OSU Micro-Benchmarks [2] のような、一般に用いられているベンチマークプログラムにおける非ブロッキング集団通信の評価では、通信隠蔽率のみを計測するため、この推進手法の実用性を十分に検証することが出来ない。

そこで本稿では、ハイブリッド並列計算と集団通信を並行して実行する簡単なベンチマークプログラムを作成し、それを用いて、プログレススレッドによる非ブロッキング集団通信の実用性を評価する。評価対象の MPI ライブラリとしては、本稿執筆時点で Mellanox 社の InfiniBand インターコネクトによる PC クラスタ上でプログレススレッドによる通信隠蔽効果が確認できた MVAPICH2-2.2 と Intel MPI 2017、さらに、Fujitsu PRIMEHPC FX100 でプログレススレッド専用のアシスタントコアを使用する富士通社製 MPI を用いる。また、Mellanox 社の SHArP [3] と呼ばれるオフロード機能についても、予備評価を行う。

本稿の主な寄与は、以下の通りである。

- ハイブリッド並列に対応し、MPI ライブラリ間の優劣の比較が可能な非ブロッキング集団通信ベンチマークプログラムの実装
- このベンチマークプログラムを用いた、プログレススレッドによる非ブロッキング集団通信のプログラム性能への効果の検証

2. 既存の非ブロッキング集団通信実装

2.1 非ブロッキング集団通信インタフェース

MPI-3.0 規格において採用された非ブロッキング集団通信インタフェースは、従来の一対一通信における非ブロッキング通信と同様、通信の開始と完了待ちをそれぞれ別の関数とすることで、通信完了を待つ間に、その通信と並行して処理可能な計算や通信の実行を可能とするものである [4]。通信開始関数としては、MPI_Iallreduce 関数や、MPI_Ialltoall 関数等、従来のブロッキング集団通信関数名に I を追加したものが用意されている。これらの関数は、完了を待つための情報を、MPI_Request 型のリクエスト変数に格納する。この変数は、一対一の非ブロッキング通信と同じ MPI_Wait 関数や MPI_Waitall 関数等に指定することで、完了を待つことが出来る。これらの関数を用いて通信の開始と完了待ちを指示し、さらにその通信と関係のない処理を開始と完了待ちの間に挿入することで、その通信の時間の一部を他の処理で隠蔽することが期待できる。

現在、MPICH [5]、MVAPICH2 [6]、Open MPI [7] といった主要なオープンソースの MPI ライブラリの他、Intel MPI Library [8] や富士通社製 MPI 等の非オープンソースの MPI ライブラリの多くで、この非ブロッキング集団通信インタフェースが提供されている。

2.2 非ブロッキング集団通信のアルゴリズム推進手法

非ブロッキング集団通信による通信隠蔽の効果は、MPI ライブラリでの実装手段に大きく影響を受ける。特に、集団通信アルゴリズムを他の処理と並行して進行させるためのアルゴリズム推進の実装手段は、通信隠蔽効果への影響が大きい。この、アルゴリズム推進手法は、集団通信アルゴリズムを構成する通信関数やメモリコピー、計算等の処理を、集団通信アルゴリズムで定義された依存関係に従って順に発行する。

現在、MPI ライブラリで採用されているアルゴリズム推進手法としては、主に以下の三通りが挙げられる。

- MPI 関数呼び出し毎の推進
- インターコネクトのオフロード機能による推進
- プログレススレッドによる推進

このうち、MPI 関数呼び出し毎の推進とは、全ての MPI 関数内で、非ブロッキング集団通信を推進させるための推進ルーチンを実行するものである。この推進ルーチンは、その時点で進行中の全ての非ブロッキング集団通信について、アルゴリズムの依存関係に基づき、実行可能な命令を発行する。この推進手法を利用する場合、プログラマは、非ブロッキング集団通信の開始関数と完了待ち関数の間に、例えば MPI_Test 関数のように、プログラムの意味を変えない MPI 関数をプログラム中に挿入することで、完了待ちの前にアルゴリズムを進行させておくことが出来る。この推進手法による性能向上の効果は、通信隠蔽による通信時間の削減と、推進ルーチンのためだけに呼び出す MPI 関数のオーバヘッドのトレードオフとなり、十分な効果が得られるか否かは、プログラムの構造やプログラマの能力に依存する。

一方、インターコネクトのオフロード機能による推進は、インターコネクトの Network Interface Card (NIC) やスイッチ等に集団通信のアルゴリズムを進行させるオフロード機能が用意されている場合に、非ブロッキング集団通信関数の内部でその機能を呼び出すものである。例えば Mellanox 社のインターコネクト技術である InfiniBand は、集団通信アルゴリズムを NIC で処理する CORE-Direct

機能や、スイッチで処理する SHArP 機能を備えている。また、これらの機能を非ブロッキング集団通信の内部で呼び出すよう実装されている MPI ライブラリとしては、MVAPICH2 や Open MPI、Mellanox 社の HPC-X [9]、等がある。基本的に、オフロード機能を有するインターコネクトが利用できる場合、効果的に通信を隠蔽することが出来る。ただし、この機能は、NIC やスイッチのハードウェア上の制約により、使用できる集団通信関数やメッセージサイズ等が制限されている場合が多い。

これらに対して プログレススレッドによる推進は、MPI プログラムを進行させるスレッドとは別に、非ブロッキング集団通信アルゴリズムを進行させるためだけのスレッドを生成する。この推進手法は、使用するインターコネクトがオフロード機能を有しない場合や、プログラム中に適切に `MPI_Test` 関数等を挿入することが困難な場合でも、集団通信と他の処理を同時に進行させることが出来るため、容易に通信を隠蔽する手段として注目されている。そのため、代表的なオープンソースの MPI ライブラリである Open MPI、MPICH、MVAPICH2 のいずれも、プログレススレッドによる非ブロッキング集団通信の推進を選択可能となっている。このうち Open MPI では、Hoefler らが開発した非ブロッキング集団通信ライブラリ LibNBC [10] をコンポーネントとして追加することにより、この推進手法を実装している。この推進手法は、図 1 に示す通り、メインスレッドが、ハンドルキューと呼ぶ待ち行列を介して、集団通信アルゴリズムを構成する処理をタスク単位でプログレススレッドに渡すことにより、アルゴリズムを推進させている。一方、MPICH および MVAPICH2 は、同様の待ち行列を持つ推進手法を独自に開発し、実装している。

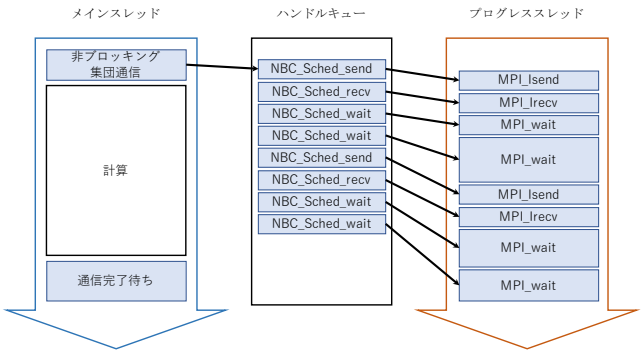


図 1　LibNBC におけるプログレススレッドによる推進機構

2.3　MVAPICH2 および Open MPI における非ブロッキング通信の実装

著者らは、過去の研究において、MVAPICH2 における非ブロッキング集団通信の通信性能と通信隠蔽効果を計測した [11]。その結果、プログレススレッドを使用したアルゴリズム推進機構を選択した場合、高い通信隠蔽率が得られることが分かった。ただし、プログレススレッドの使用を選択した場合、MPI の通信関数を排他的に実行するよう実装されており、この排他制御のための POSIX Thread の Mutex Lock 関数での待ち時間が通信性能を低下させることが分かっている。これは、本稿執筆時点の 2017 年 11 月における最新バージョンである MVAPICH2 2.2 においても、同様であることを確認している。

一方、Open MPI の最近のバージョンである Open MPI 2.0.2 について調査したところ、プログレススレッドが利用可能となるのは Ethernet を用いる場合のみであり、しかも、用意されているアルゴリズムが、アルゴリズム内の依存関係がほとんど無い Basic Linear アルゴリズムのみであることから、現時点では、プログレススレッドを使用することによる通信隠蔽の効果がほとんど期待できないことがわかった。なお、本稿執筆時点での最新バージョンは Open MPI 3.0.0 であり、このバージョンでの通信隠蔽の効果は現在検証中である。

2.4　プログレススレッドへの CPU コアの割り当て方法

非ブロッキング集団通信の使用にあたって利用者が選択すべき事項のうち、性能に与える影響が大きいものとしては、前節までで紹介したアルゴリズム推進手法の他に、プログレススレッドへの CPU コアの割り当て方法が挙げられる。この方法としては、Hoefler らの指摘 [12] にあるように、Spare Core と Fully Subscribed の二通りがある。

このうち Spare Core とは、プログレススレッドに一つの CPU コアを占有させるもので、高い通信隠蔽効果を期待できる。しかし、計算に割り当てる CPU コアが 1 プロセス当たり 1 コアずつ削減されるため、特に MPI のみによる並列プログラムや、ハイブリッド並列でプロセスあたりのスレッド数が少ない場合、計算性能の低下による影響が無視できなくなると予想される。

一方 Fully Subscribed は、計算用のスレッドに全ての CPU コアを割り当てておき、プログレススレッドにはどれかの計算用スレッドと CPU コアを共有させるものである。これは、計算スレッドの空き時間に集団通信アルゴリズムを進行させることを期待するもので、Spare Core と比べると、計算性能の低下による影響は少ないと期待できる。しかし、スレッド切り替えのオーバヘッドや、進行状況を確認する頻度の低下により、通信隠蔽効果は低下すると予想される。

なお、Intel MPI Library には、プログレススレッドに割り当てる CPU コアの番号を固定する機能が用意されている。これにより、ノード内のプロセス数が増えても、プログレススレッドに割り当てられる CPU コア数が増加せず、計算性能の低下を軽減する効果が期待できる。また、Fujitsu PRIMEHPC FX100 では、アシスタントコアと呼ばれる CPU コアをプログレススレッドに割り当てること

```
ts = MPI_Wtime();
MPI_Ialltoall();
MPI_Wait();
Tcomm = MPI_Wtime() - ts;

ts = MPI_Wtime();
MPI_Ialltoall();
tcs = MPI_Wtime();
while (MPI_Wtime() - tcs < Tcomm)
    dummy_comp();
Tcomp = MPI_Wtime() - tcs;
MPI_Wait();
Tall = MPI_Wtime() - ts;

overlap = 100 - ((Tall - Tcomp) / Tcomm) * 100;
```

図 2 OSU Micro Benchmarks の非ブロッキング集団通信ベンチマークプログラムにおける通信隠蔽率計測の概要

により、計算用の CPU コアを全て計算に使用することが出来る。

3. 非ブロッキング集団通信性能評価用ベンチマークプログラム

非ブロッキング集団通信を対象としたベンチマークプログラムとして一般的に用いられているのは、オハイオ州立大学で開発されている OSU Micro Benchmarks や、Intel 社で開発されている Intel MPI Benchmarks 等に含まれるプログラムである。これらのプログラムは、通信を隠蔽できた比率や、隠蔽できなかった通信オーバヘッドの計測を主な目的としている。そのため、通信と並行して実行する計算プログラムでは、予め計測していた通信時間を用いて、通信が行われている間に計算が終了しないよう、計算量を制御している。

図 2 に、OSU Micro Benchmarks の非ブロッキング集団通信ベンチマークプログラムにおける通信隠蔽率計測の概要を示す。このプログラムでは、最初に非ブロッキング集団通信の開始から完了までの時間（秒）を計測し、変数 `Tcomm` に格納する。その後、`MPI_Ialltoall` 関数で改めて非ブロッキング集団通信を開始し、`Tcomm` 秒が経過するまで、`dummy_comp()` という関数を繰り返し呼び出し、計算する。このようにして、十分に計算時間を費やした後に、`MPI_Wait` 関数で非ブロッキング集団通信の完了を待つ。この手順で得られた計測結果から、非ブロッキング通信で隠蔽できた通信時間と `Tcomm` の比率 `overlap` を、通信隠蔽率として出力する。

この手法は、通信隠蔽率の評価には適しているものの、以下の点から、この手法だけで実プログラムにおける非ブロッキング集団通信の効果を見積もることは困難である。

- 通信時間の実測値に応じて計測毎に計算量を自動調節するため、プログレススレッドによる計算性能低下の影響を評価できない。
- 計算部分がスレッド並列化されていないため、プログレススレッドの CPU コアへの割り当て方法による性能の変化を評価できない。
- 非ブロッキング集団通信を使わなかった場合の性能が計測されていないため、使用の有無によるプログラムの性能の変動を比較できない。

そこで本稿では、図 3 に示す簡単なプログラムにより、ハイブリッド並列プログラムにおける非ブロッキング集団通信の効果を計測する。このプログラムでは、通信量 `M` と計算量 `N` を実行時に指定し、これらのパラメータを使って、以下の時間をそれぞれ計測することにより、非ブロッキング集団通信使用の有無による並列プログラムの性能への影響の見積もりが可能となる。

`Tcomm` ：ブロッキング集団通信の所要時間
`Tnbcomm` ：非ブロッキング集団通信の所要時間
`Tcomp` ：計算時間
`Tblock` ：ブロッキング集団通信と計算を連続して実行した場合の所要時間
`Tovlp` ：非ブロッキング集団通信と計算を並行して実行した場合の所要時間

図 3 から呼ばれる計算関数 `do_comp_`S のプログラムを図 4 に示す。関数名、および関数内の `schedule` 節における S には、OpenMP におけるスケジューリングポリシーを記述する。今回の計測では、static と dynamic、それぞれのスケジューリングポリシーについて計測する。これは、特に Fully Subscribed において、スケジューリングポリシーによる性能の変化を確認するためである。

4. 性能評価

4.1 実験環境

4.1.1 ベンチマークプログラム

図 3 に示すベンチマークプログラムを利用し、非ブロッキング集団通信による並列プログラムの性能への影響を評価する。ベンチマークプログラムは C 言語で記述しており、MPI および OpenMP で並列化している。なお、プログラム中では、ウォームアップ処理として、プログラム開始直後に計測対象の通信関数を数回ずつ実行する。また、`Tcomm`、`Tnbcomm`、`Tcomp`、`Tblock`、`Tovlp` は、それぞれ 20 回ずつ連続して実行した平均値を測定結果とする。

また、計測対象の集団通信関数として図 3 に示した Alltoall 関数を用いるものの他に、Allreduce 関数を用いるものも用意し、計測する。いずれも、`M` 要素の倍精度実数をメッセージサイズとして指定する。また、Allreduce 関数では、集約操作として `MPI_SUM` を指定する。

```
get_param(&M, &N);
MPI_Comm_size(&procs);

ts = MPI_Wtime();
MPI_Alltoall(M);
Tcomm = MPI_Wtime() - ts;

ts = MPI_Wtime();
MPI_Ialltoall(M); MPI_Wait();
Tnbcomm = MPI_Wtime() - ts;

ts = MPI_Wtime();
do_comp_S(N, procs);
Tcomp = MPI_Wtime() - ts;

ts = MPI_Wtime();
MPI_Alltoall(M);
do_comp_S(N, procs);
Tblock = MPI_Wtime() - ts;

ts = MPI_Wtime();
MPI_Ialltoall(M);
do_comp_S(N, procs);
MPI_Wait();
Tovlp = MPI_Wtime() - ts;
```

図 3 ハイブリッド並列プログラムにおける非ブロッキング集団通信の効果を計測するベンチマークプログラム (Alltoall 版))

```
void do_comp_S(int N, int procs)
{
  int nn = N / procs;
#pragma omp parallel for private(j,k) schedule(S)
  for (i = 0; i < nn; i++)
    for (k = 0; k < N; k++)
      for (j = 0; j < N; j++)
        c[i*N+j] += a[i*N+k] * b[k*N+j];
}
```

図 4 do_comp_S 関数

4.1.2 使用計算機

実験に使用した計算機環境は、九州大学情報基盤研究開発センターの Fujitsu PRIMERGY CX400 と、名古屋大学情報基盤センターの Fujitsu PRIMEHPC FX100 である。それぞれの仕様を、表 1、2 に示す。

4.1.3 プログレススレッド使用の選択手段

Fujitsu PRIMERGY CX400 は、Intel 社製 CPU と Mellanox 社製インターコネクトによる PC クラスタであり、コンパイラや MPI ライブラリには様々な選択肢がある。今回は、測定した 2017 年 7 月の時点で、プログレススレッドを用いた非ブロッキング集団通信の通信隠蔽効果が確認できた MVAPICH2 2.2 および Intel MPI Library 2017 を MPI ライブラリとして使用する。また、C コンパイラとしては、MVAPICH2 を使用する場合は GCC を、Intel MPI Library を使用する場合は Intel C Compiler を、それぞれ選択する。

表 1 Fujitsu PRIMERGY CX400 の仕様

CPU	Intel Xeon E5-2680 (2.7GHz, 8core) x 2 / node
Memory	128GB / node
Interconnect	Mellanox InfiniBand FDR x 1
# of nodes	1476 (うち 32 ノード使用)
OS	Red Hat Linux Enterprise
C Compiler	GCC 4.4.6, Intel C Compiler 2017
MPI Library	MVAPICH2 2.2, Intel MPI Library 2017

表 2 Fujitsu PRIMEHPC FX100 の仕様

CPU	Fujitsu SPARC64 XIfx (2.2GHz, 32core) x 1 / node
Memory	32GB / node
Interconnect	Fujitsu Tofu2
# of nodes	2880 (うち 96 ノード使用)
OS	Fujitsu proprietary (Linux based)
C Compiler	Fujtisu C Compiler
MPI Library	Fujitsu MPI Library

MVAPICH2 では、プログレススレッドを有効にするための環境変数として、以下を指定する。

```
MV2_SMP_USE_CMA=0
MV2_ENABLE_AFFINITY=0
MV2_ASYNC_PROGRESS=1
```

一方、Intel MPI Library では、以下の環境変数指定によりプログレススレッドを有効にする。

```
I_MPI_ASYNC_PROGRESS=1
```

また、追加のオプションとして、以下の環境変数指定により、プログレススレッドを割り当てる CPU を固定することが出来る。そこで、このオプションの有無による性能の変化についても計測する。

```
I_MPI_ASYNC_PROGRESS_PIN=CPU 番号
```

一方、Fujitsu PRIMEHPC FX100 では、MPI ライブラリ、C コンパイラとも、富士通社製のものを使用する。この MPI ライブラリでは、`mpiexec` コマンドに以下のオプションを追加することにより、アシスタントコア上に非ブロッキング集団通信用のプログレススレッドを起動し、利

用できる。

```
--mca opal_progress_thread_mode モード番号
```

ここでモード番号とは、プログレススレッドの動作モードを指定するもので、以下から選択する。

mode 1 指定した区間のみプログレススレッドが動作する。区間内では MPI 関数を呼び出すことが出来ない。

mode 2 指定した区間のみプログレススレッドが動作する。区間内で MPI 関数を呼び出すことが出来る。

mode 3 非ブロッキング集団通信が呼ばれると自動的にプログレススレッドが動作する。

mode 1 および mode 2 の場合、図 3 の `MPI_Ialltoall` と `MPI_Wait` に挟まれた `do_comp_`S の呼び出し部分を、以下のようにプログレススレッドの動作区間として指定する。

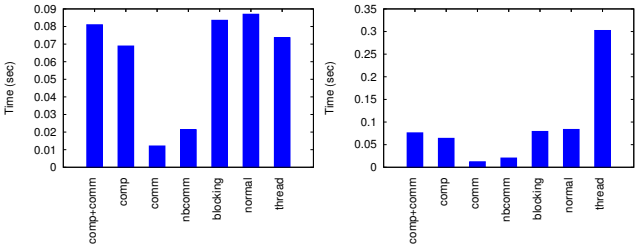
```
ts = MPI_Wtime();
MPI_Ialltoall(M);
FJMPI_Progress_start()
do_comp_S(N, procs);
FJMPI_Progress_stop()
MPI_Wait();
Tovlp = MPI_Wtime() - ts;
```

4.2 Fujitsu PRIMERGY CX400 における計測結果

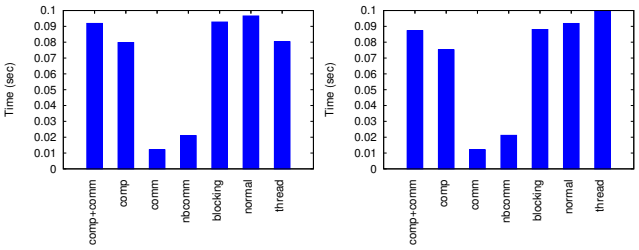
4.2.1 ノード内プロセス数が 1 の場合の Alltoall による計測

Fujitsu PRIMERGY CX400 で Alltoall 関数によるベンチマークプログラムの性能を、MVAPICH2 および Intel MPI Library で計測した結果を、それぞれ図 5 と図 6 に示す。どちらも、ノード内プロセス数を 1 とし、通信量と計算量のパラメータはそれぞれ M=131072 および N=1024 としている。

それぞれの図で、(a) と (c) は、プログレススレッドへの CPU コアの割り当て方法として Spare Core を選択した場合であり、OpenMP のスレッド数として、CPU コア数より 1 個少ない 15 スレッドを指定している。一方 (b) と (d) は、Fully Subscribed を選択した場合であり、OpenMP のスレッド数は 16 スレッドとしている。また、(a) と (b) は、ループのスケジューリングポリシーとして static を選択した場合であり、(c) と (d) は、dynamic を選択した場合である。図の横軸の項目は、プログラム中の `Tcomm`、`Tnbcomm`、`Tcomp`、`Tblock`、`Tovlp`、の計測結果をもとに、以下の通り算出したものである。

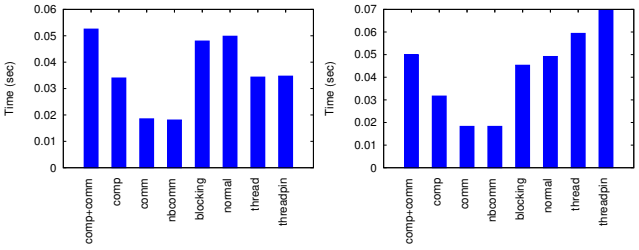


(a) Spare, Static (b) Full, Static

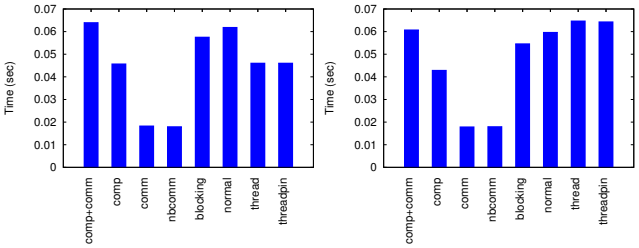


(c) Spare, Dynamic (d) Full, Dynamic

図 5 Alltoall with MVAPICH2 (32nodes, 32procs, M=131072, N=1024)



(a) Spare, Static (b) Full, Static



(c) Spare, Dynamic (d) Full, Dynamic

図 6 Alltoall with Intel MPI (32nodes, 32procs, M=131072, N=1024)

comp+comm = `Tcomm` + `Tcomp`
comp = `Tcomp`
comm = `Tcomm`
nbcomm = `Tnbcomm`
blocking = `Tblock`
normal = `Tovlp` (プログレススレッドを用いなかった場合)
thread = `Tovlp` (プログレススレッドを用いた場合)

また、threadpin は、Intel MPI Library で CPU を固定した場合の `Tovlp` である。

図 5 (a) および (c) より、MVAPICH2 では、Spare Core で CPU コアを割り当てた場合に、計算時間 comp とプログレススレッドを用いた場合の全体の時間 thread がほぼ

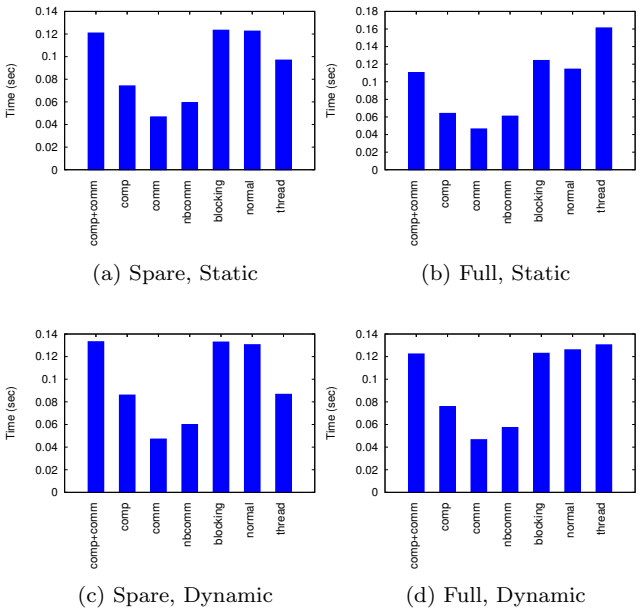
同じ値となっており、通信時間が隠蔽できていることが分かる。また、スケジューリングポリシーの選択による影響として、static の方が dynamic より計算時間 comp が若干短くなることが分かる。その結果、全体の性能としても、Spare Core で static スケジューリングを選択した場合が最も高速となった。

一方、図 5 (b) および (d) より、Full Subscribed では、非ブロッキング集団通信による性能向上は見られなかった。特に static スケジューリングでは、プログレススレッドを用いた場合に大幅な性能低下が見られた。これは、同じ CPU コアをプログレススレッドと計算スレッドで取り合うことによるコンテキストスイッチのオーバヘッド、および負荷の不均衡によるものと考えられる。

Intel MPI Library でも、図 6 の (a) から (d) により、MVAPICH2 と同様の傾向が見られる。なお、Intel MPI Library はソースが公開されていないため原因は不明であるものの、CPU コアの割り当て方法が Fully Subscribed でスケジューリングポリシーが Static の場合の性能低下は、MVAPICH2 ほど顕著ではない。

4.2.2 ノード内プロセス数が 2 および 4 の場合の Alltoall による評価

図 7 から図 10 に、ノード内プロセス数を 2 および 4 とした場合の MVAPICH2 と Intel MPI Library での計測結果を示す。

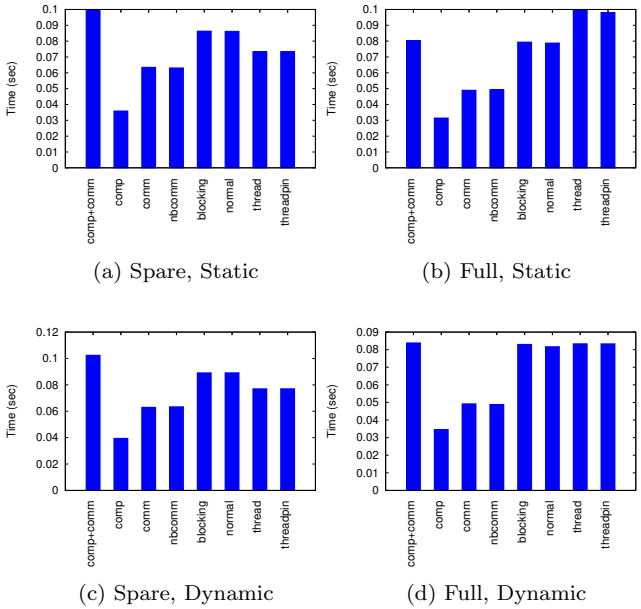


(a) Spare, Static　(b) Full, Static

(c) Spare, Dynamic　(d) Full, Dynamic

図 7 Alltoall with MVAPICH2 (32nodes, 64procs, M=131072, N=1024)

図 7 より MVAPICH2 では、ノード内プロセス数が 1 の場合と比べ、Static スケジューリングを選択した場合の通信隠蔽率が低下しており、その結果 Dynamic スケジューリングを選択した方が全体の性能が良くなっていることが分かる。この原因は不明であるものの、プロセス数の増加に伴って集団通信アルゴリズム内で発行する通信数が増加し、Spare Core でプログレススレッドが割り当てられた CPU コアでの負荷や待ち時間が増加したことが、通信隠蔽率の低下を招いた可能性が考えられる。これに対して Fully Subscribed では、プログレススレッドの待ち時間に計算スレッドが動作するため、プロセス数増に伴う通信隠蔽率への影響は少ないと考えられる。



(a) Spare, Static　(b) Full, Static

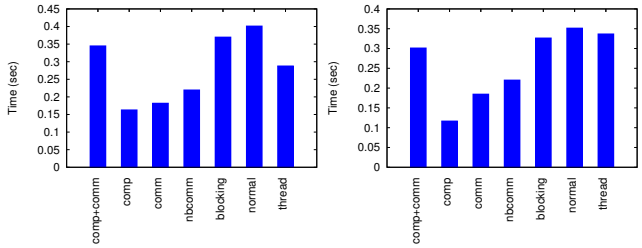
(c) Spare, Dynamic　(d) Full, Dynamic

図 8 Alltoall with Intel MPI (32nodes, 64procs, M=131072, N=1024)

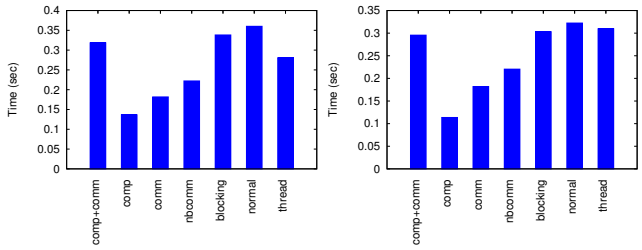
一方、Intel MPI Library では、図 8 に示す通り、MVAPICH2 を使用した場合に比べて計算時間が短くなるため、通信時間のほうが計算時間より長くなり、すべての通信時間を隠蔽することができなくなった。計算時間が短くなった原因は、コンパイラの違いによるものであると考えられる。それでも、Spare Core で CPU コアを割り当てた場合は、プログレススレッドを用いることで全体の性能を向上できているため、一部の通信時間を隠蔽できたことが分かる。また、Intel MPI Library では、スケジューリングポリシーとしては static を選択したほうが若干性能が良かった。

ノード内プロセス数を 4 とした場合、図 9、10 に示す通り、MVAPICH2 と Intel MPI Library のいずれも通信時間が計算時間を上回った。その結果、どちらの MPI ライブラリでも、図 8 と同様に、Spare Core で割り当てた場合に一部の通信時間を隠蔽できていることが分かる。

なお、Intel MPI Library で追加オプションとして指定できる、プログレススレッドを割り当てる CPU コアを固定する機能は、特にノード内プロセス数が増えた場合に、プログレススレッドが動作するコアを一つにまとめることで、計算スレッドに割り当てる CPU コアを増やす効果があると予想していた。しかし、本節での Intel MPI Library の結果から、threadpin の性能が thread と同等であり、少

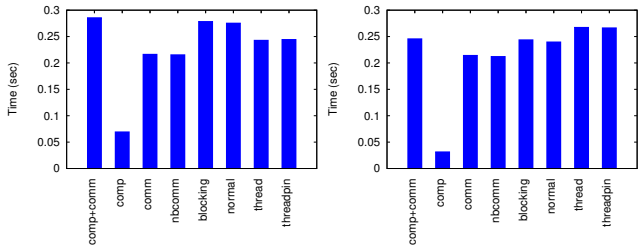


(a) Spare, Static (b) Full, Static

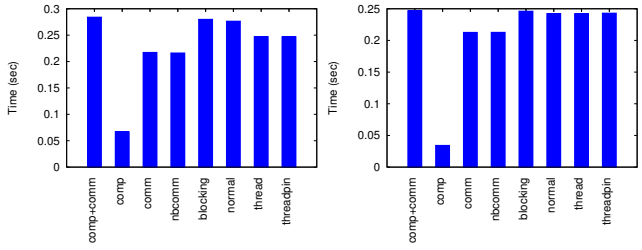


(c) Spare, Dynamic (d) Full, Dynamic

図 9 Alltoall with MVAPICH2 (32nodes, 128procs, M=131072, N=1024)



(a) Spare, Static (b) Full, Static



(c) Spare, Dynamic (d) Full, Dynamic

図 10 Alltoall with Intel MPI (32nodes, 128procs, M=131072, N=1024)

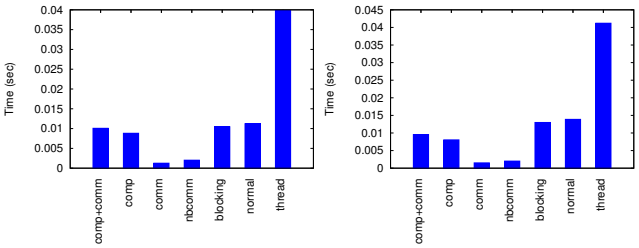
なくとも今回の実験では、期待した効果は確認できなかった。原因については、現在調査中である。

4.2.3 Allreduce の評価

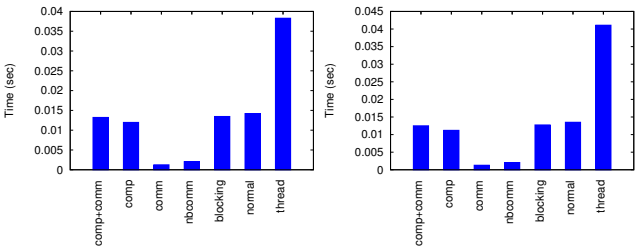
図 11 と図 11 に、1 ノード 1 プロセスでの、MVAPICH2 と Intel MPI Library による Allreduce のベンチマークプログラムの計測結果をそれぞれ示す。CPU コアの割り当て方法やスケジューリングポリシーの選択によらず、MVAPICH2 では、プログレススレッドによる非ブロッキング集団通信の実装では、性能が大幅に低下することが分かった。一方、Intel MPI Library では、Alltoall の場合と同様に、Spare Core で CPU コアに割り当てる場合に通信隠蔽の効果が得られること、およびスケジューリングポリシーとしては、static の方が dynamic より高速であることが分かった。

また、図 12 に、1 ノード 2 プロセスでの、Intel MPI Library による Allreduce のベンチマークプログラムの計測結果を示す。この場合は、MVAPICH2 の 1 ノード 1 プロセスの場合と同様、プログレススレッドによる非ブロッキング集団通信の実装で性能が大幅に低下する。

このように Allreduce は、Alltoall に比べ、プログレススレッドによる非ブロッキング集団通信の効果が得られにくくなっていることが分かる。これは、集団通信アルゴリズムに通信だけでなく計算も含んでいるため、プログレススレッドの負荷が大きいことが影響していると予想している。



(a) Spare, Static (b) Full, Static



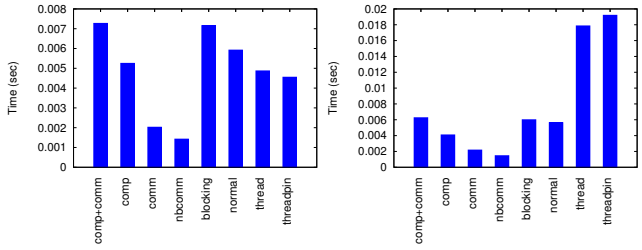
(c) Spare, Dynamic (d) Full, Dynamic

図 11 Allreduce with MVAPICH2 (32nodes, 32procs, M=131072, N=512)

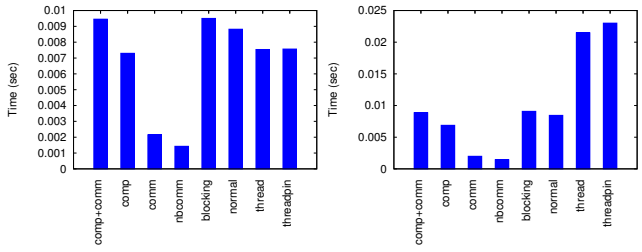
4.3 Fujitsu PRIMEHPC FX100 における計測結果

Fujitsu PRIMEHPC FX100 における計測結果として、ノード内プロセス数を 1、2、4 と増やした場合の、Alltoall および Allreduce のベンチマークの測定結果を、図 14、15、16、17、18、19 にそれぞれ示す。mode1, mode2, mode3 は、プログレススレッドの動作モードである。

これらの結果から、Alltoall では、ノード当たり 4 プロセスでも通信時間の少なくとも一部を隠蔽でき、非ブロッキング集団通信を用いない場合より性能を向上できていることが分かった。一方 Allreduce では、ノード内 2 プロセスまでは非ブロッキング集団通信の効果を確認できるものの、ノード内 4 プロセスでは性能が低下することが分かった。また、スケジューリングポリシーやプログレススレッドの動作モードについては、どれを選択しても性能に大きく影響しないことがわかった。

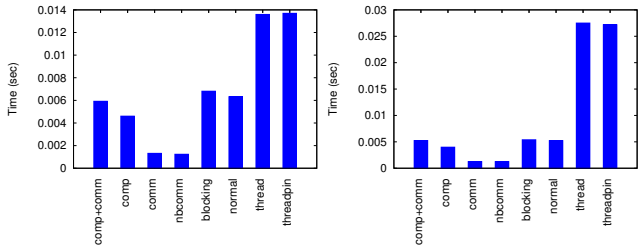


(a) Spare, Static (b) Full, Static

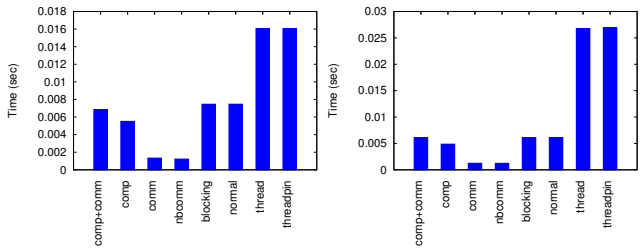


(c) Spare, Dynamic (d) Full, Dynamic

図 12 Allreduce with Intel MPI (32nodes, 32procs, M=131072, N=512)

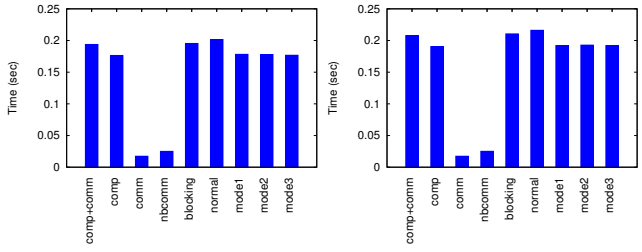


(a) Spare, Static (b) Full, Static



(c) Spare, Dynamic (d) Full, Dynamic

図 13 Allreduce with Intel MPI (32nodes, 64procs, M=131072, N=512)

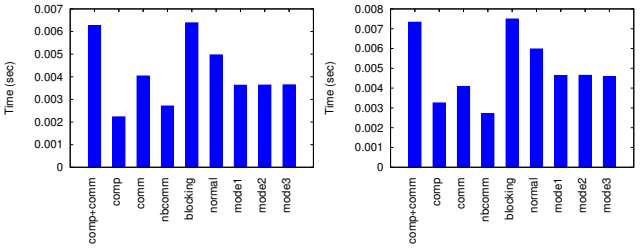


(a) Static (b) Dynamic

図 14 Alltoall on Fujitsu PRIMEHPC FX100 (48nodes, 48procs, M=131072, N=1536)

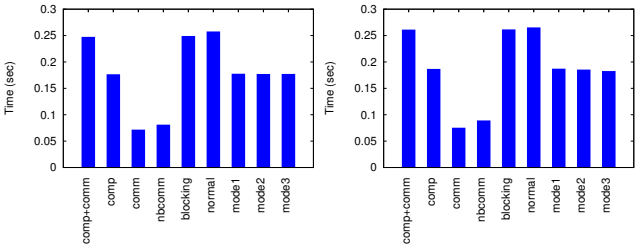
4.4 スイッチ装置へのオフロード機能に関する予備実験

2017 年 10 月に運用を開始した九州大学情報基盤研究開発センターのスーパーコンピュータシステム ITO では、インターコネクトネットワークに Mellanox 社製の InfiniBand EDR が用いられている。このインターコネクトネットワークでは、スイッチに集団通信アルゴリズムを実行させるオフロード機能である SHArP が提供されている。そこで、この機能による通信隠蔽効果を計測する予備実験を行った。



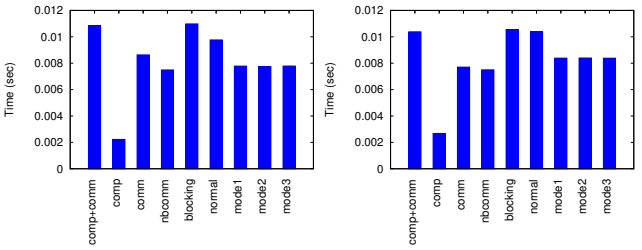
(a) Static (b) Dynamic

図 15 Allreduce on Fujitsu PRIMEHPC FX100 (48nodes, 48procs, M=131072, N=384)



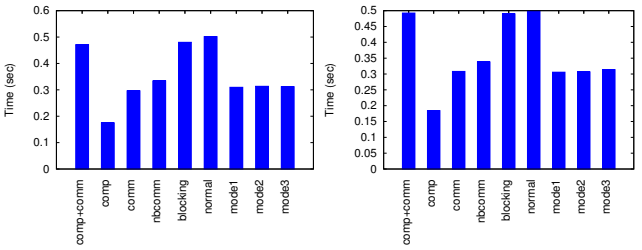
(a) Static (b) Dynamic

図 16 Alltoall on Fujitsu PRIMEHPC FX100 (48nodes, 96procs, M=131072, N=1536)



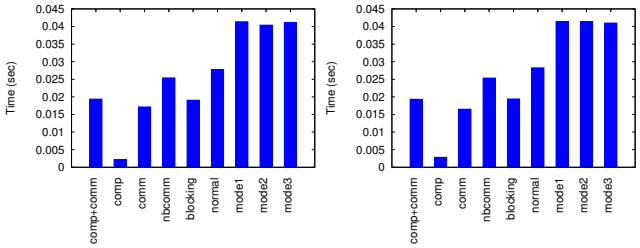
(a) Static (b) Dynamic

図 17 Allreduce on Fujitsu PRIMEHPC FX100 (48nodes, 96procs, M=131072, N=384)



(a) Static (b) Dynamic

図 18 Alltoall on Fujitsu PRIMEHPC FX100 (48nodes, 192procs, M=131072, N=1536)



(a) Static (b) Dynamic

図 19 Allreduce on Fujitsu PRIMEHPC FX100 (48nodes, 192procs, M=131072, N=384)

表 3 ITO のバックエンドサブシステム B の仕様

CPU	Intel Xeon Gold 6140 (2.3 - 3.7GHz, 18core) x 2 / node
Memory	384GB / node
Interconnect	Mellanox InfiniBand EDR x 2
# of nodes	128 (うち 16 ノード使用)
OS	Red Hat Linux Enterprise
C Compiler	GCC 4.4.6
MPI Library	Mellanox HPC-X 1.9.5

ITO のうち、今回の実験に使用したバックエンドサブシステム B の仕様を表 3 に示す。本稿執筆時点で、SHArP を利用する MPI ライブラリは、Open MPI をベースに Mellanox 社が開発した HPC-X 1.9.5 のみである。また、SHArP の使用にあたっては、`mpirun` コマンドのオプションに以下を追加した。

```
-x HCOLL_ENABLE_SHARP=2
-x HCOLL_ENABLE_NBC=1
-x HCOLL_MAIN_IB=mlx5_0:1
-x HCOLL_ENABLE_NBC_TOPO=1
-x HCOLL_POLLING_LEVEL=1
-x HCOLL_BCOL_P2P_NUM_TO_PROBE=1
-x SHARP_COLL_ENABLE_MCAST_TARGET=0
```

なお、最後の `SHARP_COLL_ENABLE_MCAST_TARGET` は、使用ノード数が少なく、一つのリーフスイッチ内に収まる場合に指定するものである。

計測に利用したプログラムは、図 3 のものとは違い、計算関数でハイブリッド並列化していないベクトル同士の加算を指定した回数行うものを用いた。また、SHArP で利用できる集団通信は Barrier と Allreduce のみであるため、今回は Allreduce について計測した。集約操作は、16 要素の倍精度実数の総和である。計測は、16 ノードで、ノード当たりプロセス数を 1 として行った。

計測結果を表 4 に示す。なお、比較のために、同じプログラムを Open MPI 3.0.0 で実行した際の結果も併記する。それぞれの計測結果は 1000 回連続して実行した平均値である。これにより、SHArP が集団通信そのものを高速化出来ていることと、及び、通信隠蔽によって性能を向上できていることが分かる。今後、ハイブリッド並列での性能検証を行う予定である。

表 4 SHArP を用いた非ブロッキング集団通信の通信隠蔽効果

	HPC-X 1.9.5	Open MPI 3.0.0
comm	3.58	6.27
nbcomm	3.92	12.2
comp	20.5	20.3
ovlp	22.3	27.9

(micro sec)

5. 考察

プログレススレッドを用いた非ブロッキング集団通信の全体的な傾向として、Fujitsu PRIMERGY CX400、Fujitsu PRIMEHPC FX100 のいずれも、Alltoall であれば通信隠蔽による性能向上が期待できることが分かった。一方、Allreduce では、使用する MPI ライブラリやハードウェアの特性により、性能向上が得られる場合があるものの、特にノード内プロセス数が増加すると、非ブロッキング集団通信を用いない場合よりも性能が低下することが分かった。また、プログレススレッドへの CPU コアの割り当て方法については、Spare Core やアシスタントコアなど、専用のコアを割り当てなければ、通信隠蔽の効果が期待できないことが分かった。

スレッドのスケジューリングポリシーについては、計算時間は、Static の方が若干早い場合があることがわかった。原因について、Fujitsu PRIMEHPC FX100 の性能解析ツールを用いて挙動を解析した結果、dynamic では static よりもキャッシュミス率が増加する傾向が見られた。これは、Dynamic では連続したループが同じコアに割り当てられない場合が多いためと考えられる。一方、ノード内プロセス数を増加させたり Fully Subscribed で CPU コアを割り当てたりすると、dynamic の方が static よりも性能が良くなる場合がある。これは、通信待ちや、負荷の不均衡で生じた CPU コアの空き時間を、Dynamic スケジューリングにより計算スレッドに割り当てることが出来た、という可能性が考えられる。

これらの結果から、非ブロッキング集団通信を用いる場合は、通信隠蔽による性能向上が見込めるか否か、事前に十分調査する必要が有ることがわかった。非ブロッキング集団通信で通信を隠蔽するには、通信と並行して進めることの出来る計算を分離する必要があるため、通常、アルゴリズムの修正を含む大幅なプログラムの修正が必要である。そのため、修正に着手する前に、想定される通信の種類やメッセージサイズ、さらに並行して進める計算プログラムの計算量やスレッド並列性を見積もり、それに近いベンチマークプログラムを使って効果を検証しておくことが推奨される。

6. むすび

本稿では、プログレススレッドを用いた非ブロッキング集団通信について、並列プログラムの性能への影響を計測し、実用性を検証した。計測には、プログレススレッドへの CPU コアの割り当て方法や、スレッドスケジューリングポリシー等の影響も評価できるように、ハイブリッド並列のベンチマークプログラムを作成し、使用した。このプログラムは、並行して実行する計算と通信の量をそれぞれ明示的に指定するため、プロセス数とスレッド数の比率など、実行条件を変えた場合の性能への影響も検証できる。

このプログラムを用いて、Fujitsu PRIMERGY CX400 および Fujitsu PRIMEHPC FX100 において、Alltoall と Allreduce を、プログレススレッドを用いた非ブロッキング集団通信とした場合の効果を評価した。その結果、使用する MPI ライブラリやプログレススレッドへの CPU コアの割り当て方法の調整により、特に Alltoall では、性能向上が見込めることを確認した。一方 Allreduce では、プログレススレッドへの負荷が大きいため、効果が得られる可能性が低くなり、かえって性能が低下する場合があることも分かった。

また、Mellanox 社のオフロード機能 SHArP を用いた通信隠蔽効果の予備実験も行い、性能向上が見込めることを確認した。

今後は、計算機環境や通信関数、メッセージサイズ、計算の種類等について、より幅広い調査を行い、プログラマが非ブロッキング集団通信の使用を選択する際の条件を明らかにしたい。さらに、Mellanox 社の CORE Direct 機能や SHArP 機能のように、インターコネクトネットワークの NIC やスイッチに集団通信を推進させるオフロード機能についても同様の調査を行い、実用性を検証したい。

謝辞

本研究の成果は、名古屋大学 HPC 計算科学連携研究プロジェクトの支援によるものである。また、本研究の実験には、名古屋大学情報基盤センターの Fujitsu PRIMEHPC FX100 ならびに九州大学情報基盤研究開発センターの Fujitsu PRIMERGY CX400 および ITO システムを利用した。

参考文献

[1] Intel MPI Benchmarks. https://software.intel.com/en-us/imb-user-guide
[2] OSU Micro-Benchmarks. http://mvapich.cse.ohio-state.edu/benchmarks/
[3] Richard L. Graham, Devendar Bureddy, Pak Lui, Hal Rosenstock, Gilad Shainer, Gil Bloch, Dror Goldenerg, Mike Dubman, Sasha Kotchubievsky, Vladimir Koushnir, Lion Levi, Alex Margolin, Tamir Ronen, Alexander Shpiner, Oded Wertheim and Eitan Zahavi. Scalable Hierarchical Aggregation Protocol (SHArP): A Hardware Architecture for Efficient Data Reduction. *Proceedings of the First Workshop on Optimization of Communication in HPC*, pp. 1–10, 2016.
[4] MPI-3.0 Draft. https://www.mpi-forum.org/
[5] MPICH. http://mvapich.cse.ohio-state.edu/
[6] MVAPICH2. https://www.open-mpi.org/
[7] Open MPI. https://www.open-mpi.org/
[8] Intel MPI Library. https://software.intel.com/en-us/intel-mpi-library
[9] Mellanox HPC-X http://www.mellanox.com/page/hpcx_overview
[10] Torsten Hoefler, Andrew Lumsdaine, and Wolfgang Rehm. Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI. *Proceedings of the 2007 ACM/IEEE conference on Supercomputing - SC '07*, p. 1, 2007.
[11] 成林晃, 南里豪志, 天野浩文. progress thread を用いた非ブロッキング集団通信の性能調査. 第 155 回ハイパフォーマンスコンピューティング研究会, 2016.
[12] Torsten Hoefler and Andrew Lumsdaine. Message progression in parallel computing - To thread or not to thread? *Proceedings - IEEE International Conference on Cluster Computing, ICCC*, Vol. Proceeding, pp. 213–222, 2008.