Vol.2016-HPC-155 No.34
2016/8/10

progress threadを用いた 非ブロッキング集団通信の性能調査

成林 晃[1,a)]　南里 豪志[2,b)]　天野 浩文[2,1,c)]

概要：近年，大規模化する並列計算機において，集団通信による通信時間の影響を減らす手法として， 非ブロッキング集団通信による通信時間を計算時間で隠蔽する高速化手法が注目されている．非ブロッキング集団通信において，計算と並行して通信を進めるための実装の一つに，スレッドを用いる手法がある．これは progress thread と呼ばれる，通信専用のスレッドを利用するものであり，この手法は他の実装に比べ，利用が容易で，かつ通信を隠蔽しやすいという特徴がある．現在，この手法による非ブロッキング集団通信の実装を選択できる MPI ライブラリはいくつかあるものの，通信隠蔽の効果が十分検証されておらず，実用性が不明である．そこで本稿では，それらの実装による通信時間隠蔽の効果を調査した．実験の結果，メッセージサイズが小さいと progress thread を利用することによるオーバヘッドによりかえって遅くなる場合がある，等の傾向を確認できた．

1. はじめに

現在の大規模並列計算機は，並列度を増大させることでより高い計算能力を達成している．大規模並列計算機では，分散メモリ型アーキテクチャが採用されており，大規模並列計算機を構成するノードと呼ばれる計算機間で通信が必要なため，性能向上を阻害している．この大規模並列計算機上で行われる通信のインタフェースとして事実上の標準となってるのが Message Passing Interface (MPI) である．MPI では，2 プロセス間で行う一対一通信の他に，多数のプロセス間で相互に行う定形の通信である集団通信が定義されている．プロセス数の増加に伴い，集団通信に要する時間が増加し，プログラム全体の所要時間に対する影響が増すため，大規模並列計算機では集団通信の所要時間削減が重要な課題となっている．

そこで，集団通信の所要時間を削減する手段の一つとして，非ブロッキング集団通信が MPI 規格のバージョン 3.0 において定義された [1]．非ブロッキング集団通信とは，データの送受信の完了を待たずに次の処理を開始することで，通信を別の処理によって隠蔽し，見かけ上の通信時間を短縮する集団通信である．このため非ブロッキング集団通信の実装には，集団通信の所要時間を短縮することだけでなく，他の処理による集団通信の隠蔽効果も求められる．MPI の規格を実装した MPI ライブラリは多数開発されており，同じ機能の実現にも，それぞれ異なる技術が用いられている．このうち非ブロッキング集団通信は，比較的新しいインタフェースであるため，上記の要求を満たすための改良が，各ライブラリで進められている．その中で，集団通信を隠蔽する手段の一つとして，progress thread を用いる技術が幾つかのライブラリで採用されている．これは通信に関わる処理を別スレッドで実行することで，高い通信の隠蔽を図ろうとするものである．しかし，progress thread を利用する場合，通信隠蔽の効果は期待できるものの，各プロセスに一つずつ progress thread が起動されるので，計算時間への影響が無視できない．特に，MPI とスレッド並列によるハイブリッド並列プログラムの場合，progress thread に一つのコアを専有させるか，もしくは他の計算スレッドとコアを共有させるかによって，性能が大きく変わる．

そこで本稿では, hybrid 並列プログラムにおいて progress thread を起動して性能を計測し，どういった場合に通信隠蔽の効果を得られるのか解析を行う．Hybrid 並列プログラムでは，ノード内に立ち上げるプロセス数，スレッド数に複数の組み合わせがあるため，各組み合わせについて通信隠蔽の効果を確認する．更に，得られた結果から計算時

[1] 九州大学システム情報科学府
Graduate School / Faculty of Information Science and Electrical Engineering, Kyushu University
[2] 九州大学情報基盤研究開発センター
Research Institute for Information Technology, Kyushu University
[a)] 2ie15092r@s.kyushu-u.ac.jp
[b)] nanri@cc.kyushu-u.ac.jp
[c)] amano@cc.kyushu-u.ac.jp

間，通信のオーバラップ率に基づき原因を考察する．そして今後利用者が progress thread を利用すべきか否か，また利用する場合は progress thread 専用の CPU コアを割り当てるべきか否か，適切な選択を支援するための情報を提供する．

2. progress thread を用いた非ブロッキング集団通信の実装例

本稿では評価対象として MVAPICH2 を用いるため，ここでは MVAPICH2 における progress thread による非ブロッキング集団通信の実装について概要を示す．

非ブロッキング集団通信関数は，通信を開始する関数と，完了を待つ関数の二つで成り立っている．一方，集団通信を実装するアルゴリズムの多くは，冗長な通信を削減するために，内部の一対一通信どうしに依存関係がある．そのため，非ブロッキング集団通信の開始関数で，全ての一対一通信を発行することが出来ない．

そこで MVAPICH2 における progress thread を用いた非ブロッキング集団通信実装では，開始関数の中で集団通信アルゴリズムを送信，受信，計算の各命令に分割し，さらに相互の依存関係を示す同期命令を追加した命令列を作成してタスクキューに格納する．一方 progress thread はこのタスクキューを監視しており，依存関係が解決して実行可能となった命令を処理する．これにより，集団通信開始関数を呼んだスレッドが他の処理を実行している間に，集団通信アルゴリズムを進行させることが出来るため，高い通信隠蔽効果が期待できる．

3. プログラムの性能に対する progress thread の影響

progress thread は，前述した通り高い通信の隠蔽率を期待できるものの，集団通信を個々の通信命令に分割して処理するため，通信時間が増加する．さらに，このスレッドを実行する CPU コアが必要であり，CPU コアの割り当て方に応じて，計算時間および通信時間が変化する．そのため，これらの影響を考慮した上で，progress thread を利用するか否か，また，利用する場合の progress thread への CPU コアの割り当て方を選択する必要がある．

Hoefler らの指摘にもある通り，progress thread を CPU コアに割り当てる方法としては，spare core を使用するものと使用しないもの (fully subscribed) が考えられる [2]．ここで spare core とは，progress thread のために一つの CPU コアを占有で割り当てるものである．spare core を使用する場合，progress thread は MPI のプロセス毎に一つ起動されるため，spare core もプロセスと同じ数が必要となる．つまり，計算スレッドが使用できる CPU コア数は，全 CPU コア数から spare core の数を減じた数であるため，その分計算時間が増加する．fully subscribed の場合，progress thread は，計算を行っているスレッドと CPU コアを共有する．そのため，タスクキュー内で発行可能となった命令を progress thread が即座に発行できない場合があり，通信時間が増加する可能性がある．また，スレッドの切り替えによって更なるオーバーヘッドが生じる場合があり，これもまた通信時間，計算への影響を与える可能性がある．

表 1　性能解析に用いるパラメータ

n_node	ノード数
n_cpu	ノード内コア数
t_node	ノード内合計スレッド数
p_node	ノード内プロセス数
t_calc	プロセスあたりの計算スレッド数

ところで，一般的な並列プログラムでは，MPI による並列化のみならず，スレッドベースの並列化を併用した hybrid 並列プログラミングが用いられることが多い．hybrid 並列プログラミングにおいて，利用するノード数を一定としても，立ち上げるプロセス数，スレッド数には複数の組み合わせが存在する．プロセス数とスレッド数の組み合わせによって，MPI による並列化では一対一である spare core の数と計算スレッドの数の比を変更することが可能である．これは，前述した問題点を持つ progress thread にとって，1 コアを progress thread のための spare core，残りのコアを計算スレッドに割り当てるといったことが可能となり，progress thread の問題点を解消できる可能性を持つ．しかしながら，これらについて調査を行った例は未だなく，計算性能と通信の隠蔽効果の両方を得るためにはのどのようなパラメータ選択をすべきか不明である．

そこで，本稿では hybrid 並列プログラムにおいて progress thread を利用するか否か，利用するのであれば spare core を使用すべきか否かも含め提示するため，通信隠蔽の効果，通信時間，計算への影響を調査し，progress thread を利用すべき調査した．

4. 実験

4.1 非ブロッキング集団通信における各パラメータの定義

本章では，ハイブリッド並列プログラムの性能に対する progress thread を用いた非ブロッキング集団通信の影響を計測し，解析する．そこでまず，性能解析に用いるパラメータを表 1 に定義する．

例えば $n_{cpu} = 8$, $t_{node} = 10$, $p_{node} = 2$, $t_{calc} = 4$ の場合，各ノードでは，図 1 に示すように spare core は確保されず，progress thread は計算スレッドと CPU コアを共有する．そのため，progress thread と計算スレッドが切り替わる際，コンテキストスイッチの時間が必要となる．

4.2 実験環境

本稿の実験には，九州大学情報基盤研究開発センター

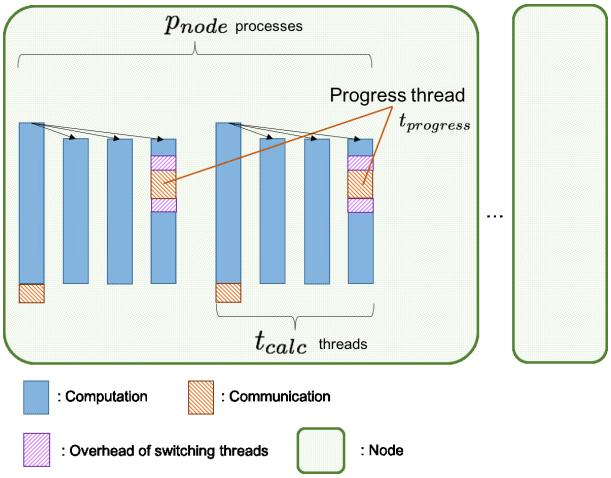

図 1　Hybrid 並列プログラミングにおいて 8CPU コアを 1 ノードに持つ並列計算機において，progress thread を利用した場合のモデル図
($p_{node} = 2$,progress thread:enable,
$t_{progress} = 2, t_{calc} = 4, t_{node} = 10$)

の Fujitsu PRIMERGY CX400 のうち 128 ノードを利用した．このシステムの各ノードには，CPU コア n_{cpu} が 16 個搭載されている．コンパイラは gcc 4.4.6 を利用した．

また，実験に用いた MPI ライブラリは，非ブロッキング集団通信において progress thread 機能が選択可能である MVAPICH2 2.2rc1 を利用した．また，MPI ライブラリにおいては，集団通信アルゴリズムによる性能変動を確認するため，複数のアルゴリズムが実装されている MPI_Iallgather 関数について，アルゴリズム選択を行わないようにし，あらゆるメッセージサイズ，プロセス数に関わらず常にひとつのアルゴリズムを選択するように変更を行った．

ベンチマークプログラムには，OSU Micro-Benchmarks 5.3 内の，MPI_Iallgather の通信オーバーラップ率を計測するプログラムに対して以下の変更を加えたものを用いた．まず，集団通信と並行して実行数 r 擬似計算について OpenMP でスレッド並列化した．また，計算量が通信時間によって変動しないよう，擬似計算の計算量を固定した．ここでの擬似計算は，メッセージサイズが 8192byte を境に変化させており，小メッセージサイズでは計算量を少なく，大メッセージサイズでは計算量を多くしている．

4.3　実験内容

progress thread の利用による通信隠蔽の効果を検証するため，1 ノードあたりに与える計算量を一定としながら，ノード内に立ち上げるプロセス数，OpenMP によるスレッド数を変化させ計算処理にかかる時間，通信時間，そして非ブロッキング集団通信のオーバーラップ率，全体の実行時間を測定した．図 2 に今回のベンチマークプログラムにおける測定の流れを示す．まず初めに，非ブロッキング集団通信関数を呼び出し，通信隠蔽のための別の処理なしに MPI_Wait 関数を呼び出す．この時間によって，非ブロッキング集団通信における純粋な通信に掛かる時間を計測する．この計測の後，通信の隠蔽を含めた全体の実行時間の測定を行う．OpenMP によるスレッド並列は，図 2 における dummy_compute 関数内部で行うようにしている．dummy_compute 関数は，3 つの配列を用意し，次の式のような計算を行う．

$$x[i] = x[i] + a[i] * a[i] + y[i] \quad (1)$$

配列の大きさ N は，1 プロセス当たりメッセージサイズが 8192byte 以下の時 $N = n_{cpu} * 100000/p_{node}$，メッセージサイズが 8192byte 以上の時 $N = n_{cpu} * 5000000/p_{node}$ とした．

プロセスあたりの計算スレッド数 t_{calc} は，spare core を用いる場合，

$$t_{calc} = n_{cpu}/p_{node} - 1 \quad (2)$$

```
for(i=0; i<iter; i++) {
    t_start = MPI_Wtime();
    MPI_Iallgather();
    MPI_Wait();

    t_stop = MPI_Wtime();

    pure_comm_total += t_stop-t_start;
    MPI_Barrier();
}

for(i=0; i<iter; i++) {
    t_start = MPI_Wtime();

    init_time = MPI_Wtime();
    MPI_Iallgather();
    init_time = MPI_Wtime() - init_time;

    tcomp = MPI_Wtime();
    dummy_compute();
    tcomp = MPI_Wtime() - tcomp;

    wait_time = MPI_Wtime();
    MPI_Wait();
    wait_time = MPI_Wtime() - wait_time;

    t_stop = MPI_Wtime();

    overall_timer += t_stop - t_start;
    tcomp_total += tcomp;
    init_total += init_time;
    wait_total += wait_time;
    MPI_Barrier();
}

MPI_Barrier();
```

図 2　ベンチマークプログラムにおける測定の流れ

progress thread を利用しない場合，また fully subscribed で実行する場合は

$$t_{calc} = n_{cpu}/p_{node} \tag{3}$$

とした．

また，ノード内プロセス数 p_{node} としては，1 から $n_{cpu}/2$ まで 2 のべき乗で変化させ，実験した．また，メッセージサイズとしては 1byte から 4Mbyte まで 2 のべき乗で変化させた．

4.4　実験結果

4.4.1　progress thread による通信隠蔽の効果

progress thread を利用しない場合，spare core，fully subscribed それぞれについて，通信隠蔽の効果を測定した．$n_node = 128$, $p_node = 1$ の場合の Recursive Doubling による MPI_Iallgather の通信オーバラップ率について，メッセージサイズが 8,192byte 未満のものを図 3 に，8,192byte 以上のものを 図 3 に，それぞれ示す．ここで No progress は progress thread を用いない場合，Spare core は progress thread を用いて spare core も利用した場合，Fully subscribed は progress thread を用いて spare core を利用しない場合のオーバラップ率である．

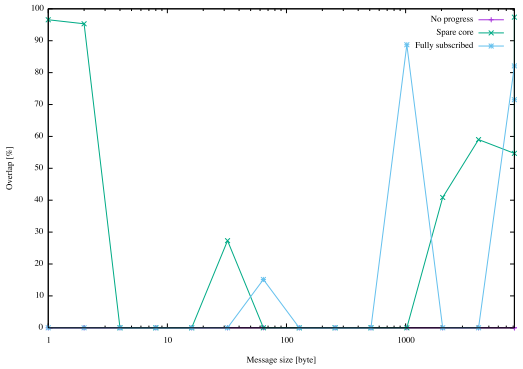

図 3　8,192byte 以下の MPI_Iallgather のオーバラップ率 ($n_node = 128$, $p_node = 1$, Recursive Doubling)

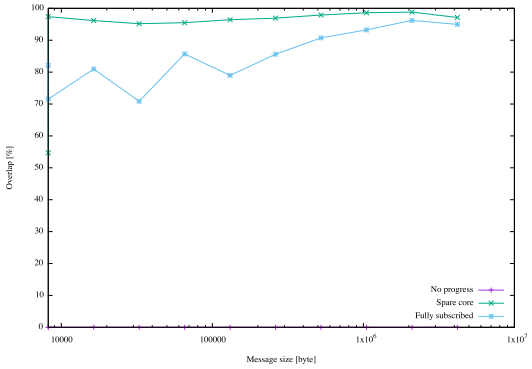

図 4　8,192byte 以上の MPI_Iallgather のオーバラップ率 ($n_node = 128$, $p_node = 1$, Recursive Doubling)

4.4.2 progress thread による計算性能への影響

progress thread による計算性能への影響として progress thread を用いない場合，progress thread を用いて spare core も用いる場合の性能の各 p_{node} 毎の dummy_compute 関数の実行時間を図 5，6 に示す．なお，fully subscribed の場合，今回の計測では非常に計測結果の変動が大きかったため，結果を図に示していない．ただし，fully subscribed の結果は，常に他の二つの場合よりも大幅に計算時間が大きくなった．ノード内プロセス数が 1，2，4 の場合には Spare core では，progress thread を利用しない場合に比べ計算時間が大きくなることが確認できた．これは，ノード内プロセス数の数だけ計算を処理するスレッド数が少なくなり，計算に時間がかかるためであると考えられる．なお，ノード内プロセスが 8 の時，spare core では実行時間が極端に短いことが確認できた．この原因は，現在調査を進めている．

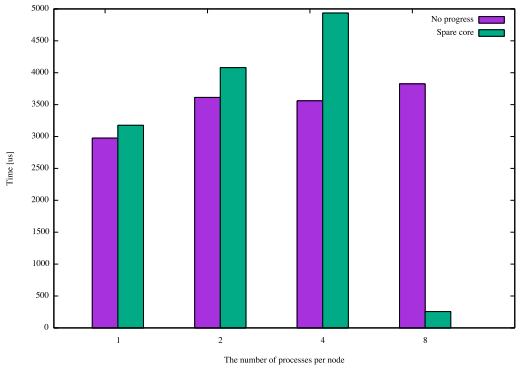

図 5　8,192byte 以下での計算時間　($n_node = 128$, $p_node = 1, 2, 4, 8$, Recursive Doubling)

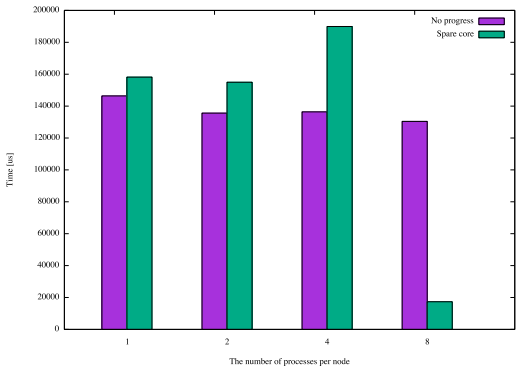

図 6　8,192byte 以上での計算時間　($n_node = 128$, $p_node = 1, 2, 4, 8$, Recursive Doubling)

4.4.3 progress thread 利用による通信性能の変化

通信時間への progress thread の影響について，アルゴリズムとして Ring 及び Recursive doubling を用いた場合の計測結果を図 7，8 にそれぞれ示す．

どちらのアルゴリズムでも，メッセージサイズが小さい

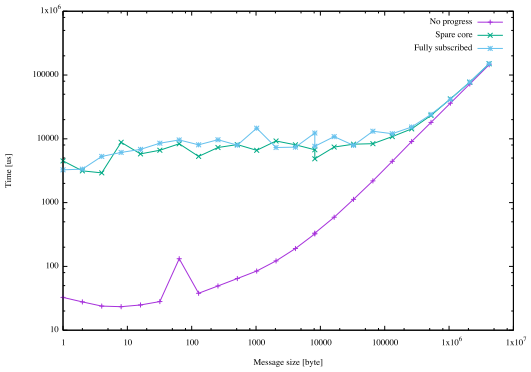

図 7　MPI_Iallgather の通信時間　($n_node = 128$, $p_node = 1$, Recursive Doubling)

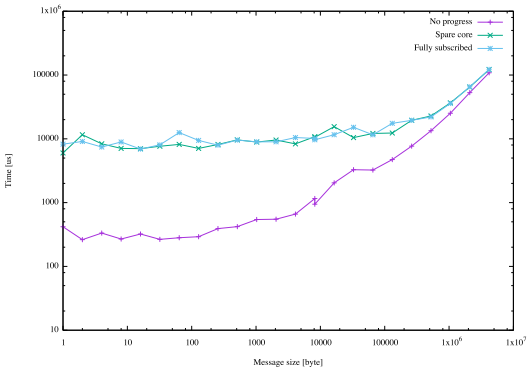

図 8　MPI_Iallgather の通信時間　($n_node = 128$, $p_node = 1$, Ring)

場合，spare core，fully subscribed 共に通信性能が著しく低下していることが判明した．これは，集団通信を一対一通信によって実現するための通信命令や同期の命令を格納するタスクキューの操作に伴うオーバーヘッドが原因である．progress thread を利用する場合，タスクキューへ計算スレッド，progress thread 双方からのアクセスがあり，排他制御を必要とする．その結果，非ブロッキング集団通信で progress thread を用いる場合，Recursive doubling のような小さいメッセージサイズ向けアルゴリズムは不利であることがわかった．

4.4.4 ノード内プロセス数と全体の実行時間の関係

ノード内プロセス数を変化させた場合の，計算と通信を合わせた全体の実行時間を図 9，10，11，12，13，14 に示す．

progress thread を利用する場合，メッセージサイズが 8,192byte 以上の場合，spare core でノード内プロセス数を 2 とする場合が最も早くプログラムを実行できることが判明した．一方，fully subscribed では，CPU コア数以上のスレッドを処理する必要がありオーバーヘッドが生じる．このため，spare core より高速に実行する場合は確認できなかった．progress thread を用いない場合，オーバーラップ率の面で spare core より大きく劣るため，MPI_Wait 関数にかかる時間が多く，全体の実行時間では spare core よりも性能が劣ることが確認された．

ノード内プロセス数を変化させた時，spare core でノード内プロセス数を 4 とした場合が遅いことが確認できた．これは他の場合に比べ，spare core をより多くノード内に確保する必要があり，計算性能が低下するためであると考えられる．また，spare core でノード内プロセス数を 2 とした場合に比べ，spare core でノード内プロセス数を 1 とした場合が僅かながら遅くなることが確認された．全体の実行時間の内訳を見ると，spare core でノード内プロセス数を 1 とした場合は計算時間が多く，通信時間は短いという事がわかった．これは，スレッド並列においてスレッド数が多い場合，並列化の効率が落ちているためだと考えられる．

また，今回の実験では利用できるノード数を固定して実験を行ったため，ノード内プロセス数が多いほど全体のプロセス数が増加する．このため，同じメッセージサイズでもノード内プロセス数が小さい方が有利である．

さらに，メッセージサイズが 8,192byte 以上の実験において，ノード内プロセス数が大きくなるにつれ，測定したメッセージサイズの上限が小さくなることが確認できる．これはベンチマークプログラムにメモリの制約が付加されているためで，利用可能なメモリに対し，通信のためのメモリ領域が不足する場合は測定を行わないよう実装されている．メモリの面からも，ノード内プロセス数は小さくすることが望ましいと言える．

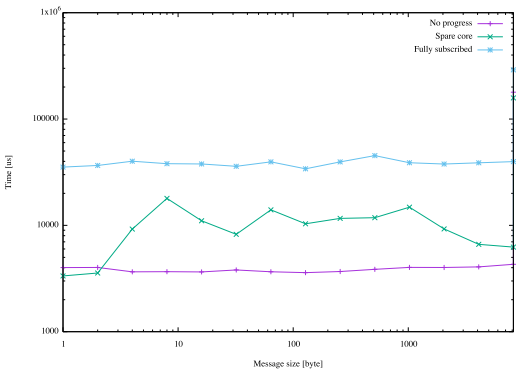

図 9 8,192byte 以下での MPI_Iallgather の全体の所要時間 ($n_node = 128$, $p_node = 1$, Recursive Doubling)

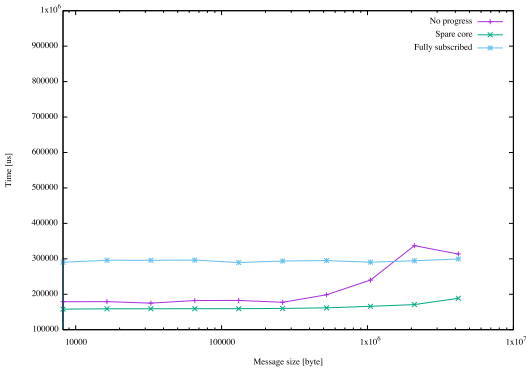

図 10 8,192byte 以上での MPI_Iallgather の全体の所要時間 ($n_node = 128$, $p_node = 1$, Recursive Doubling)

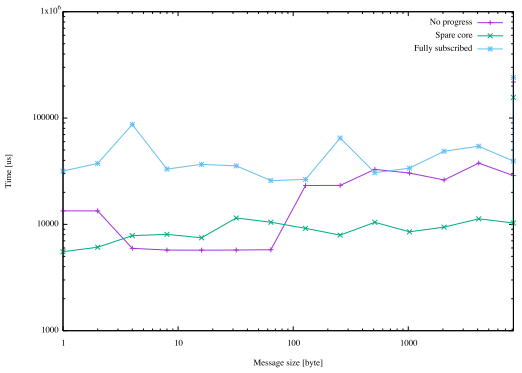

図 11 8,192byte 以下での MPI_Iallgather の全体の所要時間 ($n_node = 128$, $p_node = 2$, Recursive Doubling)

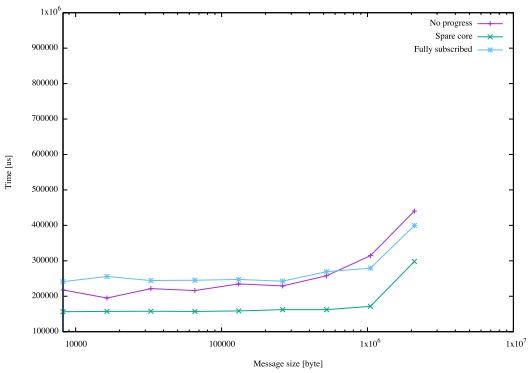

図 12 8,192byte 以上での MPI_Iallgather の全体の所要時間 ($n_node = 128$, $p_node = 2$, Recursive Doubling)

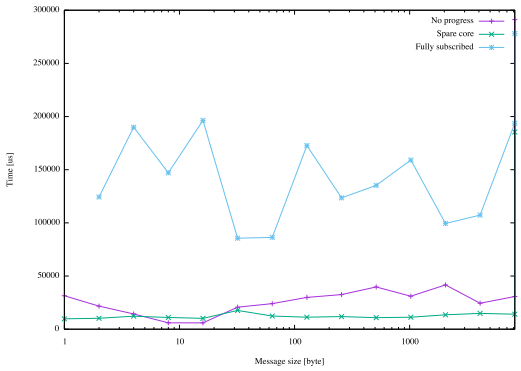

図 13 8,192byte 以下での MPI_Iallgather の全体の所要時間 ($n_node = 128$, $p_node = 4$, Recursive Doubling)

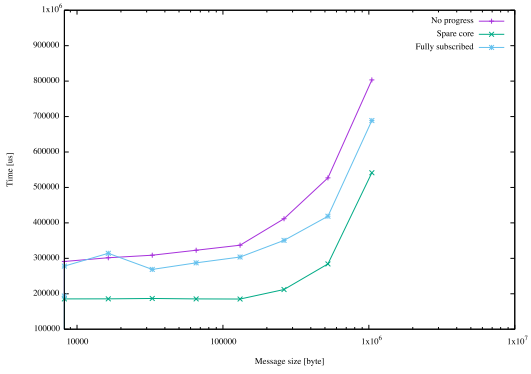

図 14 8,192byte 以上での MPI_Iallgather の全体の所要時間 ($n_node = 128$, $p_node = 4$, Recursive Doubling)

5. 関連研究・関連技術

非ブロッキング集団通信の性能向上に向け，通信隠蔽の効果を更に引き出すことと通信時間の短縮という二つの要求を満たすための様々な手法が提案されている．

progress thread について，各プロセスに付き 1 つ progress thread が起動されることから，計算への影響が無視できなくなるという問題点を Hoefler らは指摘した [2]．Hoefler らは progress thread 利用時におけるプロセス数と通信性能に関する調査を行ったが，大規模並列計算機上で広く用いられているスレッドベースの並列化技術への影響を調べられていない．このため，大規模並列計算機上で用いられるアプリケーションに progress thread を利用すべきか否か明らかにはなっていない．また，MPI とスレッドベースによる並列化技術を併用した hybrid 並列プログラミングにおいてはノード内プロセス数，スレッド並列数に複数の組み合わせが考えられるため，これらの組み合わせの選択に progress thread がどのような影響を及ぼすか検証する必要がある．

非ブロッキング集団通信における通信時間の短縮のために，Colmenares らは非ブロッキング集団通信向けアルゴリズムを提案した [3]．非ブロッキング集団通信は通信の完了を待たずに別の処理を行うことができるという特徴から，集団通信関数の呼び出し時刻が大きくずれる場合に利用されることが考えられる．この場合，ブロッキング通信で用いられるほとんどのアルゴリズムでは構成する一対一通信の順序に依存関係があるため，一プロセスの通信の遅れが伝搬し全体の通信時間に大きな影響を及ぼす．そこで，この論文によって提案された非ブロッキング集団通信向けアルゴリズムでは，非ブロッキング集団通信を最短経路問題としてモデル化を行い，これに沿って通信を行う．提案された手法は，通信時間の短縮という非ブロッキング集団通信への要求を満たすためのものであり，通信隠蔽の効果を引き出すことを目的とした progress thread を用いる手法と大きく異なる．今後これら二つの手法を組み合わせた場合，非ブロッキング集団通信に必要とされる二つの要求が達成されるのか，そうではないのか検証する必要がある．

Kandalla らは，プロセス数に限らずノード内に一つの”communication progresss servlet thread” を立ち上げ，非ブロッキング集団通信を行うようなフレームワークを提案した [4]．このフレームワークでは，ノード内に複数プロセスを実行する場合でもノード内の通信スレッドは 1 つになるため，CPU コアを計算に割り当てる事ができる数が多く，オーバーヘッドの小さなものとなる．しかし，計算に対して通信が大きなコストとなる場合，この手法を利用したフレームワークではノード内の全てのプロセスにおける通信が一つのスレッドに集中するため，通信の完了を待つ前に計算が終了する可能性が考えられる．このため，通信

の隠蔽のためにはより多くの計算を実行する必要がある．

また，集団通信におけるアルゴリズム選択については，STAR-MPI(Self-Tuned Adaptive Routines for MPI Collective Operations)，ADCL(Abstract Data and Communication Library) が提案されている [5] [6] [7]．これらは，メッセージサイズやプロセス数などパラメータからアルゴリズムを静的に選択するのではなく，集団通信におけるアルゴリズムを複数試し，その結果最速のものを以後の集団通信に用いるというものである．これらの提案については，アルゴリズムを決定するまでに遅いアルゴリズムを選択してしまうという問題点がある．また，progress thread を利用等，実装手段の選択まではできず，あくまでアルゴリズム選択のみに特化したものである．

非ブロッキング集団通信では，集団通信中で行われる一対一通信も非同期通信で行う必要がある．これは，送受信の順序が決められているアルゴリズムの場合，集団通信の処理を行うたびにメッセージが到着したかどうかを頻繁に確認するため，busy polling と呼ばれる状態に陥る．Miwa らはこの確認処理のため他の処理への影響が出ると考えた．受信側による busy polling の代わりに，メッセージの送信完了と同時に受信側へ RDMA による通知を行う．この通知によって busy polling を防ぎ，効果があることを確認している [8]．この手法は非ブロッキング集団通信を隠蔽しつつ進めるための処理を小さくすることで高いオーバラップを引き出そうというものである．提案された手法では，pingpong での遅延時間のみ評価されているため，非ブロッキング集団通信において通信の隠蔽がどれほど達成されるのか調査する必要がある．

また，ソフトウェアのみならず，progress thread の欠点をハードウェアから克服するような技術の研究開発も進められている．Fujitsu FX100 に搭載されている SPARC64 Xlfx チップでは，計算コアとは別にアシスタントコアと呼ばれる，OS に関わるプロセスや通信を専門に処理するコアが搭載されており，progress thread 機能を有効にした場合，アシスタントコアによって処理が進められるような機能が実装されている [9]．他にも，Offloading 機能と呼ばれる機能をもつ NIC(Network Interface Card) の開発も進められている．これは，progress thread では集団通信にかかる処理を CPU によってではなく，NIC によって実現するものである [10][11][12][13]．Hoefler らによって指摘された progress thread を用いた手法が持つ問題点 [2] をハードウェアの面から解決するものではあるが，Kandalla らによって提案された手法 [4] 同様，ノード内における通信処理がアシスタントコア，NIC に集中する可能性があり，通信隠蔽にはより多くの計算が必要となる可能性が考えられる．

6. まとめと今後の課題

本稿では，MPI ライブラリにおける progress thread 機能に着目し，通信隠蔽の効果の高さ，progress thread を利用することによる問題点を示した．そして hybrid 並列プログラムを用いて progress thread を利用した場合，通信，計算性能へどういった影響を及ぼすのか調査した．

実験結果から，8,192byte 以上のメッセージサイズでの通信において，通信隠蔽の効果を示した．また，hybrid 並列プログラミングにおいて，progress thread を有効にする場合，spare core を使用し，ノード内プロセス数を小さくして実行した場合が最も早く計算と通信を実行することができた．しかしこれらの結果は他の計算機では異なる結果となる可能性があるため，CPU コア数やコンパイラが異なる計算機を用いた場合，結果がどのように変化するかを調べる必要がある．

また，オーバーラップさせる計算量，非ブロッキング集団通信関数の呼び出しタイミングのずれ等，今回は考慮しなかったアプリケーションの情報によっても性能が変化するため，今後非ブロッキング集団通信関数がより一般的なアプリケーションにも利用されるようになった場合のチューニングの手間が増大することが考えられる．そのため，従来のような静的なアルゴリズム選択ではなく，チューニングの支援を行うツール，アルゴリズムのみならず実装手段をも動的に選択するような技術の研究開発が課題となる．

参考文献

[1] Lusk, E., Huss, S., Saphir, B. and Snir, M.: MPI: A Message-Passing Interface Standard, *International Journal of Supercomputer Applications*, Vol. 8, No. 3/4, p. 623 (online), available from ⟨http://www.ce.unipr.it/didattica/siselab/api_MPI/mpi1.1/mpi-report.htm\nhttp://www.mpi-forum.org⟩ (2009).

[2] Hoefler, T. and Lumsdaine, A.: 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).

[3] Colmenares, E. A. and Zhuang, Y.: Maximizing Hardware Performance via Non-blocking Collective Communication for All Pairs Shortest Paths Computation on Heterogeneous Multi-core Processors, *Parallel and Distributed Processing Techniques and Applications (PDPTA)*, pp. 1–5 (2014).

[4] Kandalla, K., Subramoni, H., Tomko, K., Pekurovsky, D. and Panda, D. K.: A Novel Functional Partitioning Approach to Design High-Performance MPI-3 Non-Blocking alltoallv Collective on Multi-Core Systems, *Proceedings of the International Conference on Parallel Processing*, pp. 611–620 (2013).

[5] Faraj, A., Yuan, X. and Lowenthal, D.: STAR-MPI: self tuned adaptive routines for MPI collective operations, *Proceedings of the 20th annual international conference on Supercomputing*, pp. 199—-208 (online), available from ⟨http://portal.acm.org/citation.cfm?id=1183431⟩ (2006).

[6] Faraj, A., Patarasuk, P. and Yuan, X.: A study of process arrival patterns for MPI collective operations, *International Journal of Parallel Programming*, Vol. 36, No. 6, pp. 543–570 (2008).
[7] Barigou, Y., Venkatesan, V. and Gabriel, E.: Auto-tuning Non-blocking Collective Communication Operations, pp. 1204–1213 (2015).
[8] Miwa, M., Nakashima, K. and Naruse, A.: Interference-aware Incoming Message Detection for MPI threaded progression, *Proceedings - 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, CCGrid 2013*, pp. 184–185 (2013).
[9] 新庄直樹：富士通の HPC に向けた取り組み，（オンライン），available from ⟨https://www.ssken.gr.jp/MAINSITE/event/2015/20150828-hpcf/lecture-04/SSKEN_hpcf2015_shinjo_presentation.pdf⟩ (2015).
[10] Kandalla, K., Subramoni, H., Tomko, K., Pekurovsky, D., Sur, S. and Panda, D. K.: High-Performance and Scalable Non-Blocking All-to-All with Collective Offload On InfiniBand Clusters: A study with parallel 3D FFT, *Computer Science - Research and Development*, Vol. 26, No. 3-4, pp. 237–246 (2011).
[11] Kandalla, K., Subramoni, H., Vienne, J., Raikar, S. P., Tomko, K., Sur, S. and Panda, D. K.: Designing Non-blocking Broadcast with Collective Offload on InfiniBand Clusters : A Case Study with HPL *, No. 1 (2011).
[12] Kandalla, K., Yang, U., Keasler, J., Kolev, T., Moody, A., Subramoni, H., Tomko, K., Vienne, J. and Panda, D. K.: Designing Non-blocking Allreduce with Collective Offload on InfiniBand Clusters : A Case Study with Conjugate Gradient Solvers (2011).
[13] Kandalla, K., Yang, U., Keasler, J., Kolev, T., Moody, A., Subramoni, H., Tomko, K. A., Vienne, J., De Supinski, B. R. and Panda, D. K.: Designing Non-blocking Allreduce with Collective Offload on InfiniBand Clusters: A Case Study with Conjugate Gradient Solvers, *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pp. 1156–1167 (online), (2012).