Bulletin of Networking, Computing, Systems, and Software – www.bncss.org, ISSN 2186–5140
Volume 3, Number 1, pages 25–28, January 2014

Performance Study of Non-blocking Collective Communication Implementations Toward Adaptive Selection

Tsuyoshi Okuma
Graduate School of Information Science and Electrical Engineering
Kyushu University
6-10-1 Higashi-ku, Fukuoka, 812-8581 Japan
Email: 2ie11005e@s.kyushu-u.ac.jp

Takeshi Nanri
R.I.I.T of Kyushu University
JST, CREST
Japan
Email: nanri@cc.kyushu-u.ac.jp

***Abstract*—Non-blocking collective communication is a key technique for enabling overlap of collective communication and computation to achieve higher scalability on large scale parallel computers. There can be many types of implementation methods for non-blocking collective communications. To sufficiently obtain effect of overlap, appropriate method should be chosen. Therefore, it is important to study characteristics of each method. In this paper, we explain a characteristic of each method of implementation and and evaluate the characteristic by performing the experiment using a simple test program. In addition to that, this paper proposes an implementation with lower overhead, and evaluates the performance.**

***Keywords*—non-blocking collective communication, overlapping communication and computation, large scale parallel computing**

I. はじめに

並列計算の大規模化に伴い，通信時間の増大が並列計算の性能向上を阻害する要因となる．そこで，通信を計算と並行して実行することによる通信時間の隠蔽技術が注目されている．これには，通信の完了を待たずに次の処理に移行できる非ブロッキング通信が用いられる．特に集団通信は，多くのアプリケーションで使われているうえに，プロセス数に応じて通信時間が増加するため，非ブロッキング集団通信による通信時間隠蔽の効果は大きいと期待できる．

通常，集団通信関数は一対一通信を組み合わせたアルゴリズムで実装される．ブロッキング集団通信では，関数内でアルゴリズムを進行させ，完了を待ってから次の処理に移る．一方，非ブロッキング集団通信では，関数内ではアルゴリズムの完了を待たず，呼び出し元に戻って次の処理を始める．そのため，アルゴリズムの進み具合を確認し，進行させるための処理が別途必要となる．そのアルゴリズムを進める処理の実装手段としては，例えば，プログラム中で進行のための命令を繰り返し呼ばせる方法や，別のスレッドでアルゴリズムを進行させる方法など，複数考えられる．それら実装手法によって通信時間隠蔽の効果が大きく変わると予想されるため，実装ごとの特性の解析が必要である．しかしそのような解析は，未だ十分に行われていない．

そこで筆者らは，非ブロッキング集団通信の実装方法の特性を調査し，状況に応じて適切に実装方法を選択する技術の開発を目指している．適応的な非ブロッキング集団通信の実装を自動選択するための指針を得るためには，それぞれの実装方法の特性を調べる必要がある．従って，本論文では，いくつかの実装方法についてその特性を解析する．今回は，既存手法として非ブロッキング集団通信のライブラリである LibNBC [1], [2] を用いて実験を行った．それに加えて，この LibNBC よりオーバヘッドの低い実装方法である TCC を提案し，その性能を評価する．

II. 既存の実装

本稿では，非ブロッキング集団通信の実装例として LibNBC を評価対象とする．LibNBC は，集団通信アルゴリズムを MPI ライブラリの非ブロッキング一対一通信や計算等の命令に分解して並べ，それらの命令を個別に実行することでアルゴリズムを進行させる手法を用いている．この手法により複数の非ブロッキング集団通信を並行して進めることが可能となる．一方、この手法では，アルゴリズムを構成する命令の単位が小さくなるため，その構成にかかるオーバヘッドが必要となる．

LibNBC の実装は，アルゴリズムを進行させる方法により，さらにプログレス関数による実装とプログレススレッドによる実装に分類される．このうちプログレス関数による実装では，アルゴリズムを進行させるための関数であるプログレス関数を，プログラム中で繰り返し呼び出すことによりアルゴリズムを進行させる．プログレス関数はユーザが明示的に呼び出す必要があり，呼び出す頻度もユーザ自身が決める．これは，ユーザがアルゴリズムの進行を自ら管理することが可能になる，という利点であると同時に，最適な頻度でプログレス関数を呼び出すように調整する負担をユーザに強いる，という欠点でもある．

一方，プログレススレッドによる実装では，メインスレッドとは別に生成されたスレッドへアルゴリズムの進行を任せる．この生成されたスレッドをプログレススレッドと呼ぶ．この実装では，プログレススレッドが自動的にアルゴリズムを進行するので，ユーザがアルゴリズムの進行を負担する必要が無い．さらに，メインスレッドでプログレス関数を呼びだすことによるオーバヘッドが必要でなくなる．ただし，プログレススレッドに専用のコアを 1 つ割り当てる場合，そのコアは計算を全く行うことが出来ない．一方，専用のコアを割り当てない場合，計算を行うスレッドとプログレススレッドでコアを取り合うこととなるため，コアに対するスレッドのスケジューリングが性能に影響する．

III. 低オーバヘッド非ブロッキング集団通信 TCC

LibNBC の実装では集団通信アルゴリズムの構成におけるオーバヘッドが必要となる．そこで，集団通信を一対一通信に分解せず集団通信ごとに処理することにより低オーバヘッドで非ブロッキング集団通信を実装する手法 TCC(Threaded Collective Communication) を提案する．TCC では，まず集団通信を進行させるためのプログレススレッドを生成する．このプログレススレッドは，メインスレッドにおける非ブロッキング集団通信命令の発行順に，対応する MPI ライブラリのブロッキング集団通信命令を実行する．

TCC では集団通信アルゴリズムの構成において，集団通信を 1 単位として扱う．そのため，LibNBC のように集団通信を分解した非ブロッキング一対一通信や演算を 1 単位とする場合に比べ，容易にアルゴリズムを構成することができる．したがって，TCC のアルゴリズム構成方法はオーバヘッドが低い．ただし，必ずプログレススレッドを生成しなければならず，LibNBC のプログレススレッドによる実装と同じく，プログレススレッドに専用のコアを 1 つ割り当てか否かを選択しなければならない．

TCC では，ハンドルおよびハンドルキューというデータ構造を用いて呼び出された非ブロッキング集団通信を管理する．ハンドルは，集団通信の種類，その集団通信に渡された引数，集団通信が終了したかどうかを判定するためのフラグからなる構造体であり，呼び出した非ブロッキング集団通信一つにつき一つのハンドルが対応している．ハンドルキューは，ハンドルを格納しておくためのリングキューである．このハンドルキューに格納されるハンドルを介してメインスレッドとプログレススレッドが連携して集団通信を進めて行く．これらのスレッドは POSIX スレッドで実装し，ハンドルキューは排他制御変数と条件変数で排他的に管理する．

TCC による非ブロッキング集団通信は，プログラミングインタフェースとして，初期化関数，各非ブロッキング集団通信関数，Wait 関数と終了処理関数を提供する．初期化関数では，ハンドルキューの排他制御変数，条件変数の初期化を行う．非ブロッキング集団通信関数では，ハンドルに集団通信の種類と引数を設定し，さらに終了判定フラグの初期化を行い，最後にハンドルをハンドルキューにエンキューする．エンキューの際にハンドルキューが一杯になっている場合は，空きができるまで条件変数によって待つ．Wait 関数では，指定されたハンドルの終了判定フラグが真になるまで待つ．終了処理関数では，まず，発行された全ての非ブロッキング集団通信の完了を待つためにハンドルキューの残り要素数が 0 になるまで待ち，その後，プログレススレッドへ終了を知らせるシグナルを送り，プログレススレッドの終了を待つ．

プログレススレッドは，ハンドルキューからハンドルを 1 つデキューし，そのハンドルに設定された集団通信の種類に対応した MPI のブロッキング集団通信を行う．デキューの際，ハンドルキューが空の場合は，新しいハンドルがエンキューされるまで条件変数によって待つ．また，集団通信が終了した後，ハンドルの終了判定フラグを真にする．

IV. 実験

A. 実験の概要

各実装による計算と通信のオーバラップの効果を評価するため，LibNBC および TCC による非ブロッキング集団通信について実験を行った．なお，LibNBC は集団通信アルゴリズムを分解して命令毎に実行するため，独自のアルゴリズムを用いて実装されている．一方 TCC では，MPI の集団通信をプログレススレッドで呼び出すため，MPI ライブラリ内部のアルゴリズムが用いられる．そのため，LibNBC と TCC の実験結果を単純に比較することはできない．MPI ライブラリ内部で用いられているアルゴリズムを分解して LibNBC で用いることにより，LibNBC と TCC の比較が可能となるため，今後の課題とする．

実験に用いたプログラムは，非ブロッキング Alltoall 通信を発行した後に行列ベクトル積の計算を行うものである．このプログラムを用いて所要時間の計測を行った．行列ベクトル積の計算では，2 重ループの外側のループに対して MPI によるブロック並列化を行うと同時に，OpenMP によるスレッド並列化を行うことでハイブリッド並列を施した．また，比較のため，LibNBC，TCC それぞれについて，行列ベクトル積の直前で非ブロッキング集団通信を完了させることにより通信を隠蔽させないプログラム，および MPI ライブラリのブロッキング集団通信と行列ベクトル積を順に実行するプログラムについても計測も行った．

B. 実験環境

実験に使用した計算機は，九州大学が所有する Fujitsu PRIMERGY CX400 である．この計算機の総ノード数は 1476 ノードであり，このうち 32 ノードを使用して実験を行った．各ノードには 16 基のコアと 128GB の主記憶容量が搭載されている．CPU は Intel(R) Xeon(R) CPU E5-2680 2.7GHz である．ノード間インターコネクトとしては，InfiniBand FDR が用いられている．OS は Red Hat Enterprise Linux Server release 6.1 であり，カーネルは 2.6.32-131.0.15.el6.x86$_{64}$ である．コンパイラは Intel Composer XE 2011，MPI ライブラリは Intel(R) MPI Library 4.0 Update 3 for Linux を用いた．また，本実験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進行を行うため，MPI のスレッドモードとして THREAD_MULTIPLE を用いた．

C. 実験結果

実験では，以下の 5 ケースについて計測を行った．

NBCF-noovlp
: LibNBC のプログレス関数による実装において通信隠蔽を行わない場合

NBCF-ovlp-test
: LibNBC のプログレス関数による実装において通信隠蔽を行い，プログレス関数を行列ベクトル積の外側ループ 1 回につき 1 度呼ぶ場合

NBCF-ovlp-notest
: LibNBC のプログレス関数による実装において通信隠蔽を行い，プログレス関数を一度も呼ばない場合

NBCT-ovlp
: LibNBC のプログレススレッドによる実装において通信隠蔽を行う場合

MPI
: MPI ライブラリの Alltoall 関数を用いて通信隠蔽を行わない場合

TCC-ovlp
: TCC による実装において通信隠蔽を行う場合

計算ノード数が 2 ノードおよび 32 ノードの場合の所要時間を図 1 および図 2 に示す．

各ノードで MPI のプロセス数は 1 とし，スレッド並列による行列ベクトル積の計算において生成するプロセスごとのスレッド数は，プログレススレッドを用いない実装の場合 16 としたのに対し，プログレススレッドを用いる実装の場合は 15 として，プログレススレッドに 1 コアを専有させた．また，メッセージサイズと行列のサイズは，ノード数 2 の場合がメッセージサイズ 5MB，行列サイズ 4,000×4,000，ノード数 32 の場合がメッセージサイズ　500KB，行列サイズ 30,000×30,000 とした．

はじめに，LibNBC による実験結果について考察する．通信隠蔽の効果については，どのケースでも通信隠蔽する場合の方が通信隠蔽しない場合より速いか，もしくは同等の所要時間であったことから，有効性を確認できた．また，プログレス関数による実装とプログレススレッドによる実装を比較すると，プログレス関数による実装の方が速い，という結果となった．この理由として，プログレススレッドによる実装におけるスレッド生成や排他制御等のオーバヘッドの影響が考えられる．一方，プログレス関数の呼び出しの有無による違いについては，32 ノードではほぼ同じ所要時間だったのに対し，2 ノードにでは大きな差が見られた．今回の実験においては，LibNBC の Alltoall 通信は，他プロセスとの非ブロッキング一対一通信を一度に全て発行し，後はその完了を待つというアルゴリズムを用いているため，アルゴリズムを進行させる操作が必要無く，プログレス関数の呼び出しの有無で結果がほぼ変わらないと予測していたため，この結果については今後原因究明のための解析を行う．

次に，TCC による実験結果について考察する．TCC においても，ノード数 2，32 の双方で通信隠蔽の効果を確認することができた．tj なお，非ブロッキング集団通信の同時呼び出し数が少ない場合，集団通信アルゴリズムを細かい命令に分けて進行させる LibNBC に比べて TCC の方式が有効だと考えられる．実プログラムでも今回の実験と同様の状況は多い．しかし，今回の実験では，前述の通り，LibNBC との単純な比較はできないので，この有効性の実測による確認は今後の課題である．

V.　非ブロッキング集団通信の適応的な選択に向けて

この章では，今回の実験結果に基づいた非ブロッキング集団通信の適応的な自動選択へ向けての指針を述べる．

まず，プログラマがプログレス関数を実行する適切な頻度を探ることができる場合について考える．今回の実験では，プログレス関数を呼び出す頻度は基本的に行列ベクトル積の内側ループ 1 回につき 1 度，もしくは外側ループ 1 回につき 1 度，の 2 通りが考えられる．このように，プログレス関数を実行する頻度の選択肢が少ない場合，最適な頻度を探ることはプログラマにとってあまり負担にはならない．実際に，今回は内側と外側の 2 通りを試し結果の良いほうを選んだが，わずか 2 通りであったため大きな負担とはならなかった．したがって，この場合には，実験により得られた通信隠蔽の効果とプログラマの負担を比較すると，プログレススレッドにアルゴリズムの進行を任せる必要はなく，プログラマがプログレス関数を明示的に呼び出したほうが良いと言える．

次に，プログラマがプログレス関数を実行する適切な頻度を探ることが困難な場合について考える．これは，非ブロッキング集団通信とオーバラップさせる計算が複雑で，プログレス関数を実行する適切な頻度の予想や実測が困難な場合である．そのような例としては，3 重以上の多重ループ

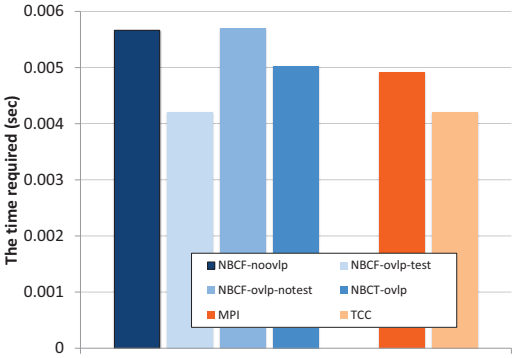

Fig. 1.　各ケースの所用時間 (ノード数 2，メッセージサイズ 5MB，行列サイズ 4,000×4,000)

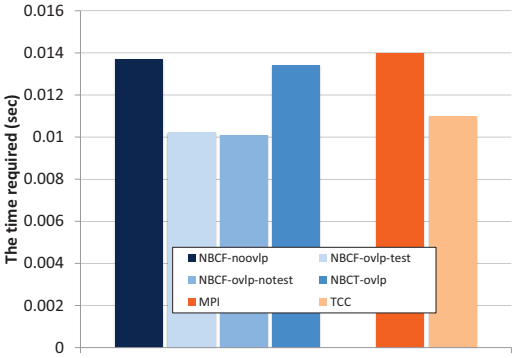

Fig. 2.　各ケースの所用時間 (ノード数 32，メッセージサイズ 500KB，行列サイズ 30,000×30,000)

や，複雑な条件分岐のある計算とオーバラップをする状況が考えられる．

この場合は，プログレス関数による実装では，プログレス関数の適切な実行頻度を探るためのプログラマの負担が大きい．そのため，プログレススレッドを用いた非ブロッキング集団通信実装の利用が好ましい．しかし今回の実験結果によると，LibNBC を用いた場合のプログレススレッドによる実装での通信隠蔽の効果は，ノード数 2 においては，通信隠蔽によるスピードアップは約 1.127 であり，プログレス関数による実装の約 1.346 に比べて小さい．また，ノード数 32 においては，プログレススレッドによる実装のスピードアップは約 1.020 であり，プログレス関数の約 1.338 との差が大きくなっている．これは，プログレススレッドのために計算スレッドを一つ減らしていること，および，プログレススレッドと計算スレッドの間の同期コストの影響が考えられる．そのため，LibNBC による実装では，可能な限りプログラマによるプログレス関数の挿入を選択する方が良いと思われる．

なお，今回の実験ではオーバラップさせる集団通信の数が 1 つだけであったのに対し，LibNBC の実装の利点として，複数の非ブロッキング集団通信を並行して進められることがある．そのため，そのような状況では LibNBC による通信隠蔽の効果がより顕著に表れる可能性がある．

一方 TCC では，ノード数 2 におけるスピードアップが約 1.166，ノード数 32 におけるスピードアップが約 1.271 で

あった．TCC は，同時に一つの非ブロッキング集団通信しか進行させることが出来ないが，構造が単純であるためオーバーヘッドが少なく，今回のようにオーバラップさせる集団通信が一つの場合，通信隠蔽の効果が得られやすいと考えられる．

以上のことから，今回の実験により，非ブロッキング集団通信の実装手段を選択するに当たり，プログレス関数の適切な実行頻度を見つけるのが容易であるか否か，および同時に進行させる集団通信の数が，重要な指標であることが分かった．また，適切な実装手段選択の実現に向け，複数の非ブロッキング集団通信を進行させる場合の LibNBC での通信隠蔽の効果の検証，同じアルゴリズムによる LibNBC と TCC の性能比較が必要であることが分かった．

なお，プログレススレッドを用いる場合については，ノード内スレッド数による性能への影響も検証する必要がある．今回の実験では，プログレススレッドを用いる際、計算スレッドを一つ減らして 1 コアをプログレススレッドに占有させた．一方，実行時の選択肢としては，計算スレッド数を減らさず，スレッドがコアを取り合うよう実行させることも可能である．これは，1 コア分の計算能力の減少と，スレッドを取り合ってコンテキストスイッチを繰り返すことによるコスト増のトレードオフとなる．

VI. おわりに

非ブロッキング集団通信の実装手法として，既存の手法である LibNBC と，著者らが提案した手法 TCC を用い，それぞれについて通信隠蔽による効果を検証した．TCC では，集団通信アルゴリズムの単位として集団通信を用いることで，その構成のオーバヘッドを低く抑えている．実験では，LibNBC，TCC 双方で通信隠蔽の効果が得られた．また，非ブロッキング集団通信の実装手段自動選択に向け，いくつかの指針を得ることができた．

今後は，同じ集団通信アルゴリズムによる LibNBC と TCC の比較や，同時に信仰させる集団通信の数，さらにプログレススレッド使用時の計算スレッド数について検証を行い，それらをもとに状況に応じた適切な実装方法自動選択技術の確立を目指す．

REFERENCES

[1] T. Hoefler, A. Lumsdaine, and W. Rehm, Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI, Proceedings of the 2007 International Conference on High Performance Computing, Networking, Storage and Analysis, IEEE Computer Society/ACM, (2007).

[2] T. Hoefler, A. Lumsdaine, Message Progression in Parallel Computing - To Thread or not to Thread?, Proceedings of the 2008 IEEE International Conference on Cluster Computing, pp.213-222, (2008).