Vol.2010-HPC-126 No.13
2010/8/3

カーネルレベル MPI 非同期集団通信機構の設計と実装

野 村 哲 弘[†1]　石 川 裕[†1]

我々は既に MPI 3.0 において規格化される予定の非ブロッキング集団通信を高速に実行する手法 KACC を提案した．KACC は集団通信の進捗処理を OS カーネル上に実装することによって，従来手法では問題となっていたオーバヘッドの問題と実行タイミングの問題を解決している．本報告では，KACC が OS カーネル上に進捗処理を実装することに起因する負荷分散に関する問題点と，その解決法を示す．KACC のテスト実装において CPU 時間の消費および通信アルゴリズムの実行時間の 2 点について従来手法と比較し，解決法の妥当性を評価する．

Design and Implemenation of Kernel-level Asynchronous MPI Collective Communications

Akihiro Nomura[†1] and Yutaka Ishikawa[†1]

We have proposed a new method, called KACC, to execute non-blocking collective communications, which will be introduced in MPI 3.0, efficiently. KACC resolves overhead and timing problems in former methods by implementing progression of collective communications in the OS kernel. In this paper, we describe the load-balancing problem, which is caused by implementing progression in OS kernel, and its solution. We evaluate CPU time consumption and execution time of collective communications among several implementations.

1. はじめに

MPI 通信ライブラリ 3.0[1] では，非ブロッキング集団通信 API が導入される予定である．非ブロッキング集団通信とは，従来の MPI 集団通信と同様の並列計算でよく使われる定型的通信の組み合わせを，MPI 非ブロッキング 1 対 1 通信と同様にバックグラウンドで実行することで，通信待ちの時間を計算に利用できるようにするものである．MPI 非ブロッキング集団通信のリファレンス実装として，LibNBC[2] ライブラリが公開されている．LibNBC ライブラリは MPI 2 上で pthread スレッドライブラリを用いて動作するライブラリであり，集団通信の進捗管理を行うスレッドをアプリケーションのメインスレッドとは別に起動することで，集団通信と計算の同時実行を実現している．しかしながら，このような実装では，通信用スレッドと計算用スレッドの総数がノード上の CPU コア数を越えた際に頻繁にスレッド間のコンテキストスイッチが起こる原因となり，パフォーマンスの低下を招く．

我々はこれらの問題を解決するために，既に KACC(Kernel-level Asynchronous Collective Communication) 機構を提案した[3]．KACC は，上記の問題点の原因となる通信の進捗処理を OS カーネル内の割り込みコンテキストで実行することによって，コンテキストスイッチおよびそのタイミングに起因する問題点を克服するものである．前記の論文では，紙面の都合上提案機構 KACC の説明を中心にしており，実装の詳細点とそこに起因するトレードオフについて十分に議論することが出来なかった．

本報告では，既存の非ブロッキング集団通信の実装における問題点と，それを解決した実装方式である KACC 機構の概要について述べた後に，KACC での通信進捗処理の負荷分散に関する問題点について論じ，複数の実装における性能の比較を行う．

2. 既存手法の問題点

集団通信を非ブロッキングで実装する際には，1 対 1 通信ではさほど問題とならない通信の進捗処理という問題に直面する．一般的に集団通信は 1 対 1 通信の組み合わせで実装されるが，これらの通信の間にはほとんどの場合依存関係がある．たとえば，受け取ったデータを他のノードに転送する場合は，該当する受信が完了してからでないと，送信を開始することが出来ない．このような依存関係を解決し，実行可能となった通信を実行する処理が通信の進捗処理である．通信の進捗管理の実装方法としては，従来スレッドを用いる手法と明示的に進捗処理を呼び出す手法の 2 つが考慮されてきた．以下に，各方法の利点と欠点を述べる．

2.1 スレッド方式による進捗処理手法

進捗処理を計算と独立して行う最も単純な実装手法は，進捗処理用のスレッドを作る方法である．この方式の実装例には LibNBC[2] の実装が挙げられる．この方式の利点は，通信を計算と独立して設計し，非同期に実行できる点である．理論的には計算と通信は互いに干

†1 東京大学
The University of Tokyo

© 2010 Information Processing Society of Japan

渉することなく最適なタイミングで実行される．

しかしながら，現実には 1 ノードあたりの CPU コアの数は限られており，当然同時実行できるスレッドの数も限られている．並列アプリケーションのユーザは一般的にノード内の CPU コアと同数の MPI プロセスもしくは OpenMP のスレッドを作成し，すべての CPU コア上で計算処理を動かそうとする．このような状況下で通信用のスレッドを作ろうとすると，スレッドの数が CPU コア数を上回り，頻繁なコンテキストスイッチが起こる原因となる．また，OS はこれらのスレッドを明示的に区別して扱うわけではないので，通信スレッドが進捗処理によって次の通信を開始できるタイミングとは無関係に通信スレッドをスケジュールする．このことによって，通信が開始できるようになったタイミングで通信スレッドがスケジュールされず，他のスレッドが CPU コアを開放するまで待たされるという問題も起こる．このように，通信スレッドを作っても通信の進捗処理が計算スレッドの干渉を受けて遅くなったり，計算自体の速度も低下する可能性がある．

2.2 明示的に進捗処理を呼ぶ手法

前節で示したスレッド方式の欠点を防ぐには，進捗処理も計算スレッドの中から呼び出す形にして実行コンテキストを増やさないようにしなければならない．そのためには，集団通信と同時に実行されている計算の途中で，定期的に進捗処理のためのルーチンを呼び出し，そのなかで `MPI_Testall` などの関数を呼び出すことで明示的に通信を進捗させる必要がある．このような実装は非ブロッキング集団通信のない MPI 規格上で同等の処理を行うために並列アプリケーションに広く実装されている．例えば，HPL ベンチマーク[4] では，この方式で非ブロッキング版のブロードキャストアルゴリズムが複数実装されており，アルゴリズムの選択がベンチマークのスコアを決める要因のひとつとなっている．

この方式では，進捗処理ルーチンの呼び出し頻度が問題となる．集団通信を構成する 1 対 1 通信が終わるごとに進捗処理ルーチンが呼ばれるまでの待ち時間の間，通信は進行しない．進捗処理ルーチンを呼ぶ頻度が低すぎると，オーバラップ対象の計算を全て終わらせて，通信の終了待ちになるまで通信が進行せず，実質的にブロッキング通信となってしまうため，計算と通信のオーバラップによる性能向上が得られなくなる．逆に進捗処理ルーチンを呼ぶ頻度が高すぎると，構成する 1 対 1 通信が終わる前に `MPI_Testall` などの MPI 関数が何度も呼び出されることとなり，これらの関数のオーバヘッドが全体の性能低下に直結する．

3. KACC の概要

3.1 CAD グラフ

我々が提案した手法である KACC[3] は，スレッド方式を基にしている．スレッド方式におけるコンテキストスイッチの問題を解決するために，通信および進捗処理の部分を OS カーネルの割り込みハンドラから起動される Linux タスクレット[5] として実装する．このことにより，通信のためのスレッドを作らないで済むようになる．割り込みハンドラとタスクレットは，プロセスコンテキストを持たないため，通信の進捗処理を行う際にコンテキストスイッチが発生しない．その一方で，通信の進捗処理を OS カーネル空間で行うため，集団通信アルゴリズムをどのようにして通信機構に渡すかが問題となる．単純にユーザがコンパイルしたプログラムを OS カーネル上で実行する方法は，セキュリティホールとなるため使えない．このため，KACC では，MPI アプリケーションと OS カーネル上の通信機構の間で通信アルゴリズムをやり取りするためのデータ構造である CAD(Collective Algorithm Design) を導入する．

集団通信は送信を示すノード `SEND`，受信を示すノード `RECV`，および `MPI_Reduce` で行われるような 2 項演算を示すノード `CALC` とそれらの間の依存関係を示す有向枝からなる非循環有向グラフ (DAG) として表現することができる[6),7]．CAD は，このようなグラフ構造を OS カーネルとやりとりしやすくするためにポインタを使わずに固定長要素の配列として表現したものである．CAD 上には始点と終点を表す特殊なノード `START`，`END` が存在し，グラフ上のすべてのノードは `START` から到達可能で，`END` に到達可能である．各ノードは，先行する全ノードの実行が終了した時点で実行が可能になるという制約の下で `START` ノードから実行が伝播していき，アルゴリズム上の全処理を完了し，`END` が実行可能となった時点でアルゴリズムの実行が終了する．これらの処理はアプリケーションの実行スレッドに対して非同期に実行される．

図 1 にリング上のブロードキャストアルゴリズムに対応する CAD グラフの例を示す．リング上のブロードキャストでは，メッセージを適切なサイズのチャンクに分割し，各チャンクを手前のノードから受け取り次のノードへ転送して行く．CAD は通信に参加する各ノード毎に独立しており，この例では中間ノードであるランク 1 のプロセスに対応する CAD グラフを示しているが，ブロードキャストの始点となるプロセスでは `SEND` に対応するノードが省略されたグラフとなり，終端のプロセスでは逆に `RECV` が省略されたグラフとなる．

集団通信の実装者は，以下に示す API を用いてアルゴリズムに対応する自ランクにおけ

© 2010 Information Processing Society of Japan

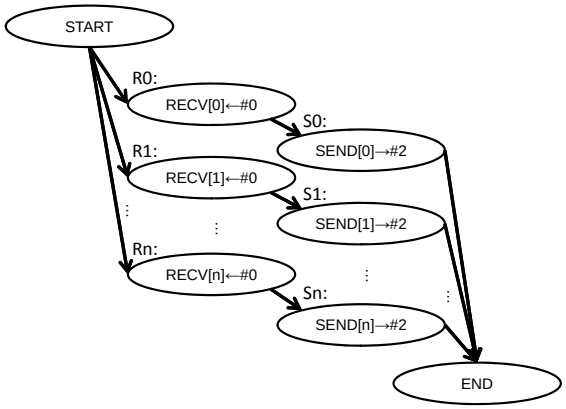


図 1 CAD グラフの例

```
 1 /* Initializing CAD Tree */
 2 cad = InitCAD();
 3 /* Making R0 and S0 Node */
 4 rn = MakeRecvNode(cad, addr[0], chunksize, 0);
 5 ConnectNode(cad, START, rn);
 6 sn = MakeSendNode(cad, addr[0], chunksize, 2);
 7 ConnectNode(cad, rn, sn);
 8 ConnectNode(cad, sn, END);
 9 /* Making R1 and S1 Node */
10 rn = MakeRecvNode(cad, addr[1], chunksize, 0);
11 ConnectNode(cad, START, rn);
12 sn = MakeSendNode(cad, addr[1], chunksize, 2);
13 ConnectNode(cad, rn, sn);
14 ConnectNode(cad, sn, END);
15 ....
16 /* Making Rn and Sn Node */
17 rn = MakeRecvNode(cad, addr[n], chunksize, 0);
18 ConnectNode(cad, START, rn);
19 sn = MakeSendNode(cad, addr[n], chunksize, 2);
20 ConnectNode(cad, rn, sn);
21 ConnectNode(cad, sn, END);
22 /* Issuing CAD Tree */
23 req = IssueCAD(cad);
```

図 2 図 1 に対応する CAD グラフを生成するコード

る CAD グラフを作成し，KACC のカーネルモジュールに渡し，終了を待つ．

- `InitCAD()`: 新しい CAD を作成する
- `MakeSendNode()`, `MakeRecvNode()`, `MakeCalcNode()`: `SEND`，`RECV`，`CALC` に対応する CAD ノードを作成する
- `ConnectNode(A, B)`: `A` の後に `B` が実行される依存関係の枝を張る
- `IssueCAD()`: 作成した CAD を KACC 通信機構に渡し，実行を開始する
- `QueryCAD()`: 対応する通信の実行が完了したかを問い合わせる

図 1 に対応する処理の流れを**図 2** に示す．このコードでは説明を簡便にするためにランクごとの条件分岐や繰り返し構造を展開している．ブロードキャストに使われるメモリ領域のアドレスはチャンクごとに配列 `addr` に格納されているものとする．集団通信のコードは，2 行目にあるように最初に `IssueCAD` を呼び，CAD グラフのためのメモリ領域を確保することから始まる．各チャンクに対応する `RECV`，`SEND` のノードは `MakeRecvNode` や `MakeSendNode` によって作成される．作成されたノードは `ConnectNode` によって前後のノードや `START`，`END` のノードに接続される．CAD グラフを構成する全ノードと接続する有向辺の作成が完了したら 23 行目にあるように `IssueCAD` によって通信機構へと転送し，非ブロッキング集団通信を開始させる．図 2 には示していないが，実行を開始した通信の終了は `QueryCAD` を呼ぶことによって確認できる．

実際に非ブロッキング集団通信を実装するときには，既存のアルゴリズムのうち，1 対 1 通信を行っている部分を `MakeRecvNode()` 等に置き換え，それらの間の依存関係を `ConnectNode` によって追加することでアルゴリズムに対応する CAD を作ることができる．

3.2 実装の概要

図 3 に示すように，KACC を CAD API，Progress Engine，P2P の 3 層に分けて実装した．CAD API はユーザレベルのライブラリとして，それ以外の 2 層は Linux 2.6 カーネル上にカーネルモジュールとして実装した．

CAD API は CAD グラフに対応するデータ構造を OS カーネルからのアクセスに適した固定長要素の配列の形で格納する．また，`SEND` や `RECV` に対応するノードの作成時には，MPI ライブラリに依存する型やコミュニケータの情報を解決し，`MPI_COMM_WORLD` 上のランクでのバイト列の送信に変換する．同様に `CALC` のノードについても MPI 依存の型名などの定数を KACC のものへと変換する．CAD を格納するメモリ領域はカーネルモジュールと共用することで，CAD グラフの転送にかかる時間を節約する．CAD のデータは OS カーネルからもアクセスされるため，プロセスの仮想アドレスに依存するポインタを使わな

© 2010 Information Processing Society of Japan

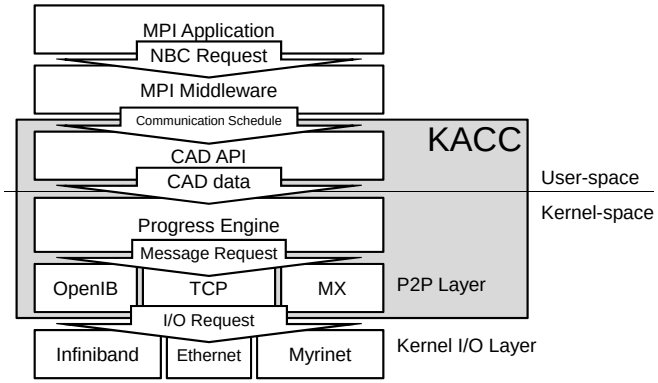


図 3 KACC 機構のデザイン

いデータ構造を用いる．

Progress Engine(PE) 層では，通信の進捗管理を行う．PE 層と P2P 層はシステムコールもしくはネットワーク割り込みによって起動されるタスクレットとして実行される．PE 層はシステムコールによって CAD API から集団通信の開始要求をうけ，通信要求を後述する P2P 層に出す．前述のタスクレット起動条件を契機として CAD グラフ上の各ノードの処理をタスクレット上で実行する．CAD グラフに対応する通信が終わると，`END` ノードの処理としてリクエストに対応する共有メモリ領域に終了を示す値を書き込むことで CAD API に報告する．CAD API とのデータ交換はタスクレットを起動する必要があるものはシステムコール経由で，そうでないものはカーネルモジュール内で確保した共有メモリ経由で行う．

P2P 層では，PE 層に対してメッセージの送受信の API を提供する．P2P 層と PE 層の間の API はネットワークインタフェースに非依存な形で MPI の 1 対 1 非ブロッキング通信と似たインタフェースを提供する．MPI の非ブロッキング通信インタフェースとの違いは，通信終了の検出方法にあり，MPI では `MPI_Test` などでポーリングするのに対して，KACC では割り込みハンドラからタスクレットを起動して PE 層を起動する形で行われる．このような終了検出方法によって，PE 層が常に適切なタイミングで実行されることを保証している．現在，P2P 層としてカーネルレベルでの非ブロッキング TCP 通信を実装している．

4. 実装における問題点

KACC の特徴は，通信の進捗処理を OS カーネル内で，コンテキストスイッチをすることなく実装することにある．Progress Engine の処理は，依存関係の前段にあたる通信の終了などを契機として起動されなければならない．そのため，P2P 層を含めて OS カーネルの割り込みハンドラからも起動できる Linux のタスクレットとして実装している．タスクレットはプロセスコンテキストを持たず，常に実行をスケジュールした CPU コアと同じ CPU コアで実行され，同一のタスクレットは同時に別の CPU コアで実行されないことが保障されているという特徴を持つ．これらの特徴から，タスクレットを用いることで排他制御や状態管理を簡略化できる利点がある．タスクレットは割り込み処理やシステムコール処理の終了時に，プロセスの実行に戻る前のタイミングで実行される．

1 台のノード上では複数の MPI プロセスが実行されており，ノード内通信を除いては各プロセスの通信は干渉しないため，プロセスごとにタスクレットを分割することで進捗処理は自明に並列化できる．`IssueCAD` によって発行されるシステムコールを契機とするタスクレットの実行は，呼び出し元のプロセスと同じ CPU 上で実行されるため，MPI プロセスの配置が適切であれば自然に負荷分散がなされているといえる．しかしながら，進捗処理のタスクレットの大部分はメッセージの受信によって発生するネットワーク割り込みから起動される．ネットワーク割り込みは全ての CPU コアに平等に発生するものではなく，割り込みを受ける CPU コアはしばしば偏っている．そのため，割り込みコンテキストから単純にタスクレットの実行をスケジュールすると，ネットワーク割り込みのコア間での偏りがそのまま PE 層を実行する CPU コアの偏りとなるため，通信間での CPU 負荷分散がなされず，ノード内の全通信処理がシリアライズされてしまうという問題がある．

この問題を解決するため，必要に応じて他の CPU コアに対してプロセッサ間割り込み (IPI) を発生させてタスクレットの実行を委譲する機構を KACC に実装した．具体的には，割り込みハンドラとしてタスクレットの実行をスケジュールする関数を登録して IPI を実行することで，IPI ハンドラ中でタスクレットを目的の CPU にスケジュールし，IPI からの復帰時に目的のタスクレットを実行させる．IPI 自体は処理に時間のかかる高コストな操作であり，その分計算に割ける CPU 時間が減少するが，負荷分散によって集団通信の実行時間の短縮につながる．

© 2010 Information Processing Society of Japan

表 1 評価環境の緒元

CPU	Dual Core Opteron(2.0GHz) x 2
ノード数	8
ネットワーク	1Gbps Ethernet
OS	Linux 2.6.18 (RedHat Enterprise Linux)
MPI ライブラリ	MPICH2/TCP 1.0.6[8]

表 2 評価対象の非ブロッキングブロードキャスト実装

アルゴリズム	非ブロッキング通信手法
ツリー	KACC
ツリー	MPI 非ブロッキング 1 対 1 通信
パイプライン	KACC
パイプライン	LibNBC(pthread による実装)

5. 評　　価

KACC のテスト実装において，プロセッサ間割り込み (IPI) による負荷分散処理の影響を評価する．本評価はすべて，**表 1** に示す 8 ノードのクラスタ上で，32 プロセスの MPI ジョブを起動して行った．

本評価環境において，何もしない関数を割り込みハンドラとして設定した IPI の実行にかかる時間は割り込みハンドラの終了を待たない場合で平均 1.45 マイクロ秒，割り込みハンドラの終了を待った場合では 1.90 マイクロ秒であった．

IPI による負荷分散処理を有効にした場合と無効にした場合とで KACC による非ブロッキング版の `MPI_Bcast` の性能を比較した．比較対象として，既存実装として LibNBC の実装および `MPI_Irecv` 等を使った明示的な実装 (以下，MPI による実装) による性能も測定した．

本ベンチマークで用いたブロードキャストアルゴリズムの一覧を**表 2** に示す．LibNBC における実装 `NBC_Ibcast` は，図 1 で示したアルゴリズムと同様にメッセージをチャンクに分割し，順に隣接するノードへパイプライン状に転送するアルゴリズムである．MPI を用いた実装では，チャンク分割でメッセージ数を大きくすると明らかに不利になるので 2 分木に沿って順次メッセージをコピーしていくアルゴリズムで実装した．KACC における実装ではアルゴリズムの差を検討することは目的ではないので，MPI で実装したツリー方式と LibNBC で用いられているパイプライン方式の両アルゴリズムを実装し，それぞれと比較した．KACC の現時点の実装にはユーザプロセスが確保したメモリを通信に使えないという制約があるため，KACC を用いたベンチマークでは，KACC モジュールがカーネルのアドレス空間上に確保したメモリを用いて性能を比較している．

本ベンチマークでは簡単のため，通信中に固定サイズの行列乗算を行い，各回の計算が終わるたびに各実装における `MPI_Test` もしくは進捗処理にあたる動作を行い，通信が終わるまでにかかった時間とその間に実行した行列乗算の回数を計測した．実行時間と計算回数を通信を行っていない状態での行列乗算 1 回当たりにかかる時間と比較することで，計算以外の処理に費やされた CPU 時間を推定することが出来る．

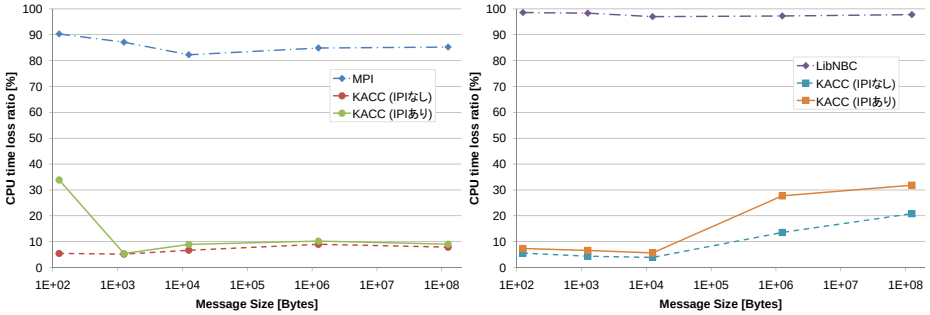
図 4 および **図 5** に，異なる計算処理の粒度における，各手法における計算以外の処理の CPU 時間の消費割合を示す．図 4 では，計算処理として 4×4 の行列乗算を，図 5 では，40×40 の行列乗算を用いた．

従来手法では，計算粒度を密にして通信の進捗および終了確認処理を頻繁に呼び出すと，ほとんどの CPU 時間が計算以外の処理に使われてしまうことがわかる．これに対して提案手法 KACC では計算粒度の変化は CPU 時間の消費量に与える影響はわずかであり，ほとんどの場合において従来手法よりも負荷が少ないことがわかる．

IPI による通信処理の負荷分散処理のオーバヘッドは，パイプラインアルゴリズムで大きなメッセージを扱うときに大きくなり，通信処理による CPU 時間の消費の約 3 分の 1 を占める．パイプラインアルゴリズムではメッセージをチャンクに分割するため，元のメッセージサイズが大きいほど通信の総数が増える．それぞれのメッセージの送信は手前のランクからの受信によって起動されるため，ネットワーク割り込みからタスクレットが起動される回数が多くなることが CPU 時間の消費を増大させている要因であると考えられる．現在の実装では通信の属する MPI ランクから処理を行う CPU コアを固定的に決めており，最大で 4 分の 3 の割り込みが IPI によって他コアに転送されている．この負荷分散手法を改善し，IPI の発行数を減らすことによってさらなる性能向上が期待できる．

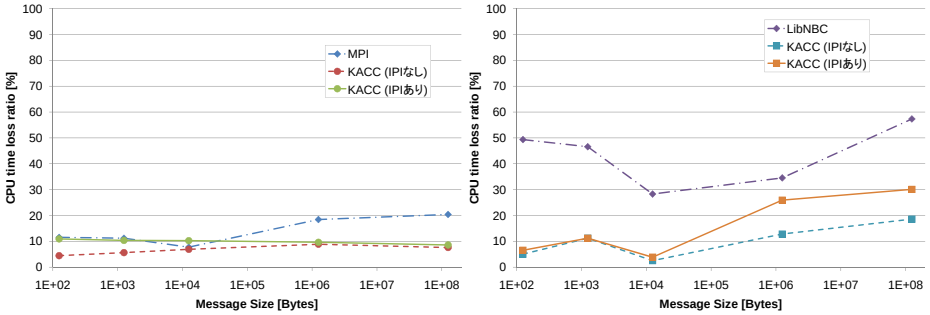
図 6 に，各アルゴリズムにおけるの比較対象の実装である LibNBC および MPI による実装を 1 としたときの KACC 実装におけるブロードキャスト処理の実行時間の比を示す．IPI による負荷分散を行わなかった場合の実行時間は比較対象の実装よりもほとんどの場合で低速であるのに対し，IPI による負荷分散を行うことで通信全体の実行時間の短縮に寄与していることがわかる．メッセージサイズが極端に小さいところでの性能は比較対象の手法に劣る場合があるものの，それ以外のところでは IPI による負荷分散が機能して比較対象の手法よりも短い時間で集団通信を終えている．ベストケースでは，LibNBC に対して

© 2010 Information Processing Society of Japan



(a) パイプラインアルゴリズム　(b) ツリーアルゴリズム

図 4　計算の粒度が密であるときの CPU 時間消費の割合

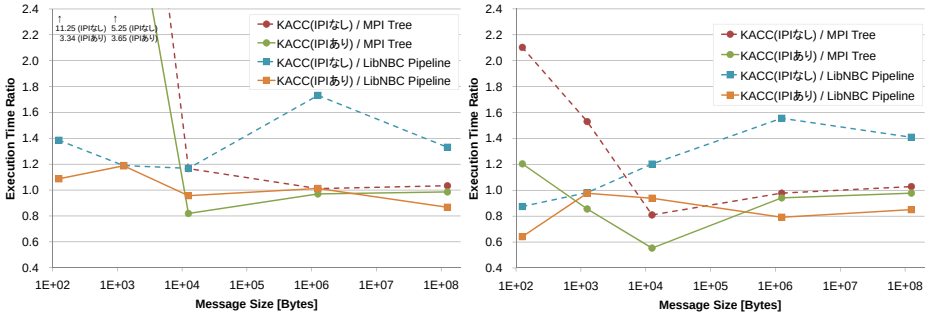


(a) パイプラインアルゴリズム　(b) ツリーアルゴリズム

図 5　計算の粒度が疎であるときの CPU 時間消費の割合

79%，MPI による実装に対して 55%の実行時間で集団通信が処理された．以上の結果からわかる通り，CPU コア間で Progress Engine の負荷分散を行うことは，集団通信の実行時間を短縮するために必要である．

以上の 2 つの考察から，IPI による Progress Engine の負荷分散処理は CPU 時間の側面からは高コストであるものの，従来手法ほどに CPU に負荷をかけるものではなく，通信全体の実行時間という側面において，十分に性能向上に寄与しているといえる．現時点の実装では，メッセージサイズが小さいときに通信に時間がかかっているという問題点があり，



(a) 計算の粒度が密であるとき　(b) 計算の粒度が疎であるとき

図 6　非ブロッキングブロードキャストの実行時間比較

その原因の解明と提案手法の改良は今後の課題である．

6. まとめと今後の課題

本報告では，MPI において非ブロッキング集団通信を実装する際における進捗処理の実装手法の問題点を示し，これを解決するために我々が提案した進捗処理を OS カーネル上で行う KACC 機構の実装上の問題点を述べた．KACC 機構では，通信の進捗処理を OS カーネルのネットワーク割り込みハンドラとしてプロセスコンテキストを用いずに起動することにより，進捗処理のタイミング問題とコンテキストスイッチのオーバヘッドの問題を解決する．OS カーネル内でタスクレットを用いて通信処理を行うために，タスクレットの負荷が特定の CPU コアに集中する問題が発生する．本報告では，タスクレットの起動時にプロセッサ間割り込み (IPI) を発生させてタスクレットの負荷分散を図る手法を提案した．

KACC のテスト実装において，集団通信の実行時間と実行中に CPU が計算処理を行った時間を比較した．IPI による負荷分散を実行した場合，従来手法である MPI の 1 対 1 通信関数の組み合わせで集団通信を実現する手法やスレッド方式で集団通信を行う LibNBC よりも少ない CPU 時間の消費で，高速に集団通信が行えることを示した．IPI による負荷分散処理は，特にメッセージの総数が多い場合に CPU 時間を多く消費するが，KACC において集団通信にかかる実行時間を短縮するためには，何らかの方法で進捗処理を行うタスクレットを CPU コア間で負荷分散しなければならないことを示した．

現時点の実装には以下に示す制限および問題点がある．まず始めに，通信対象のデータを

© 2010 Information Processing Society of Japan

置く領域にプロセスコンテキストを介さずにアクセスするために，使用メモリをカーネルモジュールから与えなければならない制限がある．この制限は一般の並列アプリケーション上で KACC を用いる際の障壁となっているため，この制限を解消すべく現在実装を進めている．この制限を解消でき次第，HPL[4] や他のアプリケーションにおいて，並列アプリケーション全体の実行時間に関する評価を行う予定である．

つぎに，メッセージサイズが小さいと，通信が比較的遅くなるという問題がある．また，ノード内通信への最適化が行われておらず，ノード内の通信に対しても TCP を用いているため無駄な処理が行われていると思われる．このような性能劣化の要因を調べ，実装を効率化していく必要がある．

また，MPI ではユーザ定義の演算子を集約操作に用いることが出来るが，KACC にはそれを実現する機構がない．OS カーネル内でユーザ定義の演算をするためには，ユーザプログラムへのコールバックを行う，もしくはユーザ定義演算ルーチンをバイトコードの形でカーネルに与えて検証しながら実行するなどの方法が考えられる．これらの方法について性能面およびセキュリティ面からの考察を行う必要がある．

最後に，現在の KACC には，P2P レイヤとして InfiniBand や Myrinet など，TCP 以外のデバイスに対する実装がない．本手法が実際の並列計算機上で使えることを示すためには，これらのデバイスに対する実装が必要である．

これらの実装上の問題点を解決することで，KACC 機構が MPI 3.0 の非ブロッキング集団通信の有力な実装となると考えている．

参 考 文 献

1) MPI Forum: Message Passing Interface, `http://www.mpi-forum.org/`.
2) Hoefler, T. and Lumsdaine, A.: Design, Implementation, and Usage of LibNBC, Technical report, Open Systems Lab, Indiana University (2006).
3) Nomura, A. and Ishikawa, Y.: Design of Kernel-level Asynchronous Collective Communication, *EuroMPI 2010* (2010).
4) Petitet, A., Whaley, R.C., Dongarra, J. and Cleary, A.: HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers, `http://www.netlib.org/benchmark/hpl/`.
5) Bovet, D.P. and Ph, M.C.: Understanding the Linux Kernel, Third Edition (2005).
6) MPI Forum: MPIplans - an alternative for all other collectives proposals?, `https://svn.mpi-forum.org/trac/mpi-forum-web/wiki/MPIplans`.
7) Hoefler, T., Lumsdaine, A. and Rehm, W.: 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, SC07*, IEEE Computer Society/ACM (2007).
8) Argonne National Laboratory: MPICH2 : High-performance and Widely Portable MPI, `http://www.mcs.anl.gov/research/projects/mpich2/`.

© 2010 Information Processing Society of Japan