T2R2 東京科学大学 リサーチリポジトリ
Science Tokyo Research Repository

# 論文 / 著書情報
# Article / Book Information

| | |
|---|---|
| 論題(和文) | 非同期コレクティブ通信の実装方式の検討 |
| Title(English) | Study in Implementing Asynchronous MPI Collective Communications |
| 著者(和文) | 野村 哲弘, 石川 裕 |
| Authors(English) | Akihiro Nomura, Yutaka Ishikawa |
| 出典(和文) | 情報処理学会 研究報告, Vol. 2011-HPC-130, , |
| Citation(English) | , Vol. 2011-HPC-130, , |
| 発行日 / Pub. date | 2011, 7 |
| 権利情報 / Copyright | ここに掲載した著作物の利用に関する注意: 本著作物の著作権は（社）情報処理学会に帰属します。本著作物は著作権者である情報処理学会の許可のもとに掲載するものです。ご利用に当たっては「著作権法」ならびに「情報処理学会倫理綱領」に従うことをお願いいたします。<br>The copyright of this material is retained by the Information Processing Society of Japan (IPSJ). This material is published on this web site with the agreement of the author (s) and the IPSJ. Please be complied with Copyright Law of Japan and the Code of Ethics of the IPSJ if any users wish to reproduce, make derivative work, distribute or make available to the public any part or whole thereof. |

Powered by T2R2 (Science Tokyo Research Repository)

# 非同期コレクティブ通信の実装方式の検討

野 村 哲 弘[†2]　石 川 裕[†1]

我々は以前，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，スレッド切替のコストを削減する手法であるが，OS カーネル上に実装されているために実際に利用するうえでの種々の障壁がある．本報告では，KACC 機構のユーザレベルでのプロトタイプ実装を行い，その過程で判明した性能上の考慮点を報告する．

# Study in Implementing Asynchronous MPI Collective Communications

Akihiro Nomura[†2] and Yutaka Ishikawa[†1]

We have proposed a method, called KACC, to execute non-blocking collective communications, which will be introduced in MPI 3.0, efficiently. KACC have reduced costs of thread-switching by implementing progression of collective communications into the OS kernel, but KACC is hard to be used by public users due to the modification to the OS kernel. In this paper, we report several performance issues, which are revealed during our preliminary implementation of KACC facility into user-level.

## 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った．しかしながら，OS カーネルの改変をともなうオリジナルの KACC 機構は，計算機環境の管理者権限を持つユーザにしか用いることができず，T2K オープンスパコンや TSUBAME 2.0，京コンピュータに代表される大規模計算環境では使うことができない．そのため，我々は KACC 機構を敢えて OS カーネルへの改変を行わずにスレッドを用いて実装し，そのうえで現時点でどのような問題が起こるのかについて検討を行った．

## 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，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
†2 東京工業大学
Tokyo Institute of Technology


© 2012 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．CPU のみの並列アプリケーションのユーザは一般的に MPI プロセスや OpenMP のスレッドに代表される実行コンテキストをノード内の CPU コアと同じだけ作成し，利用可能なすべての 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，Linpack ベンチマークの実装の一つである High-performance Linpack(HPL)[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．このように明示的に進捗処理を呼ぶ方式では，非ブロッキング集団通信のユーザに多大な負担を強いるものであるため，一般化したライブラリとして提供するにはそぐわないことがわかる．

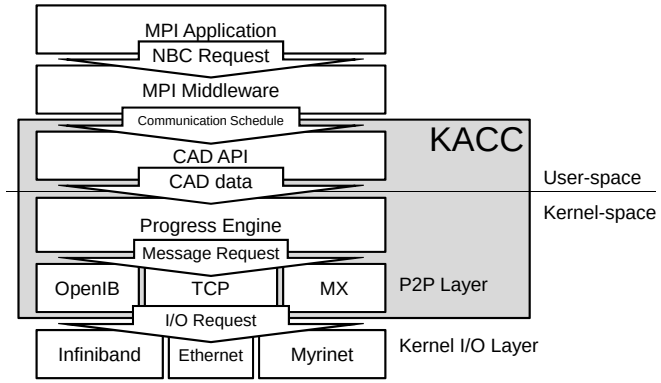


**図 1** KACC 機構のデザイン

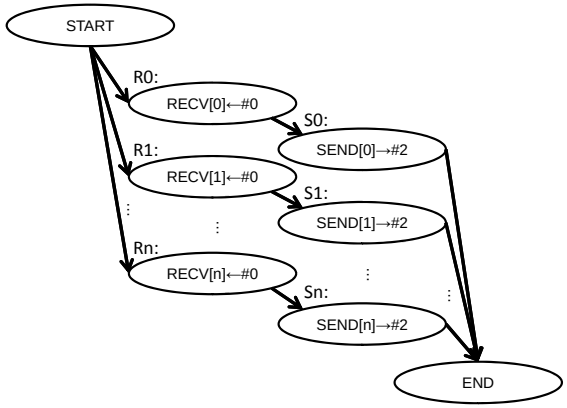


**図 2** CAD グラフの例

## 3. オリジナル KACC 機構の概要と問題点

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．KACC 機構は **図 1** に示すように，CAD API，Progress Engine，P2P の 3 層に分かれている．ユーザは


© 2012 Information Processing Society of Japan

```
 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 3** 図 2 に対応する CAD グラフを生成するコード

非同期コレクティブ通信のデザインを**図 2** のように MPI ランクごとに DAG の形で実装し，**図 3** のような API 呼び出しで 1 つの非同期コレクティブ通信リクエストとしてまとめ上げる．Progress Engine(PE) 層では CAD API で作られたリクエストを辿って適宜必要な通信を P2P 層の API を呼ぶ形で実行する．当然 PE 層と P2P 層の処理は並列アプリケーションから見て非ブロッキングで実行される必要があり，オリジナル KACC では Linux の軽量スレッド機構である tasklet[5] を用いて実装した．

しかしながら，tasklet を用いた実装を行うことによって，以下の 2 点が問題となった．第 1 に，通信を OS カーネル側から行う必要があるため，通信に必要な TCP ソケットやその他のハンドル (以下，ソケット類と表す) をユーザレベルにある MPI アプリケーションとは別個に持つ必要がある点である．このため，このようなソケット類に必要な資源が倍になってしまう点が，スケーラビリティを阻害する点で問題がある．また，通信処理が独自実装である故に，複数リンクのアダプティブな利用のような他の通信を最適化する機構を導入できない点も問題である．もう一つの問題点は，Linux tasklet は OS タスクコンテキストを持たないために，通信を OS カーネル上に tasklet がスリープしないように実装する必要があるが，たとえば Linux の TCP ドライバは元来そのような作りにはなっておらず，オリジナル KACC ではカーネルの TCP 実装を詳細に検討して各カーネル API がタスクスイッチを要求する条件を調べ上げて回避するように作る必要があり，OS カーネルの変更に追随することを難しくしている．

## 4. スレッド方式での実装上の問題点

これらの問題点を克服するため，KACC 機構をユーザレベルで実装することを考える．このことにより，P2P 層として既存の MPI の非ブロッキング 1 対 1 通信をそのまま使えるようになるため，前節で述べた種々の問題点を回避することができる．しかしながら，PE 層と P2P 層をそのまま OS のスレッドを用いて実装してしまうと，LibNBC で発生している問題をそのまま再現するだけとなってしまう．本節では，LibNBC やその周辺の実装における問題点に着目し，その対処法を検討する．

### 4.1 OS レベルスレッドを用いることによるコスト

OS のタスク管理機構を用いたスレッドは，タスクの切り替え時にプロセスに属するすべてのデータ構造を退避して，読み込む必要がある．また，CPU コアとスレッドを関連付けるために `numactl` コマンドや `sched_setaffinity` システムコールを用いることがあるが，このような処理は MPI や非ブロッキング通信の実装を知らないと使えない上に，非ブロッキング通信機構のようにライブラリ側でスレッドを作成する機構ではユーザが後から指定することにも限界がある．

### 4.2 アルゴリズム表現の自由度

LibNBC の実装では，アルゴリズムの記述を簡便にするため，Collective Schedule と呼ばれる KACC とは違うアルゴリズム記述方式を持つ．Collective Schedule では，集団通信を構成する個々の 1 対 1 通信をラウンドと呼ばれるグループに分けて配置する．同じラウンドに属する処理は同時に実行可能であり，各ラウンドは 1 つ前のラウンドに属する全通信が終了すると実行可能になる．このような比較的単純なモデルを用いることで，アルゴリズムの見通しは比較的良くなっているが，その一方で Collective Schedule が扱えない通信パターンが性能を左右する可能性がある．メッセージサイズの大きい通信ではしばしばメッセージを分割して受信と送信をパイプライン的に行う実装が考えられる．このような実装では，同じピースの受信と送信を分けるためにこれらを別のラウンドに分け，1 つのラウンド


© 2012 Information Processing Society of Japan

内では $k$ 個めのピースの送信と $k+1$ 個めのピースの受信を同時に行うように記述される．

一方で，MPI ライブラリは通信時にイーガー送信・ランデブー送信と呼ばれる異なる送信戦略をメッセージサイズに応じて選択することが多い．イーガー送信では，送信側は比較的小さいメッセージを受信側の受信命令の発行の有無にかかわらず実行する．受信側で受信命令が発行されていないときには，MPI ライブラリ側で確保した領域 (Unexpected queue) にメッセージを保存し，受信命令が発行された時点で改めてアプリケーションのメモリ領域にコピーを行う．一方ランデブー送信では，送信側はメッセージ本体を送る前に，受信側に対して受信命令の有無を確認し，受信命令が発行されていない場合には受信命令の発行まで送信を遅延させる．いずれの場合にも，送信命令の発行タイミングが受信命令の発行タイミングに先行すると，逆の場合に比べてある程度のコストが発生する．

ここで，先述のようなパイプライン転送が実行された場合，各ノードにおけるラウンドの切り替わりのタイミングはほぼ同時であり，それぞれのメッセージの送信命令と受信命令はほぼ同時に各ノードで実行される．言い換えれば，どちらの命令が先行するかは確率的であり，最適な通信順序が保障されない．この状態を脱するには，受信命令をそれよりも前に発行しておけばよいのであるが，Collective Schedule では 1 つの命令が複数のラウンドにまたがって存在できず，次のラウンドへ進むには現在のラウンドの全ての処理を完了させなければならないため，そのような実装は不可能である．

このように，KACC 機構の ADG のようなより一般的な依存関係を表すデータ構造が効率の良い通信の実現にとって必須である．

### 4.3 ブロッキング通信の実装における問題

LibNBC による手法および我々のユーザレベルの KACC 双方に共通する問題点として，ブロッキング通信の実装方式があげられる．スレッドを用いた集団通信の実装では，アプリケーションのスレッドとは別のスレッドでブロッキング通信 (`MPI_Isend` のような非ブロッキング通信を `MPI_Wait` などで待つ場合を含む) を行うことで，通信の終了を検知して次の通信を開始する．しかしながら，多くの MPI 実装ではこのような通信の待ち方を想定しておらず，MPI 待ちのスレッドが CPU コアを占有することが可能であるという仮定を持っているため，ビジーウェイトやタイマーを用いたポーリング型の待ち処理を行っている．このようなシステムでは，`epoll` のような OS のシステムコール待ちでスレッドの実行を止めるのに比べて CPU に負荷がかかり，スレッドのタイムスライスを消費した結果の優先度低下や，CPU コア数が足りないときのスケジューリングによる性能低下を招く．

このように，非ブロッキング集団通信機構を実装する際には，元となる MPI 処理系の通信待ち処理についても考慮しなければならない．

## 5. ユーザレベル KACC 機構の実装

このような点を考慮に入れて，ユーザレベルの KACC 機構を NewMadeleine 通信ライブラリ群を用いて実装した．NewMadeleine は，MPI 実装 MadMPI，ユーザレベルスレッドライブラリ Marcel，通信スケジューラ PIOMan などのコンポーネントからなる通信ライブラリ群であり，INRIA Bordeaux で開発されている．NewMadeleine アプリケーションではこれらのコンポーネントを有機的に結合することで，すべての通信処理を集中管理して最適化を行うことができる．本実装では Marcel スレッドライブラリと PIOMan 通信スケジューラを併用することで，進捗処理スレッドと NewMadeleine の通信スレッドを繋ぎ，低コストでのタスク切り替えを目指した．

ユーザレベル KACC の実装におけるオリジナル KACC からの変更点は，P2P 層の実装と PE 層および P2P 層が実行されるスレッド機構の実装である．P2P 層は MadMPI の通信機構をそのまま使うことで，他の通信メッセージと同じチャネルを使えるようにし，NewMadeleine の最適化の恩恵を受けられるようにした．PE 層と P2P 層の実装では，Linux tasklet を割り込みハンドラによって呼び出す代わりに Marcel のスレッドを CAD 上のそれぞれのタスクごとに作成することで，P2P 層をブロッキング通信を用いて実装することができる．そのため，タスクの選択を Marcel のスケジューラに任せることができるようになった．さらに，Marcel ではユーザスレッドに OS のタスク優先度とは別の優先度を付けることが可能である．pthread を用いた実装では，アプリケーションスレッドよりも高優先度のスレッドをあとから作るといった処理が難しかったが，Marcel を用いることでこれが可能になった．

## 6. 評　　価

ユーザレベル KACC のテスト実装において，同一の非ブロッキングブロードキャストアルゴリズムの実装を行い，その性能の傾向を比較した．実験環境は**表 1** に示す通りである．KACC のカーネルモジュールの安定性の問題と，LibNBC が MadMPI 上で実装されていない関数を使っている点および NewMadeleine 通信ライブラリが要求する各種ライブラリのバージョンの問題で，同一のソフトウェア環境上でテストすることはできなかった．

また，現時点の MadMPI の通信機構では，ポーリング型の通信待ち処理しか実装されていないため，各ノードでは 2 つの MPI プロセスをそれぞれ別の 2 つの CPU コアに関連付


© 2012 Information Processing Society of Japan

**表 1** 評価環境の緒元 (オリジナル KACC)

| | オリジナル KACC | ユーザレベル KACC |
|---|---|---|
| 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 (RHEL 5 相当) | Linux 2.6.32 (RHEL 6 相当) |
| MPI ライブラリ | MPICH2/TCP 1.0.6[6] | MadMPI |

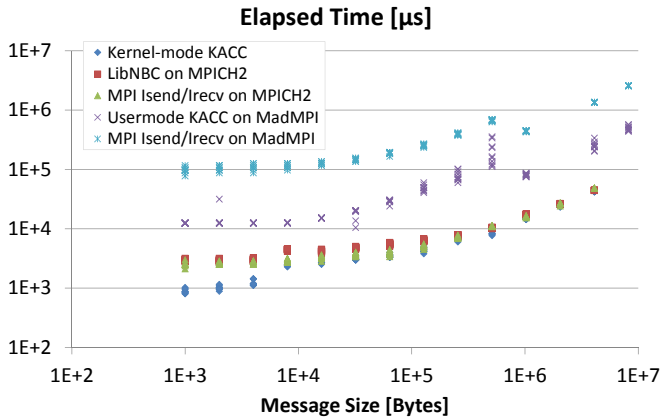


**図 4** 各実装手法における非ブロッキングブロードキャストの実行時間

けて実行した．このため，過去の報告で述べた LibNBC の欠点である同一 CPU コア上でのスレッド切替によるコストは隠蔽されている．

本評価では，図 2 で示したようなパイプライン的にメッセージを流すブロードキャストアルゴリズムの実行時間を KACC 機構，LibNBC およびスレッドを使わずに `MPI_Test` 等を定期的にアプリケーションスレッドから実行した際の実行時間の変化を調べた．バックグラウンドの計算にはコアごとにローカルな行列積演算を実行し，通信終了時にまでにどれだけの量の演算ができたかを調べた．**図 4** にそれぞれのアルゴリズムでのブロードキャスト実行時間，**図 5** に CPU 時間の通信処理による消費割合を示す．後者の値は測定された通信時間の間計算処理だけを行っていた場合に実行できたであろう演算回数と実際に行われた演算回数の比から通信処理につかわれた CPU 時間を予測したものである．

MPICH2 上での KACC の通信では，前回の報告のとおり低 CPU 時間コストで高速な

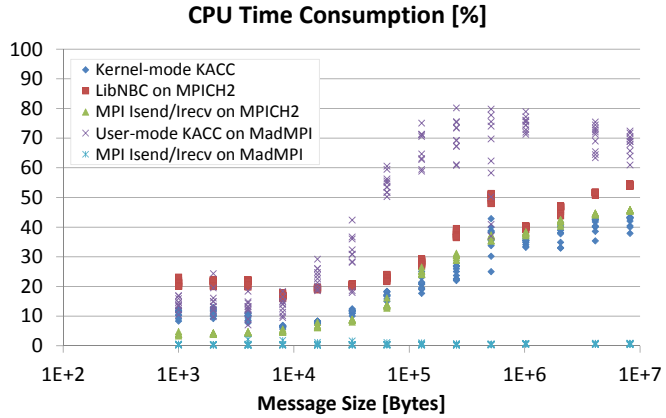


**図 5** 各実装手法における非ブロッキングブロードキャストの CPU 時間消費の割合

非ブロッキング集団通信が行えていることをがわかる．しかしながら，MadMPI 上での通信では期待された性能が一切出ていない．これには以下のような理由が考えられる．ひとつは，PIOMan とユーザ定義の Marcel スレッドを組み合わせて実行しているために Marcel のスレッドの動作が複雑化しており，コア数以上のアクティブなスレッドができてしまっている可能性がある点である．もうひとつは，NewMadeleine の通信がスレッド数に余裕があるときにはビジーウェイト，余裕がない時には 10ms ごとの定期的なポーリングで実現されているために，OS 上でのスリープで期待されるような通信終了時の継続処理の即時の再開がなされていない点である．これらは NewMadeleine の通信部分を見直して，より適切な動作ができるように修正する必要がある．

## 7. まとめと今後の課題

本報告では，MPI 非ブロッキング集団通信の OS カーネルにおける実装である KACC 機構をユーザレベルへ移植するうえでの実装上の問題点を通じて，非ブロッキング集団通信の実装に必要な技術的条件を示した．現在の MPI 通信ライブラリおよびスレッドライブラリの実装では，非ブロッキング集団通信を効率よく実装することが難しいことを示した．

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が，以上の推論を裏付ける結果を出すことはできなかった．今後の課題として，ユーザレベルの実行環境の挙動とボトルネックを明らかにして，このような推論が正


© 2012 Information Processing Society of Japan

しいかどうかを明らかにする必要があると考えている．

また，ユーザレベルに KACC を移植した本来の目的である大規模共用計算機環境での評価も取る必要があると考えている．

## 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) Argonne National Laboratory: MPICH2 : High-performance and Widely Portable MPI, `http://www.mcs.anl.gov/research/projects/mpich2/`.



© 2012 Information Processing Society of Japan