2003－HPC－95　(18)
2003／8／5

MPI通信モデルに適した通信APIの設計と実装

松田元彦[†]　石川　裕[††]
工藤知宏[†]　手塚宏史[†]

大規模クラスタ計算機やグリッドに向けたMPIを実装するための通信機構としてO2Gドライバの設計・実装を行なった．O2Gは通信レイヤ自体を変更することなく，オーバーヘッドが大きいと考えられるソケットAPIをバイパスする．それにより性能問題が懸念されるselect&readによるポーリングを排除し非同期通信処理の効率化を狙う．そのためO2GではMPIで必要になる受信キュー操作をすべてプロトコル処理ハンドラ内で実装する．O2Gは現在Linuxのローダブル・ドライバとして提供される．評価として，NAS並列ベンチマークを用いたMPICHとの比較を行なう．非同期通信が重要となるISベンチマークではO2Gを用いた実装はMPICHの2.8倍の性能を示す．他のベンチマークの結果も互角の性能であり，O2Gを用いる実装に無駄なオーバーヘッドがないことを示す．

The Design and Implementation of Kernel API for MPI Communication Model

Motohiko Matsuda,[†] Yutaka Ishikawa,[††] Tomohiro Kudoh[†]
and Hiroshi Tazuka[†]

O2G is a messaging kernel interface designed to implement MPI for large-scale clusters and networks in the Grid environment. O2G cares asynchronous communication primitives, and totally avoids the select&read system call loops. For this purpose, O2G provides the message queue management of MPI in the driver, where all message processing is completed in the protocol handler. Currently, O2G is provided as a loadable driver module of the Linux kernel. Evaluation using NAS Parallel Benchmarks shows that an MPI implementation with O2G performs better than MPICH for all benchmarks. Especially, it performs 2.8 times faster than MPICH for the IS benchmark. The results show that the O2G's approach is efficient and has no excessive overheads.

1. はじめに

大規模なクラスタ計算機が利用されるようになり，MPI (Message Passing Interface)[1] の実装も多数のノードからの通信を扱う必要が出てきた[2]．通信にはWAN，LANを問わずTCPによるコネクション指向のストリームが使用される．一方，Linux, Unix, Windowsといった OS の通信 API は，通信エンドポイントを表すソケットで提供される．元々ソケットはプロセス当たりのソケット数が小さい場合を扱うように設計されてきたため，多数のソケットからメッセージを受信する場合の性能が低い．

複数ソケットからの受信処理は，select&readループという2つのシステムコール列の繰返しによって構成される．まずselectにより通信イベントを検出し，その結果に従ってreadによってデータを読み出すという操作を繰り返す．ポーリング・ベースの実装は，システムコールの回数が多くなる．また，通信データを読み出すタイミングが遅れるためメッセージが停滞し通信性能が低下する．

これはOSとユーザープロセス間のAPIの問題である．OS内部ではI/Oは非同期に処理されており，通信に関してもコネクション数に対して性能はスケーラブルである．しかし，ユーザープロセスは逐次的にモデル化されており非同期イベントの処理効率が悪い．アプリケーションによっては，マルチ・スレッドの使用でソケット数に影響されない処理が可能である．しかし，MPIのアプリケーションは単一スレッドで記述されることが多くマルチ・スレッドを活用できない．

問題を解決するため，ユーザーレベル通信，イベント検出の高速化，非同期I/Oなどが考案されてきた．ユーザーレベル通信は汎用性がなく不都合な場面が多い．特にネットワークのハードウェア，ドライバ，TCPの実装は日々改良されており，それに追従できない．汎用性の高いイベント検出の高速化や非同期I/Oはそれ自体の効果は高いと考えられる．しかし，MPIの実装では高頻度のシステムコールを必要とする点では変

† 産業技術総合研究所
National Institute of Advanced Industrial Science and Technology
†† 東京大学 大学院情報理工学系研究科
University of Tokyo

```
/*ブロッキング通信関数*/
int MPI_Send(void *buf, int siz, MPI_Datatype typ, int dst, int tag, MPI_Comm com);
int MPI_Recv(void *buf, int siz, MPI_Datatype typ, int src, int tag, MPI_Comm com,
             MPI_Status *res);
/*ノンブロッキング通信関数*/
int MPI_Isend(void *buf, int siz, MPI_Datatype typ, int dst, int tag, MPI_Comm com,
              MPI_Request *req);
int MPI_Irecv(void *buf, int siz, MPI_Datatype typ, int src, int tag, MPI_Comm comm,
              MPI_Request *req);
```

図 1 MPI 送受信関数 API

わりなく，高い効率は期待できない．

問題は OS の API にあるので，通信レイヤを変更することなく問題を解決できるはずである．そこで我々は MPI 専用の API を実装する O2G ドライバの設計・実装を行なった．O2G はソケット API 層のみを置き換える．現在，O2G は Linux のローダブル・ドライバとして提供され，実装は非常に単純である．

本稿では，O2G ドライバの設計・実装とその性能評価を述べる．以下では，2 節でソケットの問題点，3 節で MPI 実装の基本動作，4 節で O2G の設計・実装について述べる．続く 5 節で NAS 並列ベンチマークを使った性能評価，6 節で実装に関する議論，7 節で関連研究を述べる．最後に 8 節でまとめを述べる．

2. ソケットの問題点

2.1 MPI 通信モデル

MPI (Message Passing Interface)[1] の基本操作は，メッセージ送信関数 MPI_Send と受信関数 MPI_Recv であり，これらの通信はブロックする．ノンブロック通信には MPI_Isend と MPI_Irecv を用いる．MPI 送受信関数 API を**図 1** に示す．送信側はデータバッファと，タグおよび送り先を指定してメッセージを送信する．受信側はデータバッファと，タグおよび送り元を指定してメッセージを受信する．タグ，送り元，送り先は整数で指定されるが，受信側ではタグや送り元にワイルドカードを使用することができる．タグおよび送り元／送り先がマッチする関数呼び出し間でメッセージが通信される．

2.2 ソケットの問題点

通信は TCP をベースとした 1 対 1 のコネクション指向ストリームであり，ソケットに対しては read/write といったシステムコールが利用される．これらの処理は基本的にブロックする．ソケットにはイベント待ちやポーリングを行なうための API として select システムコールが用意されている．ソケットでは非同期的な通信は select&read を繰り返すコード列で実現される．

MPI の実装には，常にストリームからデータを読み出し続ける必要があり，その実現には非同期通信を行なう必要がある．ストリーム上で MPI のメッセージを受信する場合，目的タグを持つメッセージがストリームの先頭にあるとは限らないため，常にストリームからデータを読み出し続ける必要がある．また，TCP では受信バッファサイズをウインドウサイズとして伝達するフロー制御を行なっている．メッセージが受信バッファに溜ると，フロー制御が働く．これはエンド・ツー・エンドのフロー制御であり，通信遅延によるフロー情報伝達の遅れが通信性能を悪化させる．

MPI の実装ターゲットとして，大規模クラスタ計算機を想定する必要がある．さらに，グリッドなど WAN による遅延時間やバンド幅のバラツキが大きいネットワークを想定する必要も出てきた．ここで，ソケット API には大きく三つの問題がある．

第一に，select&read ループの繰返しによるポーリング・ベースの実装ではシステムコールの頻度が高い．一般に OS のシステムコールはオーバーヘッドの大きい処理である．

第二に，ソケットに対する read がブロック操作であるため，他のソケットに対する非同期的な読み出しが停滞する．これを回避するには一回の read の読み出しサイズを小さくし，select&read ループの繰返し頻度を高くする必要がある．しかしこれはシステムコール・オーバーヘッドの増大を招く．特に広域通信では，データ転送が低速であるため read がプロセスをブロックする時間が長くなる．

第三に，Linux や Unix における select 処理の実装は規模に対して性能がスケーラブルでない．select の実装にプロセスがオープンしているソケット数に線形な処理量のアルゴリズムが使用されているためである．

3. MPI 実装の基本動作

3.1 受信キュー

MPI の標準的な実装 (例えば MPICH[4]) では，受信処理には基本的に 2 つのキューを利用する．

- 受信メッセージ・キュー
- 受信リクエスト・キュー

受信メッセージ・キューは unexpected キューとも呼ば

れ，受信はしたが，まだ MPI_Recv/MPI_Irecv による受信リクエストが発行されていないペンディング状態にあるメッセージを保持する．受信メッセージ･キューのエントリには受信メッセージ中のタグ，送り元，メッセージ内容のデータが記録される．MPI_Recv/MPI_Irecv により対応する受信リクエストが発行されると，このエントリからデータがコピーされる．

受信リクエスト･キューは expected キューとも呼ばれ，メッセージ受信受け入れ状態にあるリクエストを保持する．これは，MPI_Recv/MPI_Irecv により受信リクエストが発行されたが，メッセージがまだ到着していないリクエストのキューである．キューのエントリにはタグ，送り元，バッファ･アドレスが記録されている．メッセージを受信した場合，このキューのエントリにマッチするものがあるかどうかチェックを行なう．

3.2 受信リクエスト発行

MPI_Recv や MPI_Irecv の呼び出しにより，受信リクエストが発行される．この時まず，受信メッセージ･キューを調べて，既に受信したメッセージ中にリクエストにマッチするものが存在するかチェックする．もしマッチするエントリが存在すれば，そのメッセージにより受信リクエストを完了する．一方，マッチするエントリがない場合，リクエストを受信リクエスト･キューに挿入する．

3.3 メッセージ受信動作

MPI のランタイムがメッセージを受信した時は，まず受信リクエスト・キューを調べ既にマッチする MPI_Recv/MPI_Irecv が発行されているかチェックする．もしマッチするエントリが存在すれば，エントリに記録してあるバッファ･アドレスにメッセージ･データを書き込む．一方，マッチするエントリが存在しない場合，メッセージを受信メッセージ･キューに挿入する．以後，マッチする MPI_Recv/MPI_Irecv が発行された時点でデータを書き込む．

3.4 ランデブ・プロトコル

標準的な MPI の実装では，大きなサイズのメッセージを転送する場合にランデブ・プロトコルを用いる．ランデブ･プロトコルでは送信リクエストと受信リクエストの両方が揃ってから実際のデータ内容の転送を行なう．このためランデブでは，送信リクエストと受信リクエストの発行の際にハンドシェークを行ない，ハンドシェークが完了してからデータの通信を行なう．

ランデブを行なうと性能が通信遅延に影響を受けることになる [5]．しかし，ソケットによる MPI の実装ではデータのフロー制御のためにランデブが必要である．受信側で read を行なっていないソケットへ送信側が write する場合，ソケットへの write がブロックすることがある．ソケットへの write がブロックすると，そのプロセスは他のソケットを read しないので，全体としてデッドロックが起こり得る．

ランデブには，データのコピーを減らす効果もあるが，メモリ・バンド幅の向上は通信バンド幅の向上を上回っており，コピーを回避する重要性は相対的に小さくなっている．また，MPI のアプリケーションでは，MPI_Irecv を十分早く呼び出すことによりコピーが必要になる確率を小さくするコーディングが好んで用いられる．

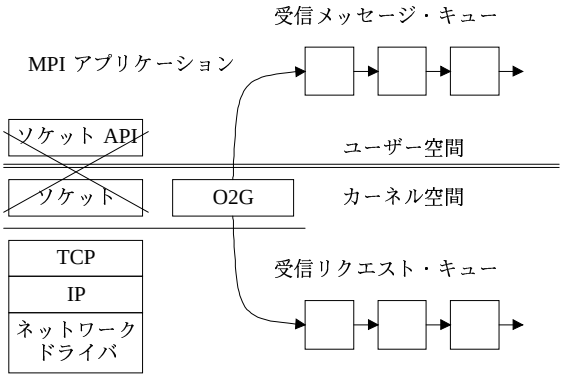

図 2 O2G 動作概要

4. 非同期受信ドライバ O2G の設計・実装

4.1 O2G ドライバの設計

O2G は Linux のローダブル・ドライバとして実装され，受信側の処理を最適化する．O2G の目的は単純であり，届いたメッセージをただちに通信レイヤから読み出すことである．MPI 実装の基本処理の節で述べた，受信メッセージ・キューおよび受信リクエスト・キューの処理をドライバ内で行なう．データ受信時のすべての処理はプロトコル処理ハンドラで処理される．

図 2 に O2G の動作概要を図示する．受信メッセージ・キューはデータ領域を含むため，ユーザー領域内にバッファを取っている．受信リクエスト・キューは受信メッセージの検索に使用するので，カーネル領域内に保持する．受信リクエストはデータを含まないので，メモリ使用量は少ない．

メッセージはソケットへの受信とほぼ同様に処理される．ソケットの処理では，受信データは Linux のプロトコル処理バッファである SKB にコピーされ，ソケット毎にある受信キューに挿入される．一方，O2G の場合，ソケットの受信キューに挿入されたデータはすぐに読み出され受信処理が行なわれる．受信メッセージに対する受信リクエスト・キューへのマッチング，あるいは受信メッセージ・キューへの挿入がただちに行なわれる．

O2G で必要になる処理としては，MPI ヘッダの解析，マッチするキュー・エントリの検索，キュー・エントリの作成がある．メッセージ・データのコピー自体は read と同様の処理である．これらは，ソケット

```
/*初期化関数*/
o2g_init(int n_socks);
o2g_register_socket(int sock, int rank);
o2g_set_dump_area(void *area, int size);
o2g_start_dumper_thread(int n_thrds);
/*エントリ操作関数*/
o2g_put_entry(struct queue_entry *e);
o2g_cancel_entry(struct queue_entry *e);
o2g_free_entry(struct queue_entry *e);
o2g_poll(void);
```

図 3 O2G ユーザー API

に対する処理に比べて大きなオーバーヘッドとはならない．

O2G を用いる MPI ではランデブ・プロトコルは使用していない．O2G では常にデータの読み出しを行なうため，受信メッセージ・バッファ領域に十分なサイズを割り当てている場合デッドロックの可能性はない．よって遅延に対して性能が低下するランデブを行なう必要はない．

4.2 ドライバ関数

O2G は Linux のデバイス・ドライバとして実装されており，ユーザープロセスからは ioctl システムコールを通じて制御を行なう．その制御をライブラリ化した API を**図 3** に示す．

初期化関数群では，`o2g_init` は初期化処理を行なう．`o2g_register_socket` はソケットと相手プロセスのランクを結び付け，O2G を使用するためにソケットを設定する．`o2g_set_dump_area` は受信メッセージ・キューの作成エリアを指定する．`o2g_start_dumper_thread` は処理スレッドを起動する．このスレッドは後ほど説明するコンテクスト不一致時の処理を行なう．

エントリ操作関数群では，`o2g_put_entry` は受信リクエストをキューに挿入し，`o2g_cancel_entry` はそのリクエストをキャンセルする．`o2g_free_entry` はユーザープロセス空間にある受信メッセージを解放する．`o2g_poll` はメッセージが受信されるまでプロセスをブロックする．

4.3 ドライバ起動フック

Linux ではカーネル内で NFS サーバーを実現している．そのため，SUN RPC を実装するための機構としてソケットの受信処理にフックが設定できる．通信レイヤは受信データを SKB バッファ・データとして受信キューに入れるが，その後のデータ処理に任意の関数を設定できる．フックはカーネル内のソケット構造体 `struct sock` 中の `data_ready` という関数へのポインタとして定義される．

図 4 にソケット受信フック関数とその使用例を示す．フック関数 `data_ready` は SKB が処理される毎に呼び出される．ここで `tcp_read_sock` は処理を簡潔に

```
{
  /*ソケット層のフック関数の設定*/
  struct sock *sk = ...;
  sk->data_ready = data_ready;
}
void data_ready(struct sock *sk,
    int len) {
  tcp_read_sock(sk, ..., data_recv);
}
int data_recv(..., struct sk_buff *skb,
    unsigned int off, size_t len) {
  char *buf=...;
  skb_copy_bits(skb, off, buf, len);
}
```

図 4 ソケット受信フック関数とその使用例

するためのユーティリティ関数である．`data_recv` 関数内に実際の処理を記述するが，ここではカーネル内のバッファ`buf` にデータをコピーする例を挙げている．

Linux ではこのように簡単に受信データを利用することが可能であり，O2G はこの機能を利用して実装されている．

4.4 ドライバ処理動作

フックにより起動されたドライバは，メッセージ・ヘッダを解析する．ヘッダにはタグ，送り元，メッセージのデータサイズ等が記録してある．O2G ドライバはヘッダ情報を使って，受信リクエスト・キューを検索する．

もし受信リクエスト・キューにマッチするエントリが見つかった場合は，メッセージ・データをユーザー領域にコピーする．その後，エントリ中の受信フラグをセットしエントリを解放する．

もし受信リクエスト・キューにエントリが見つからない場合は，受信メッセージ・キューに新たなエントリを割り付ける．エントリにはメッセージ・サイズのバッファ領域を割り当てる．ヘッダを受信した時点では，受信フラグをリセットした状態のエントリを作成する．以後，データを受信する毎にバッファにコピーを行なう．すべてのデータの受信が完了した時点で，受信フラグをセットし処理が完了する．

4.5 コンテクスト不一致時の処理

O2G では，受信処理がすべて割り込みで起動されるプロトコル処理ハンドラ中で実行される．もし，割り込み時点のプロセス・コンテクストがユーザープロセスでない場合，プロトコル処理ハンドラからユーザー領域へのデータ書き込みが行なえない．この場合は，O2G はユーザーのスレッドを起動し，そのスレッドによってデータの書き込みを行なう．このために，あらかじめ `o2g_start_dumper_thread` を呼び出して書き込み用のスレッドを起動しておく．

同様に，ページフォールトが起こる場合も，ユーザー

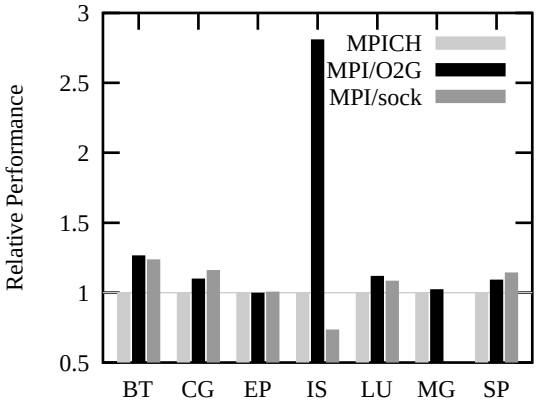

図 5 性能比較

スレッドを起動して書き込みを行なう．

このような書き込みに対するプロセス切替えは，ソケットを使用する read の場合も起こり得るものである．ソケットの場合は，read でブロックしているユーザープロセスが起動されることになるが，処理は全く同等であり，O2G にだけに必要となる特別なオーバーヘッドではない．

4.6 競合状態の回避

O2G による処理はプロトコル処理ハンドラで実行されるため，ユーザープロセスから呼び出されるキュー・エントリの操作とは競合状態が存在する．`o2g_put_entry` により受信リクエスト・キューにエントリを追加する時点で，マッチするメッセージが受信された場合などである．その場合，受信メッセージ・キューに現れているはずであるが，チェックのタイミングにより発見できない場合が起こる．競合状態の可能性を検出した場合，`o2g_put_entry` はエラーとして EAGAIN を返す．MPI の実装は EAGAIN が返ってきた場合，受信メッセージ・キューのチェックをもう一度行なう．これにより競合状態を回避できる．`o2g_cancel_entry` にも同様に競合状態があり，EAGAIN を返すことがある．

5. 性 能 評 価

5.1 ベンチマーク環境

NPB (NAS Parallel Benchmarks[3]) (バージョン 2.3) を使用して性能評価を行なった．データサイズはクラス A である．データはベンチマークの表示のうち「Mop/s total」値を比較している．

評価には，16 台の PC からなるクラスタを用いた．プロセッサは Pentium III 933 MHz，メモリは 1 GB，OS は Linux 2.4.20 ベースである．プロセッサボードは 2 CPU 構成であるが，実験では 1 ノード当たり 1 プロセスで実行した．ギガビット・イーサネット NIC は 3Com 製 3C996B である．コンパイラは gcc (バージョン 2.96)，コンパイル・オプションはすべて-O3 を使用した．

比較対象は代表的な MPI 実装である MPICH[4] である．MPICH のバージョンは 1.2.5 を使用した．また，O2G 方式がソケットと同様に効率良く実装できているかを確認するため，ソケットを用いる実装との比較も行なった．ソケットによる実装と O2G による実装は，受信部分以外では大部分のコードを共有している．

5.2 ベンチマーク結果

図 5 に MPICH との相対性能を示す．MPI/O2G が O2G による実装，MPI/sock がソケットを用いる実装である．

MPI/O2G は，すべてのベンチマークにおいて MPICH よりも高い性能を示した．特に IS においては O2G は MPICH の 2.8 倍の性能を示した．IS は全対全通信を行なっておりソケットを用いる通信では通信が停滞することが予想され，非同期通信に向いたベンチマークである．一方，IS を除く他のベンチマークでは，それほど差異はない．ベンチマークのコードを見ると，通信のほとんどにブロックする 1 対 1 通信を使用している．1 対 1 通信では，受信するソケットを特定できるため select を行なう必要なく，また複数のストリームから同時に受信することもない．これは，ソケットを用いる実装でも性能上の問題のない通信パターンである．

MPI/O2G と MPI/sock の実装を比べた場合，CG と SP の 2 つのベンチマークで性能が低くなっている．ただしその差異は 6%以下であり，コードが最適化されているソケットに比べても O2G の実現方式は十分効率の良いものであると言える．MPI/sock では MG で結果が出ていないが，これはデッドロックを引き起こすためである．MPI/sock では性能比較に向けてオーバヘッドを小さくするため，デッドロックを回避するコードを省略してある．

6. 議　　論

6.1 キュー管理

現在，O2G の管理する受信メッセージ・キューおよび受信リクエスト・キューは単純な線形リストであり，タグや送り元によるハッシュ等は一切行なっていない．この部分は計算機規模に応じて改良する必要がある．ただし，一般的な MPI アプリケーションでは，受信リクエスト・キューの長さは短いと考えられる．2 つのキューのうち受信メッセージ・キューに関しては，O2G の扱いは線形リストのままでユーザーレベルでキューにハッシュ等を追加するような実装も考えられる．

6.2 メッセージ・ポーリング

MPI では「処理の進行」という点が議論される．select&read を用いる実装では通信処理を進行させるた

め，アプリケーションに対して MPI 関数の呼び出しを適宜挿入するコーディグが要求される場合がある．一方，O2G では受信はすべてプロトコル処理ハンドラで実行されるので必要以外の MPI 関数呼び出しを挿入する必要はない．

6.3 メッセージ送信

現在，O2G はメッセージの受信のみに対処している．一方，送信側には汎用の非同期 I/O が利用できると考えている．受信側では受信リクエスト・キューや受信メッセージ・キューの操作といった処理が必要であり，汎用の非同期 I/O は利用できない．そのため専用のドライバを作成する必要があった．一方，送信側は単に write するだけでよく，特別な処理を行なう必要はない．

7. 関 連 研 究

7.1 select の効率化

kqueue[6] は一部 BSD 系の Unix で実装されている機能である．eventlist という形で通知すべきイベントをフィルターする．イベントのあるソケットのみにサーチが制限されるのでイベント検出が高速になる．

devpoll は Solaris で提供されている機能である．これも同様であり，/dev/poll というデバイスを使って指定したソケット群にサーチを限定し高速化する．

7.2 非同期 I/O

aio_read/aio_write は POSIX のリアルタイム拡張に含まれており，一部の Linux や Unix で使用可能である．**aioread/aiowrite** は Solaris (Unix SysV) に含まれている．どちらも非同期 I/O の機能を提供しており，ほぼ同様である．

非同期 I/O では，システムコール後，制御がすぐにユーザープロセスに返って来るので，ユーザープロセスは I/O 処理と並行して処理を継続することができる．これは主に大量データの I/O を想定しており，データ I/O をバックグラウンドで処理するために使用される．

7.3 リモート書き込み

RDMAP[7] は TCP/IP 上で使用されるリモート書き込みのプロトコル標準である．非同期 I/O 同様，通信性能向上のために設計されている．

MPI/MBCF[8] はユーザーレベル通信に基づく MPI の実装である．リモート書き込み以外に FIFO 受信バッファを利用しており，この FIFO を使う場合の通信は O2G の実装に近い．

7.4 O2G の優位性

MPI の実装に非同期 I/O を用いてもあまりメリットはない．MPI では，ヘッダーの受信，続いてメッセージ本体の受信と連続して read を発行し続ける必要があるうえ，非同期 I/O の終了に select と同様な複数イベント待ちが必要である．このため受信側の処理に関しては，基本的に select&read と同数のシステムコールの発行が必要であり，根本的な解決にはならない．

select では read が可能になってから read を発行し read の完了を待つ．非同期 I/O では，aio_read を発行し aio_read が終了するのを select で待つことになる．大量データの I/O であれば，バックグラウンドで read 処理が進行するのでメリットがあるが，細かい単位での処理ではメリットは小さい．

8. お わ り に

大規模クラスタ計算機やグリッドに向けた MPI を実装するための通信最適化の機構として O2G ドライバの設計・実装を行なった．O2G は通信レイヤは変更することなく，オーバーヘッドが大きいと考えられるシステムコール API を変更することで非同期処理を効率化することを狙っている．そのため，MPI で必要になる受信キュー操作をプロトコル処理ハンドラ内で実装した．

評価には LAN で接続された小規模クラスタを用いており，規模や遅延の点で対象とするシステムにはなっていない．しかし，非同期通信が重要となる IS ベンチマークでは高い性能を，そして他のベンチマークでも従来の実装と互角の性能を示した．これは，無駄なオーバーヘッドがないことを示している．

今回は MPI のサブセットによる評価を行なったが，今後は MPI フルセット実装に対して O2G を適用する予定である．そして，大規模クラスタや遅延の大きなネットワークを使った評価を行ないたいと考えている．

参 考 文 献

1) Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, May 5, 1994.* University of Tennessee, Knoxville, Report CS-94-230, 1994.
2) The GridMPI Home Page. http://www.gridmpi.org/
3) NAS Parallel Benchmarks. http://science.nas.nasa.gov/Software/NPB
4) W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. Parallel Computing, Vol.22, No.6, pp.789–828, 1996.
5) M. Matsuda, T. Kudoh, and Y. Ishikawa. Evaluation of MPI Implementations on Grid-connected Clusters using an Emulated WAN Environment. CCGrid'03, pp.10–17, 2003.
6) J. Lemon. Kqueue: A generic and scalable event notification facility. BSDCon 2000, pp.141–154, 2000.
7) RDMA Consortium. http://www.rdmaconsortium.org/
8) 森本，松本，平木. メモリベース通信を用いた高速 MPI の実装と評価. 情報処理学会論文誌, Vol.40, No 5, pp.2256–2268, 1999.