MPI通信モデルに適した非同期通信機構の設計と実装

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に向けたMPIを実装するための通信機構であるO2Gドライバの設計・実装を行っている．O2Gでは，TCP/IPプロトコル通信レイヤ自体は変更せず，MPIの実装に必要となる受信キュー操作をプロトコル処理ハンドラに組み込んでいる．割込みで起動されるプロトコル処理ハンドラ内で，TCP受信バッファから受信データを読み出しユーザ空間にコピーする．これによって，TCP受信バッファの溢れにともなう通信フローの停滞が抑制され，通信性能を劣化させることがなくなる．さらに，従来のソケットAPIで必要だったポーリングが不要になり，システムコール・オーバヘッドが低減される．NAS並列ベンチマークのISベンチマークでは，O2Gを使用することで従来のMPI実装に比べて3倍の性能が得られる．さらに，ソケットによるMPI実装ではコネクション数が増大すると通信バンド幅が低下するが，O2Gではコネクション数に関係なく高性能なデータ受信を達成していることが示される．

The Design and Implementation of Asynchronous Communication Mechanism for MPI Communication Model

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

In order to implement an efficient MPI communication library for large-scale commodity-based clusters, a new communication mechanism, called O2G, is designed and implemented. O2G introduces receive queue management of MPI into a TCP/IP protocol handler without modifying the protocol stacks. Received data is extracted from the TCP receive buffer and copied into the user space within the TCP/IP protocol handler invoked by interrupts. This avoids message flow disruption due to the shortage of the receive buffer and keeps the bandwidth high. In addition, it totally avoids polling of sockets and reduces system call overheads. An evaluation using the NAS Parallel Benchmark IS shows that an MPI implementation with O2G performed three times faster than other MPI implementations. An evaluation on bandwidth also shows that an MPI implementation with O2G was not affected by the number of connections while an MPI implementation with sockets was affected.

1. はじめに

Ethernetがコモディティ化するとともにギガビットへと性能向上し，1000台規模の大規模クラスタがLinux，Ethernet，TCP/IPの組合せで構築されるようになってきた．また，グリッド環境上のクラスタや並列コンピュータを用いた並列アプリケーション実行環境構築の試みも行われている[14]．このような環境上の並列アプリケーションでは，トランスポート層にTCPを使った並列計算用の通信ライブラリであるMPI[11]が使用される．TCP以外の独自ネットワークプロトコルが使用されないのは，そのようなネットワークプロトコルが実装されたシステムソフトウェアがすべての実行環境上で使用できなければならないという強い制約があるためである．

TCPによる通信はコネクション指向のストリームをベースにしている．Linux，Unix，Windowsといった OSでは，通信APIは通信エンドポイントである「ソケット」として提供される．元々ソケットはプロセスあたりのソケット使用数が小さい場合を想定して設計されてきた．このため，多数のソケットを使用する場合に性能が低下する問題が指摘されている[2]．大規模クラスタではノード数に応じて多数のコネクションが作成されるため，ソケットが多数必要になり性能

† 産業技術総合研究所グリッド研究センター
Grid Technology Research Center, National Institute of Advanced Industrial Science and Technology
†† 東京大学大学院情報理工学系研究科
Graduate School of Information Science and Technology, The University of Tokyo

が低下する．

複数ソケットからの受信処理は，select と read の 2 つのシステムコール列の繰返しによって構成される．処理は，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 を使ってデータを読み出すという操作を繰り返す．このようなポーリングをベースにする実装は，システムコールの回数が多くなる問題がある．また，通信データを読み出すタイミングが遅れるため通信フローが停滞し通信性能が低下する．

これは OS とユーザプロセス間の API の問題である．OS 内部では通信処理は割込みによって非同期に行われており，ポーリングのようなオーバヘッドはない．通信処理自体も，コネクション数の増加に対して通信処理コストが一定になるように実装されている．しかし，ユーザプロセスは逐次的にモデル化されており，非同期に発生する通信イベントの処理効率が悪い．

この問題を解決するため，ユーザレベル通信[5]，イベント検出の高速化[4),9),18]，非同期 I/O[15] などが考案されてきた．U-Net のようなユーザレベル通信は NIC を直接アクセスすることでオーバヘッドを削減する．しかし，デバイスドライバを含め通信レイヤ全体を置き換える必要があり，複数ベンダのハードウェアや OS が使用される環境をサポートすることは容易ではなく，コモディティ化したクラスタ計算機にはそぐわない．イベント検出の高速化や非同期 I/O は汎用性の高い API を目指すもので，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変わりなく，高い効率は期待できない．

そこで我々は，大規模クラスタにおいても高性能な通信を提供する MPI を実装するため，MPI 専用 API を実装する「O2G ドライバ」の設計・実装を行った．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．O2G では TCP 通信レイヤ自体はそのまま使用し，オーバヘッドが大きいソケット API をバイパスする．そして，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処理する．O2G は Linux のローダブル・ドライバとして実装されており，実装は非常に単純なものになっている．

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

2. ソケットの問題点

2.1 MPI 通信モデル

MPI[11] 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キング操作である．ノンブロッキング通信には MPI_Isend と MPI_Irecv を用いる．ブロッキング操作では，処理が終了するまで呼び出しから戻らない．ノンブロッキング操作では処理の終了を待たずに呼び出しから戻り，処理終了の判定には MPI_Wait などの関数を利用する．MPI では，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．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

MPI の実装では，常に通信ストリームからデータを読み出し続けることが要求される．理由の 1 つは，MPI の通信モデルによるものである．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．そのため，順次メッセージを読み出し続ける必要がある．理由のもう 1 つは，TCP による通信性能の低下を避けるためである．TCP では受信バッファサイズをウィンドウサイズとして伝達するフロー制御を行っている．メッセージが読み出されずに受信バッファにたまると，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ことになる．

MPI 通信ライブラリの一般的な実装[3),7] では，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を使用し，ソケットのインタフェースである read/write システムコールが利用される．また，ソケットにはイベント待ちやポーリングを行うために 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サポートされていないので，ストリームからデータを読み出し続けるにはポーリングを行う必要がある．ポーリングは，select とノンブロッキングな read を繰り返すコード列を使って実現される（ノンブロッキングな read は，ファイル・ディスクリプタをノンブロッキングに設定して使用する）．

MPI の実装ターゲットとしては，大規模クラスタ

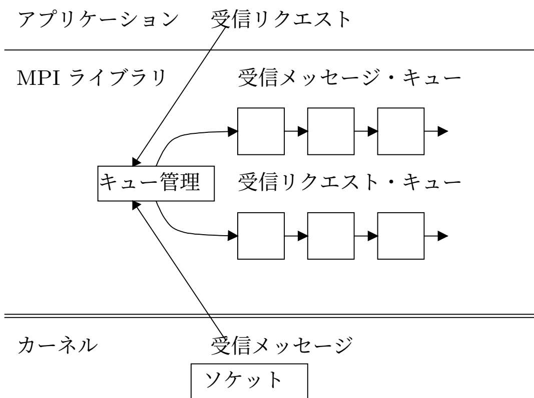

図 1 MPI の実装で用いられる受信キュー
Fig. 1 Receive queues found in typical MPI implementations.

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[10]．このような環境において，ソケット API には大きく 3 つの問題がある．

第 1 に，select と read の繰返しループによるポーリング・ベースの実装では，データの読み出しがポーリング時点に限られるので，つねにストリームからデータを読み出し続けることができない．そのため，通信フローの停滞が発生する．

第 2 に，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くなる．ノンブロッキングな read では，実際に読み出した量が read で指定したデータ量に満たなくてもシステムコールから戻ってくる．1 回の読み出し量が小さくなるので，システムコールが高頻度になる．一般に OS のシステムコールはオーバヘッドの大きい処理であり性能に影響を与える．

第 3 に，これは OS の実装上の問題ともいえるが，Linux や Unix における select 処理はコネクション数の増加に対して性能が低下する．ポーリングを行う select 処理の実行で，プロセスが使用するソケット数に比例する処理が行われるためである．

3. MPI 実装の基本動作

3.1 受信キュー

MPICH[7] や LAM/MPI[3] といった MPI の標準的な実装では，受信処理に基本的に 2 つのキューを利用する（**図 1**）．

- 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 ライブラリは，メッセージを受信した場合，このキューにエントリを挿入する．

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．MPI ライブラリは，MPI_Recv/MPI_Irecv により受信リクエストが発行されると，このキューにエントリが作られる．

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．このときまず，受信メッセージ・キューを調べて，すでに受信したメッセージの中にリクエストにマッチするエントリが存在するかどうかをチェックする．もしマッチするエントリが存在すれば，そのエントリに対応するメッセージによって受信リクエストが完了する．

一方，マッチするエントリがない場合，受信リクエスト・キューにリクエストを挿入する．受信リクエスト・キューに挿入されたリクエストは，その後，マッチするメッセージが受信された時点で削除される．

3.3 メッセージ受信動作

MPI のランタイム・ライブラリがメッセージを受信したときは，まず受信リクエスト・キューを調べ，すでに発行されている受信リクエスト中にマッチするエントリが存在するかどうかをチェックする．もしマッチするエントリが存在すれば，そのエントリに対応する受信リクエストに対してメッセージ・データを書き込み受信リクエストを完了する．

一方，マッチするエントリが存在しない場合，受信したメッセージを受信メッセージ・キューに挿入する．受信メッセージ・キューに挿入されたメッセージは，その後，マッチする受信リクエストが発行された時点で削除される．

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 の目的は，届いたメッセージをただちに通信レイヤから読み出し受信側の処理を最適化する．その

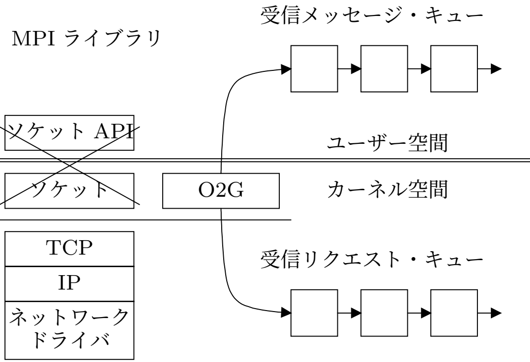

図 2　O2G 動作概要
Fig. 2　Overview of operation of O2G.

ため O2G では，3 章「MPI 実装の基本動作」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．O2G は Linux のローダブル・ドライバとして実装されており，Linux カーネルに対するパッチ等は不要である．利用にあたって OS の再構築や再起動は必要ない．

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，ユーザ領域内にバッファをとっている．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．

受信したメッセージは，通常のカーネル内の受信処理とほぼ同様に処理される．通常のソケットによる受信処理の場合，受信データはまず Linux のプロトコル処理バッファである SKB にコピーされる．次に，SKB はソケットごとにある受信バッファに挿入され，ユーザプロセスが read を行うまで保持される．一方，O2G の場合，受信バッファに挿入された SKB はただちに読み出され MPI の受信処理が行われる．受信メッセージに対して，MPI 受信リクエスト・キューへのマッチング，あるいは MPI 受信メッセージ・キューへの挿入が遅延なしに行われる．

O2G の実装に必要な処理として，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，ソケットに対する処理に比べて大きなオーバヘッドとはならない．

4.2　ドライバ関数

O2G はデバイス・ドライバであり，ユーザプロセスからは ioctl システムコールを通じて制御を行う．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*/
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 API
Fig. 3　User API of O2G.

初期化関数群では，`o2g_init` は初期化処理を行う．`o2g_register_socket` はソケットと相手プロセスのプロセス番号（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` はユーザプロセス空間にある受信メッセージ・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に用いられる．

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 サーバを実現している．そのため，SUN RPC[17] 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・データとして受信バッファに入れるが，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 `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 が処理されるごとに呼び出される．ここで `tcp_read_sock` は処理を簡潔に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あげている．

Linux ではこのようにカーネル内部で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．

```
{
  /*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
Fig. 4　Hook function in the socket layer and its sample usage.

4.4　コンテクスト不一致時の処理

O2G ではすべての受信処理が，割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，割込み時点のプロセス・コンテクストがユーザ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領域へのデータ書き込みが行えない．この場合，O2G はユーザ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，ユーザスレッドを起動して書き込みを行う．

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 の場合も起こりうるものである．ソケットの場合は，read でブロックしているユーザプロセスが起動されることになるが，処理は同等であり，O2G にだけに必要となる特別なオーバヘッドではない．

4.5　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，ユーザプロセスから呼び出されるキュー・エントリ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．

表 1　PC クラスタ仕様
Table 1　PC cluster specification.

ノード PC	
CPU	Pentium4 2.4C （2.4 GHz）
マザーボード	Intel D865GLC
メモリ	512 MB DDR400
NIC	Intel 82547EI（オンボード CSA 接続）
OS	RedHat 9（Linux-2.4.20）
NIC ドライバ	Intel e1000 5.2.16
ネットワークスイッチ	
	Dell Powerconnect 5224

5. 性能評価

5.1　ベンチマーク環境

O2G による MPI 実装の効果を評価するため，ソケット API を使用する MPI 実装とのベンチマーク比較を行った．主な比較対象は，我々が開発した TCP 上の MPI-1.2 実装である YAMPII [19] である．YAMPII は TCP 上のソケットを使うポータブルな MPI の実装である．O2G を用いる MPI も YAMPII をベースに実装しており，コードの大部分は YAMPII と共通である．2 つの実装の違いは，受信処理にかかわる受信メッセージと受信リクエストのそれぞれのキュー操作部分だけである．O2G での実装ではカーネルドライバとして組み込めるように YAMPII のキュー操作を変更している．以後の評価では，ソケット版の YAMPII/Sock に対して O2G 版を YAMPII/O2G と呼び区別する．

評価には，16 台の Pentium 4 PC からなる小規模なクラスタを用いた．**表 1** にクラスタに使用した PC およびネットワークスイッチの主な仕様を示す．使用した PC は NIC が CSA Bus [8] 接続であり，Linux 上の通信ベンチマークプログラム（iperf）で Gigabit Ethernet のほぼ限界である 941 Mbps の性能が出ることを確認している．PC のプロセッサは SMT（Simultaneous Multi-Threading）機能を持つが，実験では使用せず，OS もシングル CPU 版の Linux カーネルを用いた．コンパイラは C 言語も Fortran も GCC（バージョン 3.2.2）を使用し，すべてのコードは最適化オプション-O3 でコンパイルした．

5.2　NPB ベンチマーク

ソケットと O2G の比較として，MPI の一般的なベンチマークである NPB（NAS Parallel Benchmarks [1]）（バージョン 2.3）を使用して性能評価を行った．

ソケットによる YAMPII/Sock と O2G による YAMPII/O2G の比較が主目的であるが，ベースとなる YAMPII の基本性能が低くては比較に意味がな

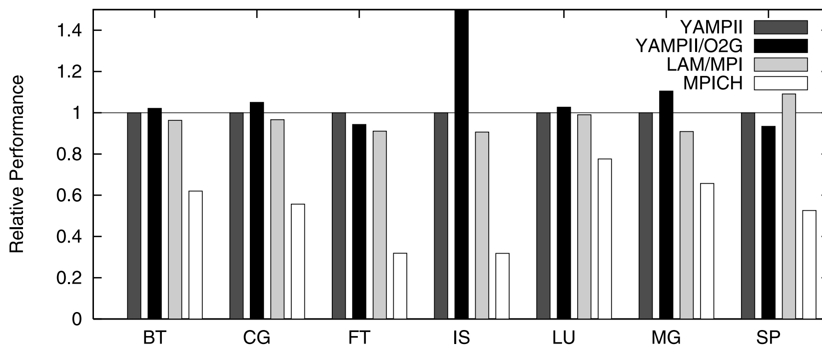

図 5 NPB ベンチマーク比較（相対値）
Fig. 5 Relative performance of the NPB benchmarks.

い．そこで NPB のベンチマークでは，代表的な MPI 実装である MPICH [7] と LAM/MPI [3] を比較対象に加えた．それぞれのバージョンは，評価時点で最新の MPICH は 1.2.5.2，LAM は 7.0.4 を使用した．

NPB の評価では，ネットワーク・パラメータ等は Linux-2.4.20 の既定値のままである．ただし，TCP のパラメータのうち TCP_NODELAY だけはすべての実装が設定している．TCP_NODELAY はパケット送信を遅延する Nagle アルゴリズムを無効にするものである．

図 5 に YAMPII/Sock の性能を 1 とした相対性能を示す．NPB ベンチマークのデータサイズはクラス A を使用した．性能はベンチマークの表示のうち「Mop/s total」値の比較である．**表 2** にはベンチマークの絶対値を表示した．MPICH の性能が全体的に低いが，それ以外の YAMPII/Sock，YAMPII/O2G，LAM はほぼ同等の性能を示している．これはベースとしての YAMPII が十分実用レベルであることを示している．特に，YAMPII/Sock と LAM の比較では SP ベンチマーク以外で YAMPII/Sock の性能が高い．

IS を除くベンチマークではほとんど差がみられない．ベンチマーク・プログラムを調べると，通信のほとんどが 1 対 1 通信であることが分かる．1 対 1 通信では複数ストリームから同時に受信する非同期通信を行う必要がない．また，ノード数が 16 では select のオーバヘッドも影響が出るほど大きくならない．

IS ベンチマークにおいては，YAMPII/O2G が他の実装の 3 倍以上の性能を示していることが目立つ．IS は比較的小さな 100 KB 程度のメッセージサイズによる全対全通信を行っており，ソケットを用いる通信では通信の停滞や通信オーバヘッドの影響が予想される．O2G の効果が出やすいベンチマークであると考えられる．

YAMPII/Sock と YAMPII/O2G の比較では，差は小さいが FT と SP の 2 つのベンチマークで YAMPII/O2G の性能が低くなっている．原因として，まず，O2G の実装は未熟であり，コードが最適化されているソケットレイヤに比べ単純に性能が低いことが考えられる．また，6 章で述べるが，O2G の利用には性能上のトレードオフが存在する．YAMPII/Sock では，受信メッセージは read するまでカーネル内の TCP バッファに置かれており，受信処理はポーリングの時点まで遅らされる．一方，YAMPII/O2G では，メッセージの到着と同時に受信処理が行われるため受信処理のタイミングが早い．このため，受信処理がアプリケーションでの MPI_Recv 発行に先行する可能性が高くなり，ユーザプロセス内でのバッファリングに必要なコピーの確率が高くなる．メッセージのコピーは性能低下の原因となる．

表 2 NPB ベンチマーク比較（'Mop/s total' 値）
Table 2 Performance of the NPB benchmarks.

	YAMPII /O2G	YAMPII /Sock	LAM	MPICH
BT	5493.2	5380.9	5182.4	3334.1
CG	854.8	814.5	787.0	453.3
EP	80.6	80.5	80.7	80.2
FT	2855.1	3026.3	2758.4	965.4
IS	181.5	58.9	53.3	18.7
LU	5904.0	5751.6	5696.5	4463.9
MG	4892.8	4426.3	4025.5	2908.8
SP	2629.6	2816.8	3072.9	1481.3

5.3 バンド幅ベンチマーク

O2G の開発目的はスケーラビリティの向上である．1,000 台規模のクラスタに向けた通信性能をみるため，基本的な性能であるバンド幅の計測を行った．

ここでは，コネクション数（ソケット数）による性能変化の違いをみる．MPI ではすべてのノードからの通信をつねに受信するので，すべてのコネクションに対してポーリングする必要がある．そこで，コネクション数に対して性能が変化することが予想される．

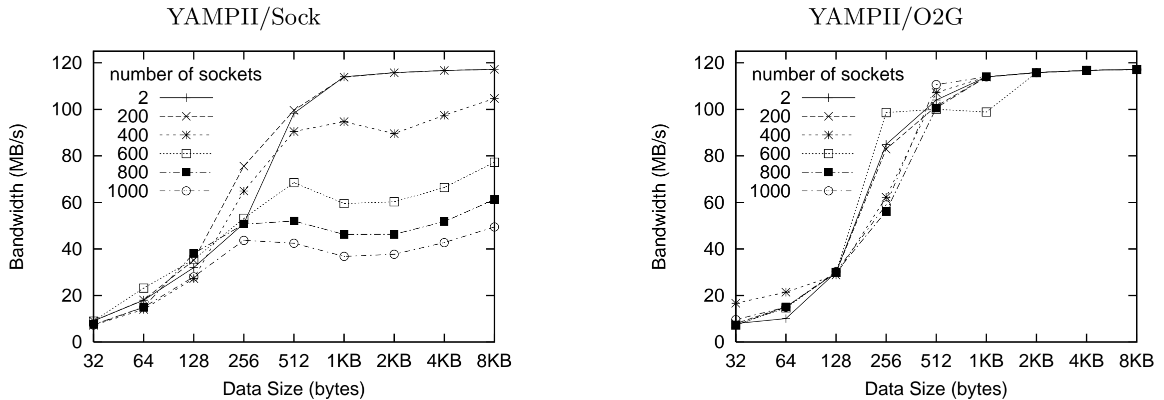

図 6 コネクション数によるバンド幅の変化
Fig. 6 Bandwidth degradation due to the number of connections.

この実験では，2 ノード間でバンド幅を計測するが，第 3 のノードがクラスタ 1,000 台構成を模擬する．第 3 のノードはバンド幅計測を行う 2 ノードへ残り 998 台分のコネクションを作成する．第 3 のノードはコネクション作成するだけであり，実際の通信はまったく行わない．

このベンチマークでは，TCP のバッファサイズを指定しない場合 YAMPII/Sock でバンド幅の性能が大きくゆらぎ，意味のあるデータが得られなかった．そこで測定では TCP の受信/送信バッファサイズを指定した．受信/送信バッファサイズはソケットあたり 128 KB に設定した．YAMPII/O2G ではバッファサイズを設定しなくても揺らぎは小さく性能も高かった．しかし評価の条件は同じにしている．

図 6 にコネクション数を変化させた場合のバンド幅の変化を示す．YAMPII/Sock と YAMPII/O2G の性能変化をそれぞれ別のグラフに示した．YAMPII/Sock では，コネクションの数が増大するに従いバンド幅が低下していくことが分かる．一方，YAMPII/O2G は，コネクションの数に関係なく一定のバンド幅が得られている．両者は同一の TCP/IP プロトコルスタックを使用しているが，YAMPII/Sock は select と read ループによる実装用いている．この結果から，Linux カーネル自体はコネクション数に影響されない設計がなされているのに対して，ユーザ API であるソケットがコネクション数の影響を受けていると判断できる．

5.4 select システムコールのオーバヘッド

バンド幅の結果から，ソケット API がコネクション数の影響を受けていることが分かった．そこで，次にその基本的な計測データとして，select システムコールにかかる時間を計測した．

バンド幅計測と同じ環境設定で，ソケット数を変化させた場合の select システムコールにかかる時間を測定した．測定には CPU のクロックカウンタを使用し，連続して 10 回測定しその最小値を採用した．すべてのソケットに受信データがない状態である．

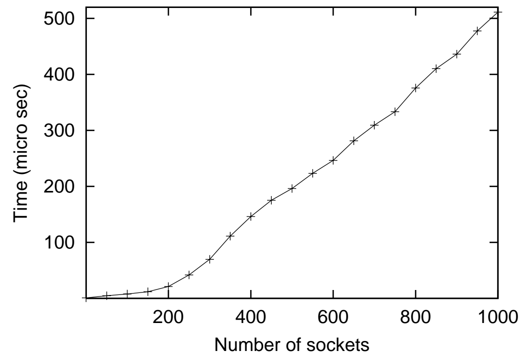

図 7 ソケット数と select システムコールの時間
Fig. 7 System call overhead due to the number of connections.

図 7 に結果を示す．ソケット 1 つあたりでは 0.5 μsec と処理時間は小さい．ただし，すべてのファイル・ディスクリプタに対して行うのでコネクション数が増えるとほぼ線形に時間がかかることになる．

select（poll システムコールも同様）の処理は，まず，ファイル・ディスクリプタ番号からファイル・ディスクリプタ構造体を参照．次にそこからソケット構造体，次に TCP 実装の構造体を順に参照する．ソケット構造体は参照に際して排他される．実際のポーリングは TCP 実装の構造体のフィールドを参照し，フラグとシーケンス番号を調べ受信できるデータが存在するかチェックするだけでありそれ自体は簡単な処理である．

この実験結果から，1,000 ノード規模のクラスタで

は一度のポーリングに 0.5 ミリ秒程度の時間がかかることになる．MPI レベルでの 1 回の受信処理には，通常，ヘッダの読み込みとデータ本体の読み込みのために最低 2 回のポーリングが必要である．このポーリングに必要な 1 ミリ秒は純粋なオーバヘッドであり，1 Gbps の通信では 1 ミリ秒に 100 KB 以上のデータが通信できることを考えるとオーバヘッドとしてはとても大きい．

6. 議　　論

6.1 O2G の性能トレードオフ

受信したメッセージが MPI の受信メッセージ･キューに挿入された場合，それに対応する受信リクエストがまだ発行されていないことを意味する．この場合，メッセージデータはいったんバッファにコピーされるので，メッセージを受信するには 2 度のコピー処理が必要になる．

O2G ではメッセージがただちにユーザ空間に読み出されるが，これには悪い点もある．早い時点でメッセージを処理するため，受信リクエストが発行される前であり受信メッセージ・キューに挿入される可能性が高い．つまり，2 度コピーが行われる確率が高くなる．一方，select と read による実装では，アプリケーションがポーリングしない限り read を行わないので，2 度コピーを行う確率が低くなる．

O2G では，プロセッサのメモリ・バンド幅が向上していることを考慮し，通信フローの停滞による性能低下の方を重視したトレードオフを選択しているといえる．

6.2 メッセージ・ポーリング

MPI では「処理の進行」[11] という点が議論される．これは，ポーリングを用いる MPI 実装を使用する場合，アプリケーション・プログラム中に MPI 通信関数の呼び出しを適宜挿入する必要があるという要請である．ポーリングを用いる実装では，アプリケーションが MPI のいずれかの関数を呼び出さない限りポーリングが行われない．最悪の場合，通信がデッドロックする可能性がある．そのため，通信処理を進行させるためにアプリケーション・プログラム中に MPI 通信関数の呼び出しを適宜挿入する必要がある．一方，O2G では受信はすべて割込みで実行されるので必要以外の MPI 通信関数の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 [15] 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，特別な処理を行う必要はないためである．

7. 関 連 研 究

7.1 select の効率化

devpoll [18] は Solaris で提供されている機能である．これは，/dev/poll というデバイスを使い，サーチを行うソケット群を指定する．select システムコールの制限を除くために導入された poll システムコールでは，select に比べて引数のサイズが大きくなった．システムコールでは，ユーザ/カーネル空間の間で引数をコピーする必要があるが，poll で大きくなった引数のコピーが問題になった．そのコピーを省略するのが devpoll 導入の主目的である．

新たに Linux に導入された epoll も，インタフェースは異なるが devpoll と同様の機能を提供するものである．

kqueue [9] 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．

他に文献 4) では，リアルタイムのシグナル・イベントを非同期イベントの通知に利用する方法が報告されている．

7.2 非同期 I/O

非同期 I/O とは，システムコール後，制御がすぐにユーザプロセスに返ってくる処理をさす．これによって，ユーザプロセスは 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 を行う場合を想定しており，I/O をバックグラウンドで処理するために使用される．

aio_read/aio_write は POSIX のリアルタイム拡張[15] 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 に含まれている．どちらもノンブロッキングな I/O システムコール呼び出しを提供しており，ほぼ同様である．

7.3 リモート書き込み

RDMAP [16] 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 [12] はユーザ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．ただし，MPI/MBCF は独自プロトコルと独自の実装を使っており，TCP のスタックをそのまま利用している O2G とは異なる．

7.4 ドライバでの MPI 実装

Myricom 社の MX[13] は，Myrinet のドライバで MPI のモデルに近い通信処理を実現するものである．これを使う MPI の実装は O2G に近いものになると考えられる．

7.5 O2G の優位性

select の効率化は引数コピーの問題を軽減する手段として導入された．O2G はポーリングを必要としないので，この問題は起こらない．また，select の効率化は処理を少数の受信ポートに限定することで効率化しているが，MPI のように一時に多くの受信を行う条件で適用できるかは明らかではない．O2G のドライバによる実装では TCP を直接参照するので，効率化のためにポートを限定する必要はない．

非同期 I/O は MPI の実装に用いてもあまりメリットはない．非同期 I/O の完了には select と同様の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 と read と同様のシステムコールの発行が必要になる．MPI では，ヘッダの受信，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が，ヘッダを解析するまでメッセージ本体の受信は発行できない．ヘッダを解析せずに受信するには，ユーザプロセス内でバッファリングが必要になりメッセージの余分なコピーが発生する．このため，処理が行われるのがアプリケーションから MPI の通信が呼び出される時点，つまりポーリングと同じ動作になり，根本的な解決にはならない．

8. おわりに

大規模クラスタ計算機に向けた 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・実装を行った．O2G は通信レイヤを変更することなく，オーバヘッドが大きいと考えられるシステムコール API を変更することで処理を効率化することを狙った．そのため，MPI で必要になる受信キュー操作をカーネル内のプロトコル処理ハンドラ内で実行する設計をとった．

O2G の並列ベンチマークに対する評価は，LAN で接続された小規模クラスタを用いており規模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 ベンチマークで高い性能が出ることが確認できた．また，ノード数に対するスケーラビリティを調べるため，疑似的に 1000 ノードのコネクションを張った環境を作りバンド幅測定を行った．実験の結果，ソケットではコネクション数に影響を受けるが，O2G ではコネクション数にほとんど影響されないことが確認できた．

CPU や NIC の性能向上により性能バランスが変化し，Ethernet と TCP/IP の組合せを使った高性能計算が可能になってきた．しかし，OS のプログラム・モデルや API がその性能に追従できていない．問題点は指摘されて続けているが，汎用の解決は提供されていない．そこで，O2G ではクラスタ利用で重要な MPI に特化した形で，問題を避けることができることを示した．

参 考 文 献

1) Bailey, D., Harris, T., Saphir, W., van der Wijngaart, R., Woo, A. and Yarrow M.: The NAS Parallel Benchmarks 2.0, *Intl. Journal of Supercomputer Applications* (1995). http://www.nas.nasa.gov/Software/NPB
2) Banga, G. and Mogul, J.C.: Scalable Kernel Performance for Internet Servers under Realistic Loads, *Proc. 1998 USENIX Annual Technical Conference* (1998).
3) Burns, G., Daoud, R. and Vaigl, J.: LAM: An Open Cluster Environment for MPI, *Proc. Supercomputing Symposium*, pp.379–386 (1994). http://www.lam-mpi.org
4) Chandra, A. and Mosberger, D.: *Scalability of Linux Event-Dispatch Mechanisms*, Hewlett Packard Laboratory, HPL-2000-174 (2000).
5) von Eicken, T., Basu, A., Buch, V. and Vogels, W.: U-Net: A User-Level Network Interface for Parallel and Distributed Computing, *Proc. 15th ACM Symposium on Operating Systems Principles* (1995).
6) The GridMPI Home Page. http://www.gridmpi.org
7) Gropp, W., Lusk, E., Doss, N. and Skjellum, A.: A High-performance, Portable Implementation of the MPI Message Passing Interface Standard, *Parallel Computing*, Vol.22, No.6, pp.789–828 (1996).
8) Intel, Corp.: Communication Streaming Architecture — Reducing the Bottleneck for PCI Networking (Brochure).
9) Lemon, J.: Kqueue: A Generic and Scalable Event Notification Facility, *BSDCon 2000*, pp.141–154 (2000).
10) Matsuda, M., Kudoh, T. and Ishikawa, Y.: Evaluation of MPI Implementations on Grid-connected Clusters using an Emulated WAN Environment, *Proc. 3rd Intl. Sympo-*

sium on Cluster Computing and the Grid (*CC-Grid2003*), pp.10–17 (2003).

11) Message Passing Interface Forum: *MPI: A Message-Passing Interface Standard, May 5, 1994*, University of Tennessee, Knoxville, Report CS-94-230 (1994).
12) 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).
13) Myricom, Inc: Myrinet Express (MX): A High-Performance, Low-Level, Message-Passing Interface for Myrinet, MX API Reference (2003).
http://www.myri.com/scs/MX/doc/mx.pdf
14) 超高速コンピュータ網形成プロジェクト NAREGI. http://www.naregi.org
15) POSIX: POSIX 1003.1 (Realtime Extensions), IEEE POSIX Std. 1003.1-2001 (2001).
16) RDMA Consortium.
http://www.rdmaconsortium.org
17) Srinivasan, R.: RPC: Remote Procedure Call Protocol Specification Version 2, RFC1831, Internet Engineering Task Force (1995).
18) Sun Microsystems: Solaris Manual Pages. poll (7d).
19) YAMPII, Yet Another MPI Implementation.
http://www.ilab.is.s.u-tokyo.ac.jp/yampii

(平成 16 年 1 月 31 日受付)
(平成 16 年 5 月 21 日採録)

松田 元彦（正会員）

1988 年京都大学理学部卒業．同年住友金属工業（株）入社．1995 年から 1999 年まで技術研究組合新情報処理開発機構に出向．2003 年より独立行政法人産業技術総合研究所．現在同研究所グリッド研究センター主任研究員．工学博士．並列計算システム，クラスタシステムおよびグリッド環境での高性能計算に関する研究に従事．

石川 裕（正会員）

1987 年慶応義塾大学大学院理工学研究科電気工学専攻博士課程修了．工学博士．同年電子技術総合研究所入所．1993 年技術研究組合新情報処理開発機構出向．2002 年より東京大学大学院情報理工学系研究科コンピュータ科学専攻助教授．クラスタ・グリッドシステムソフトウェア，高信頼システムソフトウェア開発技術，実時間分散システム，次世代高性能コンピュータシステム等に興味を持つ．

工藤 知宏（正会員）

1991 年慶應義塾大学大学院理工学研究科博士課程単位取得退学．東京工科大学助手，講師，助教授を経て，1997 年より新情報処理開発機構並列分散システムアーキテクチャつくば研究室長，2002 年より産業技術総合研究所グリッド研究センタークラスタ技術チーム長．博士（工学）．並列処理，通信アーキテクチャに関する研究に従事．電子情報通信学会，IEEE CS 各会員．

手塚 宏史

1957 年生．1985 年ソニー株式会社入社．UNIX ワークステーションの開発に従事．1989 年ソニーコンピュータサイエンス研究所勤務．1993 年北陸先端科学技術大学院大学研究生．1995 年新情報処理開発機構研究員．ワークステーション/PC クラスタの開発に従事．2001 年株式会社オムニサイソフトウェア入社．2003 年より産業技術総合研究所グリッド研究センター勤務．