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・実装を行なっている．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．割り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，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．また，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．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．小規模クラスタでも，NAS 並列ベンチマークのいくつかで O2G により性能が改善されることが示される．

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

MOTOHIKO MATSUDA ,[†] YUTAKA ISHIKAWA ,[††] TOMOHIRO KUDOH [†]
and HIROSHI TAZUKA[†]

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 on bandwidth shows that an MPI implementation with O2G was not affected by the number of connections, while an MPI implementation with sockets was affected. An evaluation using the NAS Parallel Benchmarks shows that an MPI implementation with O2G performed faster than other MPI implementations even on a small cluster.

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．並列計算用の通信ライブラリである MPI[11] の実装も，大規模クラスタに対応する必要にせまられている[6]．

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当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 とユーザープロセス間の API の問題である．OS 内部では通信処理は割り込みによって非同期に行なわれており，ポーリングのようなオーバヘッド

† 産業技術総合研究所 グリッド研究センター
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

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，ユーザー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，ユーザーレベル通信[5]，イベント検出の高速化[4),9),17)]，非同期 I/O[14)] などが考案されてきた．U-Net のようなユーザーレベル通信は 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，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・実装を行なった．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，オーバー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 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 節で並列ベンチマークとバンド幅の性能評価，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[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．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．理由の一つは，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．理由のもう一つは，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なっている．メッセージが読み出されずに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 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，ソケットのインターフェイスである 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なうために 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なう必要がある．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 には大きく三つの問題がある．

第一に，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．一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ーヘッドの大きい処理であり性能に影響を与える．

第二に，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，常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．

第三に，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．ポーリングを行なう 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なわれるためである．

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 つのキューが利用され

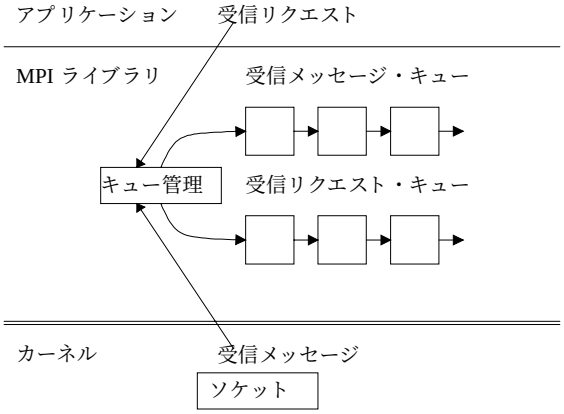

図 1 MPI の実装で用いられる受信キュー

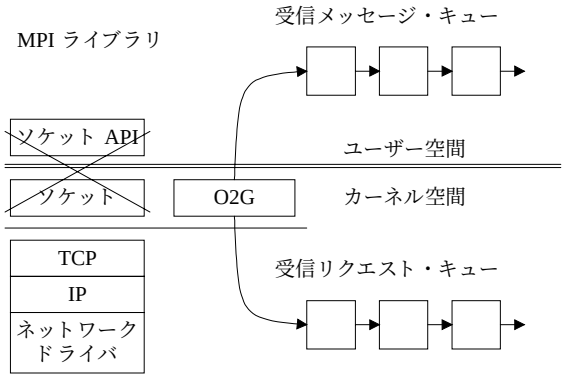

図 2 O2G 動作概要

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作る．

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を作る．

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時は，まず受信リクエスト・キューを調べ，既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すことである．そのため O2G では，3 節「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．ここで，O2G は Linux のローダブル・ドライバとして実装されており，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．

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 はソケット毎にある受信バッファに挿入され，ユーザープロセスが read を行なうまで保持される．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 の受信処理が行なわれる．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

```
/*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

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なわれる．

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ー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` はユーザー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[16] 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．

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，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．

4.5 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，ユーザー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

表 1 PC クラスタ仕様

ノード PC	
CPU	Pentium4 2.4C（2.4 GHz）
マザーボード	Intel D865GLC
メモリ	512MB 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

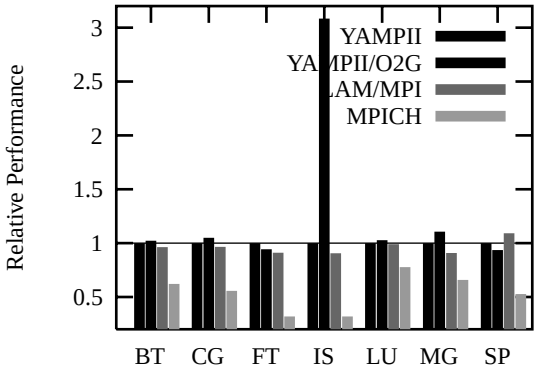

図 5 NPB ベンチマーク比較（相対値）

表 2 NPB ベンチマーク比較（'Mop/s total' 値）

	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

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

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なった．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 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．二つの実装の違いは，受信処理に関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 941Mbps 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なった．

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なう必要がない．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

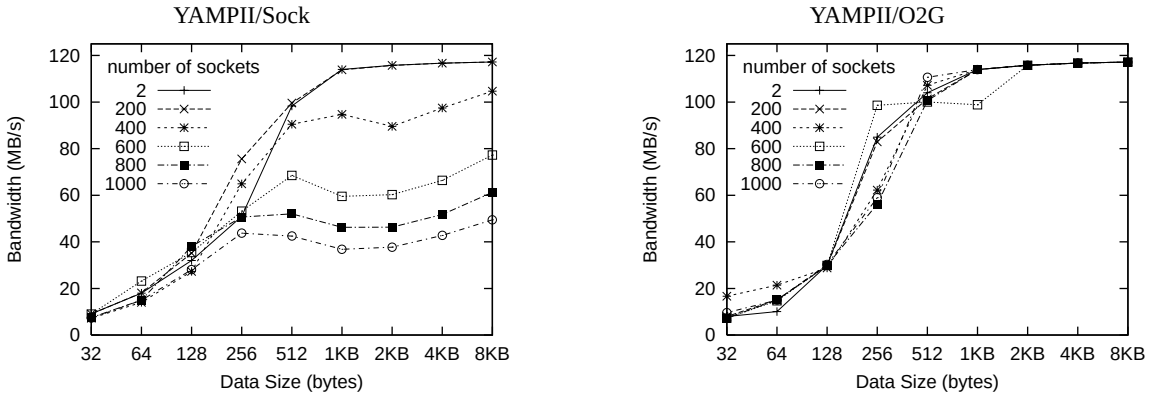

図 6 コネクション数によるバンド幅の変化

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 100KB 程度のメッセージサイズによる全対全通信を行なっており，O2G の効果が出易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，その差は小さいが 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．

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．1000 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なった．

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 では全てのノードからの通信を常に受信するので，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．

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 1000 台構成を模擬する．第 3 のノードはバンド幅計測を行なう 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，実際の通信は全く行なわない．

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当たり 128KB 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わかる．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，ユーザー 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．

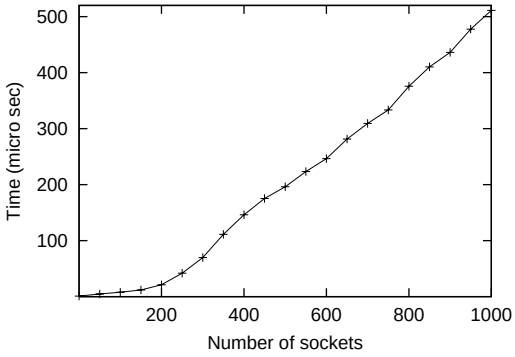

図 7 ソケット数と select システムコールの時間

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．すべてのソケットに受信データがない状態である．

図 7 に結果を示す．ソケット一つ当たりでは 0.5μsec 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な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．ポーリング処理

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．

この実験結果から，1000 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 レベルでの一回の受信処理には，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二回のポーリングが必要である．

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一旦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二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，二度コピーが行な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 を行なわないので，二度コピーを行なう確率が低くなる．

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なわれない．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り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[14] 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ためである．

7. 関 連 研 究

7.1 select の効率化

devpoll[17] 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，サーチを行な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，ユーザー／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，インターフェイスは異なるが 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．

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ープロセスに返ってくる処理をさす．これによって，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 を行な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[14] 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[15] 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 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．また，少数の受信イベントを効率化することを主目的としており，MPI のように一時に多くの受信を行なう条件に適用できるかは明らかではない．O2G はポーリングを必要としないので，これらの問題はない．

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 と read と同様のシステムコールの発行が必要である．また，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．ヘッダの解析が行なわれるのはアプリケーションから MPI の通信が呼び出される時点である．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・実装を行なった．O2G は通信レイヤ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採った．

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なった．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) D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow. The NAS Parallel Benchmarks 2.0. International Journal of Supercomputer Applications, 1995. http://www.nas.nasa.gov/Software/NPB
2) G. Banga and J. C. Mogul. Scalable Kernel Performance for Internet Servers under Realistic Loads. Proc. of the 1998 USENIX Annual Technical Conference, 1998
3) G. Burns, R. Daoud, and J. Vaigl. LAM: An Open Cluster Environment for MPI. Proc. of Supercomputing Symposium, pp.379–386, 1994. http://www.lam-mpi.org
4) A. Chandra and D. Mosberger. Scalability of Linux Event-Dispatch Mechanisms. Hewlett Packard Laboratory, HPL-2000-174, 2000.
5) T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. Proc. of the 15th ACM Symposium on Operating Systems Principles, 1995.
6) The GridMPI Home Page. http://www.gridmpi.org
7) 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.
8) Intel, Corp. Communication Streaming Architecture — Reducing the Bottleneck for PCI Networking. (Brochure) http://www.intel.com/design/network/events/idf/csa.htm
9) J. Lemon. Kqueue: A Generic and Scalable Event Notification Facility. BSDCon 2000, pp.141–154, 2000.
10) M. Matsuda, T. Kudoh, and Y. Ishikawa. Evaluation of MPI Implementations on Grid-connected Clusters using an Emulated WAN Environment. Proc. of The 3rd International Symposium on Cluster Computing and the Grid (CCGrid2003), 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) POSIX. POSIX 1003.1 (Realtime Extensions). IEEE POSIX Std. 1003.1-2001, 2001.
15) RDMA Consortium. http://www.rdmaconsortium.org
16) R. Srinivasan. newblock RPC: Remote Procedure Call Protocol Specification Version 2. RFC 1831, Internet Engineering Task Force, 1995.
17) Sun Microsystems. Solaris Manual Pages. poll(7d).