リモートメモリ書き込みを用いた MPI の効率的実装

建 部 修 見[†]　児 玉 祐 悦[†]
関 口 智 嗣[†]　山 口 喜 教[†]

MPI は point-to-point 通信において, 対応する送信と受信のマッチングにおけるコストが大きく, 通信遅延が大きくなる原因となっている. 本研究では, ノンブロッキング受信が先行発行される通信パターンにおいて, 送信時に受信側に問い合わせることなくリモートメモリ書き込みにより送信を行う方式を提案し, 高並列計算機 EM-X に実装しその評価を行った. その結果, 通信遅延 15.3 μsec., スループット 31.4 MB/s を達成し, 他 MPP に実装されている MPI に比べ優位な性能を示した. 本手法は, 他システムにおいても適応可能であり, ハードウェアスペック通りの低遅延, 高スループットを得る為には重要な方式と考えられる.

Efficient implementation of MPI using remote memory write

Osamu Tatebe,[†] Yuetsu Kodama,[†] Satoshi Sekiguchi[†]
and Yoshinori Yamaguchi[†]

MPI point-to-point communication is a basic operation, however it requires runtime-matching of send and receive that causes to reduce performance. This paper proposes a new approach to send messages by remote memory write without inquiring of the receiver under a communication pattern such that nonblocking receive is issued in advance. Basically, this approach makes it possible to gain low latency and high bandwidth as the hardware specification. MPI-EMX, our implementation of the MPI on the EM-X multiprocessor, achieves a zero-byte latency of 15.3 μsec. and a maximum bandwidth of 31.4 MB/s, which can compete with commercial MPPs. This approach to reduce communication latency is widely applicable to other systems and is quite a promising technique for achieving low latency and high bandwidth.

1. はじめに

MPI はポータブルかつ実用性を目指したメッセージパッシングインターフェースの標準として '94 に MPI-1[9], '97 に MPI-2[5] の仕様が制定され, 現在数多くのベンダ, 大学, 国研により大多数の MPP, 殆どの WS クラスタ上に実装され, 現実的にポータブルなものとなっている.

MPI はポータビリティを保ちつつ実行性能を極力落さないために, 実装方式を規定せず, また余分な処理を行わないようにデザインされている. また通信と計算のオーバラップを行い, 通信遅延を隠すことができるようにデザインされている.

Point-to-point 通信はメッセージパッシングライブラリにとって基本的な操作というだけではなく, 送信, 受信を行うことで局所的な同期をとることが出来るため, 並列プログラミングを行う上で有用な操作である. しかしながら, 現在の MPI の point-to-point 通信の実装では, 送信元プロセッサにワイルドカードを用いる受信が可能であるため, 対応する送受信のマッチング処理は一般に受信側で行われている. が, この場合送信時に一度受信側に問い合わせる必要があり, ハードウェアの性能に比べ通信遅延, 通信オーバヘッドが大きくなってしまう原因となる.

本研究ではその処理を送信側, 受信側の双方で行うことにより, ノンブロッキング受信が先行発行される通信パターンにおいて, 送信時に受信側に問い合わせることなく, ユーザ空間からユーザ空間へリモートメモリ書き込みにより送信を行う方式について提案と検討を行う. そして, 高並列計算機 EM-X 上にリモートメモリ書き込み, リモートスレッド起動, I-structure 等の通信機構を用い MPI の実装を行い, MPI ライブラリが効率的に処理をする余地のある MPI プログラムについての解説を行うと共に, 我々が実装した MPI-EMX の評価を行う.

† 電子技術総合研究所
Electrotechnical Laboratory, AIST, MITI
E-mail: {tatebe,kodama,sekiguchi,yamaguti}@etl.go.jp

2. MPI の point-to-point 通信

MPI において計算と通信のオーバラップを行い, 通信遅延を隠蔽するためには, ノンブロッキング通信あるいはマルチスレッド処理を用いる必要があるが, マルチスレッド処理に関しては MPI では規定されないため, ポータブルに実装を行うためには以下のノンブロッキング通信を用いることになる.

```
int MPI_Isend(buf, count, datatype, dest, tag,
              comm, req)
    void *buf;
    int  count, dest, tag;
    MPI_Datatype datatype;
    MPI_Comm comm;
    MPI_Request *req;

int MPI_Irecv(buf, count, datatype, source, tag,
              comm, req)
    void *buf;
    int  count, source, tag;
    MPI_Datatype datatype;
    MPI_Comm comm;
    MPI_Request *req;
```

ここで, buf は送, 受信バッファ, count, datatype は送, 受信データのデータ数とデータ型, dest, source はそれぞれ送信先, 送信元, comm はコミュニケータ, req はそれぞれ送, 受信リクエストである. ノンブロッキング送受信は実際の通信処理とは無関係にローカルに (自プロセスの処理だけで) 終了し, それぞれの通信リクエストを返す.

通信の完了のためには, MPI_Wait() 等を用いる. MPI_Wait() は送信または受信バッファが利用可能☆となるまでブロックし, 受信情報を引数として渡されるステイタスに入れる. ブロッキング通信はこの通信完了までブロックされる.

ノンブロッキング通信を用い, 通信と計算のオーバラップを行うためには, 以下のように記述する.

```
/*  通信リクエスト発行  */
MPI_Irecv(recv_buf, n, datatype, source, tag,
          comm, &recv_req);
計算 1
MPI_Isend(send_buf, n, datatype, dest, tag,
          comm, &send_req);
計算 2          //  ここでオーバラップ
// 送信リクエスト待ち
MPI_Wait(&send_req, &send_status);
計算 3          //  ここでもオーバラップ
// 受信リクエスト待ち
MPI_Wait(&recv_req, &recv_status);
```

ノンブロッキング送受信が発行され, MPI_Wait() 等により完了を待つ間が通信と計算のオーバラップが行われる余地のある部分となり, 上のコードでは計算 2 と計算 3 がその部分にあたる. 計算 1 はオーバラップのためには必要ないが, この論文における提案である片側送信を行うために必要な部分となる.

☆ 送信の場合は, 送信バッファのデータは既にコピーあるいは転送されて書き込み可能, 受信の場合は受信バッファにデータを受信したという意味である.

3. Point-to-point 通信のリモートメモリ操作による実装

3.1 リモートメモリ操作

リモートメモリ操作は他プロセッサのローカルメモリの操作を行うことであり, 富士通 AP1000+ の PUT/GET, 日立 SR-2201 のリモート DMA 等のようにハードウェアによる実装と, Berkeley Active Messages を用いる等のソフトウェアによる実装がある. ハードウェアによる実装では, ローカルメモリを持つプロセッサの実行を妨げること無くリモートメモリ操作を行うことができるが, ソフトウェアによる実装ではローカルメモリを持つプロセッサがメッセージハンドラの起動等の処理を必要とするため, 実際の通信遅延以外のオーバヘッドが大きい.

共有メモリ計算機, 分散共有メモリ計算機におけるロード, ストア命令も, ローカルメモリ, リモートメモリという概念は薄れるが, リモートメモリ操作と考えることができる.

3.2 片 側 送 信

対応する送信と受信のマッチングは送受信先, メッセージタグ, コミュニケータにより行われるが, 受信側で送信元, メッセージタグにワイルドカードを指定している場合, また MPI のプロセスがマルチスレッド実行している場合, また複数受信完了命令を実行している場合等では, その対応関係に非決定性がある. その非決定性を任意の静的順番で解消した場合, デッドロック, スターベイションの可能性があり, それらの静的解析が必要となる. MPI-2 では動的プロセス生成, また別々に立ち上がった MPI プロセス同士の通信をサポートしており, 対応する送受信の静的解析をますます困難なものとしていている. いずれにしても動的, ランタイムの効率的な送受信のマッチング機構が重要となっている.

ワイルドカード指定の受信の場合, 基本的には受信した順番に送受信のマッチングが行われるため, マッチングは通常受信側で行われる. しかしながらこの受信側でマッチングを行う方法では, 送信時に受信側へ必ず問い合わせを行う必要があり, その問い合わせは通信コストがかかるだけではなく受信側のプロセスに影響を及ぼすため, 通信遅延が増大する原因となる.

我々のアプローチでは, 送受信のマッチングを送信側で行い,

- (MPI_Irecv() を先行発行し,)MPI_Irecv() で送信側の連想メモリに受信アドレス, 受信リクエストアドレス, メッセージタグ, コミュニケータを書き込む (以後この処理を受信要求という).
- MPI_Isend() あるいは MPI_Send() ではローカルにある連想メモリを調べ, 対応する受信要求があればリモートメモリ書き込みを行う.

という実装方式 (図 1) をとる. この方式により, 送信側

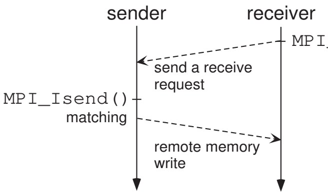

図 1　送信側でメッセージマッチングを行う片側送信

は送信時に受信側に問い合わせることなく，リモートメモリ書き込みによる送信を行うことができる．以後この送信を片側送信と呼ぶ．ここでの連想メモリのキーは送信先，メッセージタグ，コミュニケータである．また，同一キーを持つ受信要求が複数起る可能性があり，MPI の仕様によりそれらの順番を守らなければならないため，連想メモリのそれぞれのエントリはキューになっている必要がある．

しかしながら，受信の送信元としてワイルドカードを指定した場合，送信側に受信要求を送ることができないため☆，この送信側のマッチングだけで全てのマッチングを行うことは効率が悪く，受信側におけるマッチングが必要となる．一方で，point-to-point 通信を片側送信で実装するためには送信側で対応する送受信のマッチングを行う必要があるため，送信側，受信側双方における送受信マッチングが必要になる．次節では矛盾のない送信側，受信側双方における送受信のマッチングのデザインについて述べる．

4.　送受信のマッチングの一貫性制御

送信側，受信側双方によるマッチングを行う場合，理想的には受信側がワイルドカード指定の受信であるかどうか，また送信に対しその対応する受信の発行が遅くないかどうかで，送信側でマッチングを行うか，受信側で行うかの判断を行えば良いが，それらはマッチングを行ってはじめて判断出来るものであり，いずれにしても送信側だけでは送信側，受信側のどちらでマッチングを行うのか判断することができない．そのため，一貫性を保つための制御が必要となる．

4.1　受信側によるメッセージマッチング

受信側で対応する送受信のマッチングを行う場合，いつマッチングを行うのか，またメッセージのコピーを行うかにより幾つかプロトコルがある．メッセージのコピーを行わない零コピー通信の場合，送信側は送信バッファ，リクエストオブジェクトのアドレス，メッセージタグ，コミュニケータ (以後送信要求) を受信側に送り，対応する受信が既に発行されているか調べるためにマッチングを行う．このマッチングが成功した場合，受信側はリモートメモリ参照によりメッセージの受信を行う (図 2)．失敗した場合，受信側は送信要求を連想メモリに格納し，対応する受信が発行されたときにリモートメモリ参照によりメッセージの受信を行う (図 3)．以後この方法を同期プロトコルと呼ぶ．同期プロトコルは MPIAP[7),8)] の protocol method とほぼ同じ方法で，一般に零コピー通信を行うためにはこのプロトコルとなる．同期プロトコルはメッセージの余計なコピーを必要としないが，必ず MPI の同期モードの送信となってしまうことに注意が必要である．

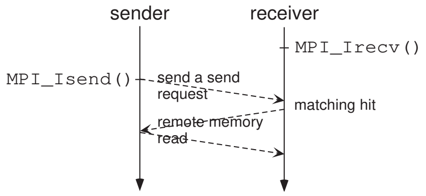

図 2　ノンブロッキング受信が既に発行されている場合の同期プロトコルと get プロトコル

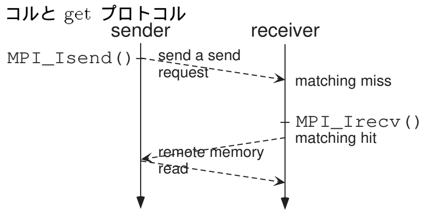

図 3　ノンブロッキング受信が発行されていない場合の同期プロトコル

同期プロトコルでは，対応する受信が発行され，リモートメモリ参照が終了するまでブロッキング送信 `MPI_Send()` (あるいはノンブロッキング送信の送信完了命令) はブロックしてしまう．MPI の標準モードの送信をこの同期プロトコルで実装する場合，このブロックは性能低下の原因となってしまう．必要以上のブロックを防ぐためには，(システム，ライブラリ領域の) 一次バッファを用い，一度送信バッファから別のバッファへコピーを行う．この一次バッファはリモートメモリ操作が使える場合送信側，受信側どちらにもっても構わないが，受信時の遅延を少しでも小さくするために受信側に持つ事にする．この時，送信側は受信側の一次バッファのアドレスを知らないので，一度受信側にそのアドレスを問い合わせる必要がある．ここで，その問い合わせに対し対応する受信が発行されているかのマッチングを行うか行わないかで，二つの方法がある．問い合わせ時にマッチングを行わない場合，送信側は受信側に一次バッファの確保を要求し，そのアドレスにリモートメモリ書き込みを行い，送信要求を受信側に送る (図 4)．同様の操作は一次バッファ確保の後，受信側がリモートメモリ参照を用いても行うことができる．以後この方法をバッファプロトコルと呼ぶ．問い合わせ時にマッチングを行う場合，既に対応する受信が発行されていたら，同期プロトコル同様リモートメモリ参照により余計なコピー

☆ 受信要求をブロードキャストしたとしても，対応する送信を一つだけ決定するための処理が必要となり，送信側だけでリモート書き込みによる送信を行うことはできない．ワイルドカード指定が送信元ではなくメッセージタグの場合は問題なく受信要求を送ることができる．

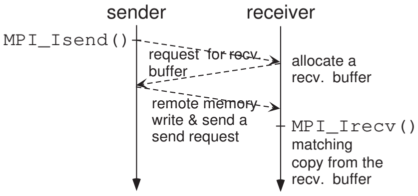

図 4 バッファプロトコル

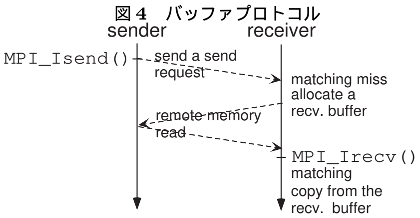

図 5 ノンブロッキング受信が発行されていない場合の get プロトコル

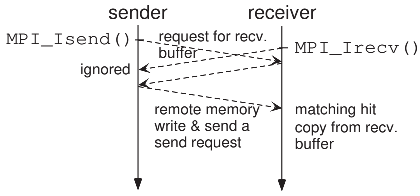

図 6 バッファプロトコルの場合

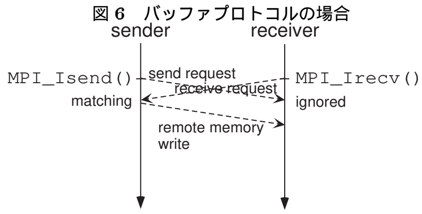

図 7 同期プロトコルまたは get プロトコルの場合

なしに通信が行われる (図 2). 対応する受信が発行されていなければ, バッファプロトコル同様に受信側が一次バッファを確保し, 送信側がリモートメモリ書き込みを行うか, 受信側がリモートメモリ参照を行う (図 5). 以後, この方法を get プロトコルと呼ぶ. このプロトコルは MPICH の get プロトコルとほぼ同じである.

4.2 双方によるメッセージマッチングの一貫性制御

送信側でメッセージマッチングを行い, リモートメモリ書き込みにより送信を行う片側送信は, `MPI_Irecv()` が先行発行されているコードを効率的に実行するためのデザインであるため, 基本的なポリシとして送信発行時に対応する受信要求がない場合は, 片側送信をあきらめる事にする. この時,

- 送信側は対応する受信要求があればリモートメモリ書き込みでメッセージを送信し, なければ送信要求を送る.
- 受信側は対応する送信要求があればリモートメモリ参照, あるいはローカルメモリコピーで受信する. 送信要求がない場合, 送信元がワイルドカード指定でなければ受信要求を送る.

しかしながら, このデザインでは送信と受信がほぼ同じタイミングで発行され, 送信側は対応する受信要求がないから送信要求を送り, 受信側も同様に対応する送信要求がないので受信要求を送った場合, お互いがお互いにマッチングを依頼することになり, 正しくマッチングを取ることができない.

基本的な解決策はどちらかの要求を無視することである. バッファプロトコルの場合は, 送信要求により受信側へメッセージをコピーしてしまうため受信要求を無視するのが適当であり (図 6), 同期プロトコル, get プロトコルの場合は, リモートメモリ書き込みの方が速いため, 送信要求を無視するのが妥当である (図 7). この解決策を実装するために, 送信側, 受信側ともに送信要求, 受信要求を出した時に, 要求を出した (相手にマッチングを依頼した) ことを示すキュー☆(送信, 受信要求キュー) にそれらの要求を入れる. 無視すべき要求が来た場合, その要求を無視し要求キューから取り除く. ただしこの時, 一貫性に問題がなく無視すべき要求が来ない場合にも, 次の無視すべきでない要求が無視されないように要求キューから取り除く処理が必要となるため, 注意が必要である.

また, 連想メモリによるメッセージマッチングが成功し, エントリを消去, あるいは失敗し要求キューにエントリ入れるという操作はクリティカルセクションとなるため, マルチスレッドで実行している環境では排他ロックが必要となるため注意が必要である.

4.3 通信の順序

MPI では通信の順序関係について 2 プロセス間では保証している. 我々の提案している方法は, 送信元にワイルドカード指定を行わない限り, 連想メモリが FIFO であり, ネットワークが FIFO を保証していれば通信の順序関係は保証される. 連想メモリ, ネットワークが FIFO であれば送信, 受信要求を出した順番を 2 プロセス間で保証することができるため, 発行した順にマッチングを行うことができるからである.

問題になるのが, 送信元にワイルドカードの指定をする受信を含む場合である. 送信元にワイルドカードを指定した場合, 受信要求を出さないため, 後続の受信により受信要求を出してしまうと, 送信側でその要求とマッチングを行ってしまう. 従って, MPI の要求する FIFOness を守るためには, 送信元にワイルドカードの指定をする受信を発行した後は, その受信に対応する送信が発行されるまで受信要求を発行することができない.

5. MPI-EMX

送信側, 受信側双方による対応する送受信のマッチングを行う MPI を高並列計算機 EM-X[3] に実装を行っ

☆ キューと書いているが, 送信要求, 受信要求と同様に連想メモリ, ハッシュ表を用いる.

た.

5.1 EM-X

EM-X は電総研で開発された分散メモリ型並列計算機であり, 要素プロセッサ EMC-Y[2] はデータ駆動機構に基づくパケット送出機構, パケットマッチング機構を備え, また基本的な実行単位となる強連結ブロック内ではノイマン型計算機に基づく逐次実行機構を備えている. EM-X は現在 80 PE のシステムが 16MHz で動作しており, ネットワークはサーキュラオメガ網である.

強連結ブロックはアトミックに実行され, 割り込み等により途中で中断することのない命令ブロックであり, 1 入力パケットあるいは 2 入力パケットで起動される. パケットはアドレス部, アドレスタグ部, データ部, データタグ部の 2 ワード☆で構成されている. パケットが到着すると, 入力バッファに格納される. 2 入力待ち合わせパケットの場合はここでマッチングがとられ, もう一方がまだ到着していなければメモリに待避させられる. 入力バッファ中のパケットは強連結ブロックを実行可能なパケットであり, それらは FIFO で実行される. また, 強連結ブロックはオペランドセグメントと呼ばれる関数フレームを用いることによりコンテキストを保存する事ができる. 以後, 強連結ブロックの起動のことをスレッド起動, 他プロセッサのスレッド起動を特にリモートスレッド起動と呼ぶ.

I-structure は関数型言語に導入された 1 read, 1 write のデータ構造であり, EM-X では 2 入力待ち合わせパケットを用い実装されている. また I-structure を FIFO キューを用い複数 read, 複数 write に拡張した Q-structure も同様に実装され, キューに関してはソフトウェアで実装されている.

リモートメモリ操作はリモートスレッド起動で実装されているが, SYSWR, SYSRD パケットと呼ばれる特殊パケットに関してはハードウェアで実装され, 入力バッファにそれらのパケットが入ると, 実行ユニットを通らないでそれぞれリモートメモリ書き込み, リモートメモリ参照を行う.

プログラミング言語としてはマルチスレッド処理, 大域ポインタ, I-structure, Q-structure などの拡張を行った C 言語 EM-C[6] が開発されている.

5.2 MPI-EMX の実装

我々の提案している片側送信の実装のためには, リモートメモリ操作の他に送信要求, 受信要求の発行が必要である. これらの操作は他プロセッサの連想メモリへの書き込みであるが, 連想メモリの各要素はキューであり, また書き込む前に対応する要求が既に送られたかどうか調べるために他プロセッサの要求発行キューを調べる必要があるため, これらの処理はリモートスレッド起動により実装した. この時, クリティカルセクションでは排他ロックを掛ける必要があるため注意が必要である.

我々の実装では連想メモリのキューの操作が主な通信遅延となるため, アセンブラを用い高速化を図った. また, アセンブラにより強連結ブロックとして記述したことにより, この排他ロックを明示的に行うことなく実装している.

メッセージの通信にはリモートメモリ操作を用いているが, メッセージの通信だけではなく MPI の受信リクエストオブジェクトへの送信元, メッセージタグ, サイズの書き込みもリモートメモリ操作を用いている.

また受信完了を調べるためにはポーリング, 割り込みという方法が良くとられるが, EM-X では I-structure をサポートしているため, 受信待ちには I-structure の局所同期機構を用いた. この機構によりプロセッサに負荷を与えることなく, 理想的な受信確認を行うことができる. ここで問題になるのは`MPI_Test()` では受信完了かどうかのチェックを行う必要があるが, I-structure を用いる場合 read がブロックしてしまうためそのチェックが出来ないことである. この問題に対して我々は I-structure にブロックしない非同期 read 操作を加え, EM-X に実装を行った. この非同期 read 操作は, write が発行されていれば通常の read と同じであり, 発行されていない場合はブロックしないですぐに戻るというもので, EM-X では I-structure のタグの部分を調べることで実装を行っている.

対応する送受信のマッチングに用いる連想メモリとして Q-structure の機構を用いる事も考えられるが, 幾つかの問題点がありごく限られた状況でしか使用することができない. 一つの問題は Q-structure のアドレス空間で, MPI では送受信プロセス, メッセージタグ, コミュニケータの 3 ワード (12 バイト, あるいは 96 ビット) が必要となることである. もう一つの問題は受信でワイルドカード指定をした場合, Q-structure のアドレスを定めることができないということである.

EM-X の通常のリモートスレッド生成は, オペランドセグメントの要求, 引数のリモートメモリ書き込み, リモートスレッド起動という処理となるが, 引数が 2 ワード以下でかつオペランドセグメントが必要ない場合は, 特殊パケットハンドラを用いることにより特殊パケットだけでスレッド起動を行うことができる. 我々の実装では送信要求, 受信要求, 要求キューからの取り除きをリモートスレッド起動で実装しているが, 要求キューからの取り除きに関しては引数が少ないため, 特殊パケットハンドラにより実装を行った.

その他, 送信 / 受信リクエスト領域は固定長領域となるため, その領域の確保, 廃棄を, フリーリストを用いアセンブラ記述を行い最適化を行った. また, `MPI_Wait()` もアセンブラ記述を行い高速化を図っている.

6. 性能評価とプログラミング例

この節では MPI-EMX の性能評価と, MPI ライブラ

☆ 1 ワードはデータ 32 ビット, タグ 6 ビットの 38 ビットである.

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Barrier(MPI_COMM_WORLD);
if (rank == 1) {
    time = MPI_Wtime();
    for (i = 0; i < NUM_ITER; ++i) {
        MPI_Irecv(rbuf, 1, MPI_INT, 2, 100,
                  MPI_COMM_WORLD, &req);
        MPI_Wait(&req, &st);
        MPI_Isend(sbuf, 1, MPI_INT, 2, 100,
                  MPI_COMM_WORLD, &req);
        MPI_Wait(&req, &st);
       /*  or MPI_Request_free(&req); */
    }
    time = MPI_Wtime() - time;
}
else if (rank == 2) {
    time = MPI_Wtime();
    for (i = 0; i < NUM_ITER; ++i) {
        MPI_Isend(sbuf, 1, MPI_INT, 1, 100,
                  MPI_COMM_WORLD, &req);
        MPI_Wait(&req, &st);
        /*  or MPI_Request_free(&req); */
        MPI_Irecv(rbuf, 1, MPI_INT, 1, 100,
                  MPI_COMM_WORLD, &req);
        MPI_Wait(&req, &st);
    }
    time = MPI_Wtime() - time;
}
```

図 8 普通の pingpong のコード

リが効率的に実行できる余地のあるプログラミングについての解説を行う.

6.1 Pingpong

MPI を用い pingpong のプログラムを書くと, 図 8の様に一方が送信し, 受信を行い, もう一方が受信し, 送信を行うというプログラムになり, 図 8ではそれらを繰り返して時間を計測している.

しかしながら, このプログラムでは, MPI_Irecv() を先行発行している訳ではないため, 片側送信が行われるとは限らない. そこで, このプログラムを図 9の様に予め全ての MPI_Irecv() を発行してから pingpong を始めるように書き換える☆. このプログラムでは図 8と異なり先行発行した分の受信リクエストが必要になる. この場合, 送信を行う時には既に受信要求が出ているため, 常に受信側に問い合わせることなくリモートメモリ書き込みによる片側送信となる.

図 8 及び図 9 のプログラムを用い, 双方によるマッチング, あるいは受信側によるマッチングを行い, それぞれのプロトコルにおける MPI-EMX の pingpong 転送の片道の時間を表 1 に示す. 受信を先行発行するプログラムでは, 双方でメッセージマッチングを行う場合, 常に送信側でマッチングが成功するため, プロトコル間の差はなく, この時通信遅延は 15.4 μsec. であった.

図 8 の普通の pingpong のプログラムでは, 同期プロトコル, get プロトコルは送信側のマッチングは全て失敗するため, その分受信側だけでマッチングを行った方

☆ ベンチマークとしての意味はこのプログラムで正しいが, 一般的には等しい受信アドレスのノンブロッキング受信を先行発行すると race 状態となるため, 別々の受信アドレスを割り当てる必要がある.

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 1) {
    for (i = 0; i < NUM_ITER; ++i) {
        MPI_Irecv(rbuf, 1, MPI_INT, 2, 100,
                  MPI_COMM_WORLD, &req[i]);
    }
    MPI_Barrier(MPI_COMM_WORLD);

    time = MPI_Wtime();
    for (i = 0; i < NUM_ITER; ++i) {
        MPI_Isend(sbuf, 1, MPI_INT, 2, 100,
                  MPI_COMM_WORLD, &r);
        MPI_Wait(&r, &s);
        MPI_Wait(&req[i], &st[i]);
    }
    time = MPI_Wtime() - time;
}
else if (rank == 2) {
    for (i = 0; i < NUM_ITER; ++i) {
        MPI_Irecv(rbuf, 1, MPI_INT, 1, 100,
                  MPI_COMM_WORLD, &req[i]);
    }
    MPI_Barrier(MPI_COMM_WORLD);

    time = MPI_Wtime();
    for (i = 0; i < NUM_ITER; ++i) {
        MPI_Wait(&req[i], &st[i]);
        MPI_Isend(sbuf, 1, MPI_INT, 1, 100,
                  MPI_COMM_WORLD, &r);
        MPI_Wait(&r, &s);
    }
    time = MPI_Wtime() - time;
}
else {
    MPI_Barrier(MPI_COMM_WORLD);
}
```

図 9 ノンブロッキング受信を先行発行する pingpong のコード

表 1 MPI-EMX の pingpong 通信の片道時間 (μsec.)

	both sides			only receiver side		
	buf.	get	sync	buf.	get	sync
通常	36.1	25.4	27.9	42.8	21.4	22.1
先行発行	15.4			42.9	20.6	21.4

が速くなっているが, バッファプロトコルは初めにメッセージを受信するプロセスの送信側のマッチングが全て成功するため, 双方でマッチングをする方が速くなっている.

同期プロトコル, get プロトコルのいずれも, 受信を先行発行して双方でマッチングを行う場合以外では, 受信側での初めのマッチングが成功し, 本質的にこれらのプロトコル間の差はない. get プロトコルのほうが少し速くなっているのは実装上の問題である.

零バイトの通信を行う場合, 図 9のプログラムで 15.3 μsec. であった. この遅延の大部分はプロセッサローカルの処理であり, プロセッサがもう 10 倍速ければ 2 - 3 μsec. 程度になるはずである.

次に同プログラムを AP1000, AP1000+, Cenju-3, SR-2201, Wiz で実行させた場合の, pingpong の片道の時間を表 2に示す. Wiz[11] は DEC Alpha 333 MHz の WS を 100Base/TX Ether switch でつないだクラスタである. AP1000, AP1000+ では MPIAP[7),8)] を用い, Cenju-3 では mini-MPI を用いた. SR-2201, Wiz に関しては MPICH を用いている.

表 2　Pingpong (μsec.)

	AP	AP+	Cenju-3	SR2201	Wiz
通常	72.8	26.1	48.9	32.6	199
先行発行	80.3	27.0	50.2	79.9	197

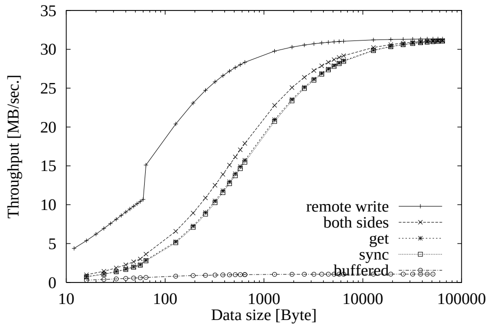

図 10　EM-X の通信スループット

その他のシステムでは送信側で送受信のマッチングを行っていないため図 8及び図 9のプログラムによる差が余りない．それどころか, SR2201 の様に倍以上遅くなってしまったり, Cenju-3 の mini-MPI の様に 1000 程度の `MPI_Irecv()` の先行発行でも実行できないシステムもあり☆, 現在使用可能な MPI では受信を先行発行しても功を奏さないシステムが多いのも現状である．

しかしながら重要なのは図 9の様に`MPI_Irecv()` を先行発行するプログラムは MPI ライブラリの実装として効率的に実行することが出来る余地があるということであり, ハードウェアの機能を十二分に使うことの出来る可能性があるということである．

6.2　スループット

`MPI_Irecv()` を先行発行する pingpong プログラム (図 9) を用い, 通信スループットの測定を行った．基本的にこの通信パターンでは送信側は受信要求の確認, リモートメモリ転送を行うだけであり, スループットに関してはほぼリモートメモリ転送の性能が出ることが期待される．

図 10に MPI-EMX のそれぞれのプロトコルのスループットとリモートメモリ書き込みのスループットを示す．remote write と書いてある一番上の線は用いているリモートメモリ書き込みの性能で, この関数は 64 バイト単位でループアンローリングを行っているため, 64 バイトあたりで段差ができている．

双方でマッチングを行う場合が, スループットの立ち上がりが一番良く, 512 バイトでピークの半分の性能を越え, 図にはないが 1.6 MB のデータでは 31.4 MB/sec. の性能であった．このスループットはリモートメモリ書き込みのスループットとほぼ等しい性能である．が, pingpong の節でみたように零バイトの通信でも遅延が 15.3 μsec. もあるため, 立ち上がりが少し鈍くなっている．

EM-X は通信遅延が極めて小さいことと, このプログラムでは受信側は他に計算をすることなく受信のためにただ待っているだけであるため, 受信側の処理が速く, 同期プロトコル, get プロトコルを用い受信側だけのマッチングによるスループットとの差は僅かなものになっている．バッファプロトコルは必ず一次バッファを割り当て, コピーを行うためスループットはデータサイズを増やしても大きくならない．

6.3　後退代入

後退代入はガウスの消去法の求解部にあらわれ, 上三角行列の解法である．上三角行列を行ブロックサイクリックに分割し, パイプライン処理により並列実装を行っている．

表 3に 5 台のプロセッサを用い, 行列の大きさ 2000, ブロックサイズ 25 の場合の結果を示す．単体プロセッサにおける性能は 3.66 MFlops であり, 先行発行の場合, 双方でマッチングを行うときは 85% の並列効率が出ている．

後退代入のプログラムの場合, EM-C の生成するコードではブロック中のデータの計算の二重ループが強連結ブロックとなり, 実行中は他の強連結ブロック (スレッド) は起動できない．表中 w. `yield()` は `yield()` をループ中で明示的に呼んで強連結ブロックを中断させているプログラムである．

受信側だけでメッセージマッチングを行う場合, 同期プロトコル, get プロトコルはループ中で `yield()` を呼ばないと, 受信側で送信側が生成したスレッドの起動が遅れ, 極めて性能が低下してしまっている．

しかしながら, `yield()` は計算にとっては純粋なオーバヘッドである．受信の先行発行と双方によるメッセージマッチングを行う場合, 片側送信となり受信側のプロセスに影響を及ぼさないため, `yield()` を呼ぶ必要がない．片側送信は, 単に通信遅延を減少させるだけではなく, 全体としての性能を上げることができ, 重要な実装方式といえる．

7. 関 連 研 究

MPI の実装は現在までに数多くあるが, もっとも多く使われているのが MPICH[1] である．MPICH はメッセージ通信のプロトコルとして eager, rendezvous, get の 3 つを持っている．が, いずれのプロトコルも送信側はまずコントロールメッセージを送り, 受信側の返事を待ち, 送受信のマッチングは受信側で行われる．この場合, いくら `MPI_Irecv()` を先行発行しても, 送信時に一度受信側にコントロールメッセージを送るため, 我々が提案している手法に比べ通信遅延が大きくなってしまう．また, MPI-FM[4], MPI-AM[10] 等の MPICH を用いている実装では送信側で送受信のマッチングは行われ

☆ 表 2の Cenju-3 のデータは 100 回先行発行したものである．

表 3 MPI-EMX の後退代入の性能 (MFlops/#PE)

		both sides			only receiver side		
		buf.	get	sync	buf.	get	sync
w/o yield()	通常	2.55	2.33	2.17	2.55	0.71	0.71
	先行発行	3.10			2.55	0.71	0.71
w. yield()	通常	2.73	2.78	2.84	2.73	2.78	2.84
	先行発行	2.84			2.74	2.85	2.85

(#PE = 5, N = 2000, n of block = 25)

ていない.

MPIAP[7),8)] は富士通 AP1000, AP1000+, AP3000 における MPI の native な実装であり, 送信方式として in-place method, protocol method という二つの方式を実装している. さらに受信方法もポーリング, シグナル (割り込み), それらを合わせた方法を実装し, 比較検討を行っている. が, やはりそれらは全て受信側で送受信のマッチングを行っている.

8. 終わりに

MPI の point-to-point 通信において, 対応する MPI_Irecv() が先行発行されるパターンで送信が受信側に問い合わせることなくリモートメモリ書き込みにより送信を行う片側送信の実装についての提案と検討を行った. EM-X における実装では, 通信遅延が 15.3 μsec. であった. この遅延は他 MPP システムに比べても少ないが, この遅延の大部分はプロセッサローカルの処理であり EM-X のプロセッサスピードが今の 10 倍となれば 2 - 3 μsec. 程に短縮される見込みがある. また, スループットに関してはほぼピーク性能であった. が, 片側送信は単に遅延を減少させ, スループットを向上させるだけではなく, 受信プロセスの実行に影響を及ぼさないためプログラム全体の性能を上げることができ, ハードウェアの性能を十二分に出すためには片側送信が不可欠といえる.

ここで対応する MPI_Irecv() が先行発行されるパターンは現在の処理系では必ずしも功を奏するとは限らないが, 重要なのは MPI の実行時の処理としてハードウェアの機能を用いる余地を与え, 効率的に実行する余地のあるプログラミングを行うということである.

現在, point-to-point 通信, バリア同期, 一部のリダクション演算, Cartesian トポロジ等が実装されており, 今後フルスペックの仕様をサポートしていく予定である.

9. 謝辞

本研究を遂行するにあたり貴重なご助言, ご討論いただいた大蒔和仁部長, 佐藤三久氏, 坂根広史氏, 山名早人氏, 小池汎平氏ほか, ご協力いただいた諸氏に感謝致します.

参考文献

1) 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, pp. 789–828 (1996).

2) Kodama, Y., Koumura, Y., Sato, M., Sakane, H., Sakai, S. and Yamaguchi, Y.: EMC-Y: Parallel Processing Element Optimizing Communication and Computation, *Proceedings of ICS'93*, pp. 167–174 (1993).

3) Kodama, Y., Sakane, H., Sato, M., Yamana, H., Sakai, S. and Yamaguchi, Y.: The EM-X Parallel Computer: Architecture and Basic Performance, *Proceedings of ISCA'95*, pp. 14–23 (1995).

4) Lauria, M. and Chien, A.: MPI-FM: High Performance MPI on Workstation Clusters, *Journal of Parallel and Distributed Computing*, Vol. 40, No. 1, pp. 4–18 (1997).

5) Message Passing Interface Forum: *MPI-2: Extensions to the Message-Passing Interface* (1997). http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html.

6) Sato, M., Kodama, Y., Sakai, S., Yamaguchi, Y. and Koumura, Y.: Thread-based Programming for the EM-4 Hybrid Dataflow Machine, *Proceedings of ISCA'92*, pp. 146–155 (1992).

7) Sitsky, D. and Hayashi, K.: Implementing MPI for the Fujitsu AP1000/AP1000+ using Polling, Interrupts and Remote Copying, *Proceedings of JSPP'96*, pp. 177–184 (1996).

8) Sitsky, D. and Hayashi, K.: An MPI library which uses polling, interrupts and remote copying for the Fujitsu AP1000+, *Proceedings of ISPAN'96*, IEEE (1996).

9) Snir, M., Otto, S., Huss-Lederman, S., Walker, D. and Dongarra, J.: *MPI: The Complete Reference*, The MIT Press (1996).

10) Wong, F. C. and Culler, D. E.: *Message Passing Interface Implementation on Active Messages*. http://now.CS.Berkeley.EDU/Fastcomm/MPI/.

11) 益口 摩紀, 建部 修見, 関口 智嗣, 長島 雲兵, 佐藤 三久: アルファワークステーションのクラスタ etlwiz の性能評価, 情報処理学会研究報告, 98-HPC-70, pp. 61–66 (1998). HOKKE'98.