計算機アーキテクチャ 102－7
（1993．10．21）

並列計算機 EM-4/X の RICA による メッセージ通信の実現

佐藤三久 児玉祐悦 坂井修一† 山口喜教
電子技術総合研究所
RWC つくば研究センタ†

本稿では、並列計算機 EM-4、ならびに現在開発中の EM-X での RICA による細粒度パケットによるメッセージ通信の実現とその性能について述べる。EM-4/X では、そのデータフロー機構により、細粒度パケットによる通信が効率的に処理できる。その結果、従来のメッセージ通信型プロセッサに比べて、極めて低レーテンシの通信が可能になるだけでなく、相手のメモリに対し直接書き込めることにより、柔軟なプログラミングが可能になる。簡単なメッセージ通信における実効性能、いくつかのアルゴリズムで complete exchange や broadcast などの集中的な通信操作の性能を測定し、十分が性能が得られていることがわかった。

Implementation of Message Passing by RICA in the EM-4/X

SATO Mitsuhisa, KODAMA Yuetsu, SAKAI Shuichi†, YAMAGUTI Yoshinori
Electrotechnical Laboratory
1-1-4 Umezono, Tsukuba, Ibalaki 305, Japan
RWC Tukuba Research Center†

In this paper, we describe the implementation and the performance of message passing by RICA (Reduced Interprocessor Communication Architecture) in the EM-4 and EM-X multiprocessor. Dataflow mechanism of EM-4/X supports fine-grain packet communication very efficiently, which provides low latency communication, and flexible message-passing with direct remote memory access. We measured the sustainable bandwidth of simple messages, and the performance of communication intensive operations such as complete exchange and broadcast by several algorithms.

1 はじめに

本稿では、我々の開発した並列計算機 EM-4[4] ならびに開発中の EM-X[8, 1] におけるメッセージ通信の実現方式と性能について述べる。

現在、その構成の容易さやスケーラビリティから、従来の逐次プロセッサをノードとするメモリ分散型マルチプロセッサが典型的な形態の一つとなっている。このタイプのマルチプロセッサ上でのプログラミングとして、ノードごとに逐次プログラムを記述し、これらの間の同期、データの交換は、メッセージ通信で行なう方法が一般的になっている。このメッセージ通信によるプログラミングではプログラムを各ノードのプログラムとしてメッセージ通信の send/receive で同期・データ交換をするように再構成しなくてはならなく、プログラミング上、共有メモリ型のプログラムよりも難しいという難点はあるものの、プロセッサ間の通信に規則性がある場合にはプログラミングは比較的容易であり、陽に必要なデータが相手のメモリに書き込みが行なわれるため、効率的である場合もある [2]。特にデータ並列性がある場合には通信は規則的になることが多く、データ並列言語コンパイラを用いて自動的にメッセージ通信操作に変換し、分散メモリ・メッセージ通信型のプロセッサで効率的に実行する研究が多く行なわれている。

EM-4 は、データ駆動機構をもつ分散メモリ型の並列計算機である。データ駆動機構は、プロセッサ間の通信によって、それに対応するスレッドを高速に起動・同期することを可能にしている。プロセッサはネットワークに対してメッセージを直接送りだしたり、受けとったメッセージに対するスレッドの起動をハードウエアの機構としてサポートしている。EM-4 では、この機構はデータフロー実行だけでなく、遠隔のメモリの読みだし・書き込みにも拡張されている。我々はこのアーキテクチャを RICA (Reduced Interprocessor Communication Architecture) を呼んでいる。EM-4 は、基本的には分散メモリのマルチプロセッサであり、各プロセッサはローカルなメモリをもち、グローバルな共有メモリはもたない。しかし、メッセージはシステムで定義されている特定のスレッドを実行させるようにすることができる。この機能を用いて、他のプロプロセッサのメモリに対してアクセスすることが可能ある。

我々は、EM-4 のプログラミングのために C 言語の superset である EM-C を開発した。EM-C では、逐次実行をベースとし、プロセッサの並列性の制御、メッセージ通信によるデータの分散、同期は、陽に指定して行なう。このプログラミングモデルを、データフロー計算機本来のデータフローモデルに対して、マルチスレッドプログラミングモデルと呼んでいる [5]。EM-C はリモートメモリアクセスの機能を用いて、ユーザに対して、仮想的な共有メモリを提供しており、共有メモリ的なプログラミングも可能になっている [7]。

EM-4 ではすべての通信は、2 ワードからなるパケットを単位として行なわれる。EM-4 のネットワークは FIFO 性を保証しており、この性質を利用し、データの相手のメモリへの書き込みのパケット、同期のためのパケットにより、メッセージ通信操作を実現した。これにより、低レーテンシでかつ、柔軟なメッセージ通信が可能なことがわかった。

2 章において、並列計算機 EM-4 の概要について述べ、3 章で、EM-4/X での細粒度パケットによるメッセージ通信の方法、その実現について述べる。4 章で、実際の性能について報告する。

2 並列計算機 EM-4

2.1 アーキテクチャの概要

EM-4 の要素プロセッサ EMC-R[9] は、それぞれローカルなメモリを持っており、サーキュラオメガネットワークで接続されている。

EMC-R は、RISC アーキテクチャとなっており、パイプラインは逐次の命令実行とデータ駆動機構のためのネットワークとの通信と同期処理が融合されるように設計されている。ネットワークに対して直接、メッセージを送ったり、メッセージでデスパッチできる。メッセージは、アドレス部とデータ部からなる 2 ワードの固定長で、これをパケットと呼んでいる。

パケットが到着するとデータ駆動機構によって、そのアドレス部で指定されるスレッドがデータ部にある値と共に起動される。パケットのアドレス部では、起動する関数フレーム（オベランドセグメントと呼んでいる）とオフセットを指定する。関数フレームの先頭をあらかじめ、関数コード（テンプレートセグメント）にリンクしておき、スレッド起動時にはオベランドセグメントの先頭からテンプレートセグメントが取り出され、オフセットで指定されるスレッドが起動される。

スレッドが終了すると、次のパケットがキューの中から取り出され、処理される。パケットは、データフロートークンとして解釈することができる。パケットにおいてマッチングの属性を指定すると、他

方のデータフロートークンが到着していない場合は、そのパケットは起動するスレッドに対応するメモリに退避され、他方のトークンが到着するとそれらのデータと共にスレッドを起動する。スレッドの終了は、命令フィールドにおいてプログラム中に明示され、スレッドが終了する前にレジスタの値などスレッドの live な値などを実行中のスレッドに対応する関数フレーム（逐次実行の場合のスタックにあたる）に退避しておくようにすることができる。

いくつかの機能を実現するためにパケットの解釈をソフトウエアで定義することができる。パケットにおいて、パケットタイプのフィールドを指定することによって、パケットタイプに対応するスレッドが実行される。これらのパケットを特殊パケットと呼んでおり、これを用いて他の PE のメモリの書き込み、読みだしを行なうことができる。また、他の PE に対してリモートに関数を起動するのに必要なの関数フレームなどのリソースの割り当てにも用いられている。

2.2 EM-4 プロトタイプ

EM-4 プロトタイプは、80PE から成り、1990 年 4 月から稼働している。EM-4 は、12.5MHz のクロックで動作しており、メモリ参照命令などを除く大部分の命令は 1 クロックサイクルで実行される。ネットワークの性能は PE のポート当たり 6.25 Mpackets/sec(60.9 Mbytes/sec) である。

2.3 EM4 プログラミング言語 EM-C

我々は、EM-4 上のプログラミング環境として、C 言語の superset である EM-C を開発した。本稿での評価に用いたプログラムは EM-C とアセンブラで記述されたライブラリで記述した。

EM-C は EM-4 のベース言語として、ユーザが並列プログラムを陽に書ける操作を提供することを目的としている。EM-C コンパイラは、現在のインプリメントでは基本的に関数を逐次に実行するために、1 つのスレッドあるいは逐次に実行される複数のスレッドにコンパイルする。

関数呼びだしは、パケットを使って行なっているため、スレッドの逐次実行は関数の呼びだし、リターンの単位でスケジューリングされる。しかしながら、プロセッサを一つのスレッドで占有されるのを避けるため、スレッドを明示的に終了させる基本関数が提供されている。ハードウエアのキューにあるパケットは、中断しているスレッドの最小のコンテキストと考えることができるが、パケットは、到着順 (FIFO) で処理されるため、それに対応するスレッドの FIFO でスケジュールされる。EM-4 では、メモリ書き込みパケットも同様にスケジュールされているが、EM-X ではメモリ書き込みパケットは他のスレッドが実行中でもバイパスして実行される。

3 メッセージ通信型プログラムの実現

従来の分散メモリ型のマルチプロセッサでメッセージ通信型プログラムでは、同じプログラムを全プロセッサにおき、一つのスレッドで実行するのが一般的である。スレッドが同期・メッセージ通信という点から、全プロセッサでほぼ同じ部分を実行するようなプログラムを SPMD(Single Program/ Multiple Data) 型のプログラムという。EM-C では、プログラムの先頭で全プロセッサにスレッドを生成することによって、容易に実現できる。

3.1 メッセージ通信の基本操作

我々は任意の PE に関数単位のスレッドを生成し、メッセージ通信を行なう方法をすでに提案している [6]。この方法では、指定された関数を実行するスレッドとは別に、メッセージを受けとるためのオペランドセグメントを割り当て、ここにおいてメッセージ通信を行なうための操作を実行する。対応するスレッドがメッセージを受けとる場合にはポートのオペランドセグメントにおいて待ち合わせを行なう。このセグメントはプログラマからは、スレッドと交信を行なうポート（あるいはメールボックス）として扱われる。

メッセージ通信型プログラムの SPMD プログラムでは、PE ごとに固定番地にポートとなるセグメントを割り当て、これをつかって、メッセージ通信を行なう。プログラムからは PE 内で実行されるスレッドは 1 つであるが、実際はメッセージを受けとるためのスレッドが実行されていることになる。

基本操作として、ワードメッセージ通信とパケット通信の 2 組の基本関数が提供されている。`NODE_PORT(pe_id)` は固定番地に割り当てられた PE のポートを得る関数である。

ワードメッセージ通信 —

```
word_send(NODE_PORT(pe_id),msg_type,word)
word = word_recv(msg_type)
```

このメッセージ通信は、バッファ付き非同期通信である。複数のメッセージが到着した場合には、バッ

ファリングされる。recv 操作は、指定されたメッセージが到着していない場合には、メッセージが到着するまでブロックする。

パケット通信 —

```
pkt_output(NODE_PORT(pe_id),msg_type,word)
word = pkt_input(msg_type)
```

このメッセージ通信は、非同期であるが、バッファリングはしない。そのため、受信したデータがスレッドによって受け取られる前にさらにデータが到着した場合はエラーとなる。この制限が満たされる場合には、データフローのマッチング機構を用いることができるため、前者のメッセージ通信より、オーバヘッドが少ない。

msg_type は、メッセージをマルチプレックスするもので、現在のインプリメンテーションでは、0 から 16 になっている。

3.2 細粒度パケットによるブロックメッセージ通信

ネットワークには FIFO 性があるため、複数ワードからなるメッセージの場合は、相手の PE のメモリへの書き込み、同期のためのワードによる通信で行なうことができる。その様子を図 1に示す。

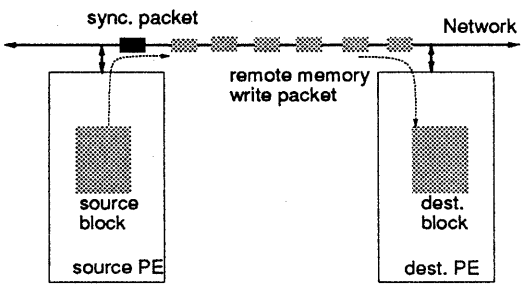

図 1: 細粒度パケットによるメッセージ通信

メッセージデータのブロック転送のために、ライブラリ関数 mem_copyout が提供されている。

```
mem_copyout(src,global_dest,nword)
```

src は、転送するブロックのローカルアドレス、globa_dest は、転送先のブロックの PE の番号を含むグローバルアドレスである。これにより、非同期に nwrod ワードの転送が行なわれる。ブロックメッセージの転送を行なう時には、mem_copyout でブロック転送を行なった後、それぞれに場合に応じて、word_send または pkt_output で同期情報を送る。

従来のメッセージパッシングと比較し、EM-4/X のメッセージパッシングの違いは以下の特徴をもつ:

- 非同期通信を基本としている。同期通信をする場合は最後に hand-shake を行なえばよい。
- 1 ワードの場合には、最適化が可能である。単に、word_send または pkt_output を用いればよい。
- また、バッファリングの処理がいらない場合には、pkt_output を用いることによって、データフローマッチング機構により最適化できる。
- 直接相手のメモリに書き込みができる。従来のメッセージ通信では、受けとり側で所定のメモリアドレスにメッセージを取り込む場合が多いが、EM-4 では、送り手側で指定して、直接書き込むことができる。これは、複雑なメッセージを送る場合、データの pack/unpack の手間を省くことができる。
- その代わり、バッファリングの semantics を用いたい時には、メモリ領域の確保をしなくてはならない。領域を確保する場合には、hand-shake を行なわなくてはならないため、オーバヘッドになることがある。
- メッセージタイプが 16 または 32 程度に限られている。これは、オペランドセグメントまで拡張可能である。

4 メッセージ通信の性能

EM-4 プロトタイプにおいて、メッセージ通信プログラムにおいて重要と思われる以下の点について、評価をおこなった。

(1) 最小パスでのメッセージ転送性能。

(2) Complete Exchange — all-to-all の通信を行なうもので、メッセージサイズとアルゴリズムを変えて行なった。

(3) Broadcast — 一つのソースからのブロードキャスト通信。2 つのアルゴリズムでおこなった。

(4) データ並列プログラムに良く用いられる reduction、scan、shift の基本操作。

(2) と (3) の評価に使ったアルゴリズムは、Ponnusamy ら [3] による CM-5 でのメッセージ通信の性能評価で用いたものを用いた。評価はプログラムしやすくするために、80PE の内、64PE を用いた時のものである。

4.1 round-trip による基本性能の測定

最小パスでのメッセージ通信を測定した。EM-4 のオメガネットワークは 5PE でループを構成しているため、このループでメッセージを round-trip し、それにかかる時間を 5 で割ったものになる。

Program	Size (words)	Time (μs)	Time/Link (μs)
packet		10.24	2.05
word		15.04	3.01
block	1	38.25	7.65
block	10	81.45	16.29
block	100	567.45	113.49

表 1: メッセージの round-trip の実行時間

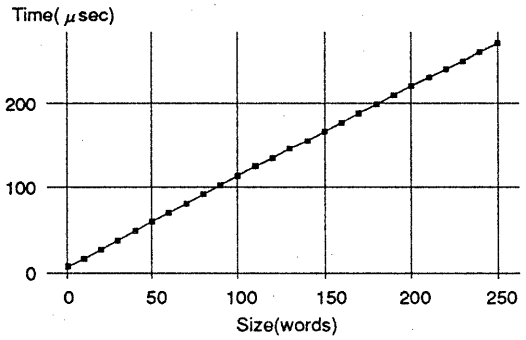

図 2: round-trip のメッセージサイズによる変化

表 1に結果を示す。packet と word はそれぞれパケット通信とワードメッセージ通信を用いて、1 ワードのみの転送をおこなった場合である。block は、同期にパケット通信を用いて、ブロック転送を行なったものである。図 2にメッセージサイズによる変化を示す。この結果から、通信にかかる時間は以下の式で計算できる。

$$t_comm = 1.07 \times n + 6.58 \ \ (in\ \mu sec.)$$

ここで、n はワード数である。4 バイト／ワードとすると、0.2675 になり、[3] に報告されている CM-5 のベストケースの半分のバンド幅になる。しかし、起動までのオーバヘッドは CM-5 の 88μsec に比べて、6$\mu sec.$ と 10 分の一になっている[1]。また、EM-4 のネットワークのバンド幅は 6.25Mpackets/sec であり、この結果は 1/6 程度になっている。これは、メッセージ転送する際のループのオーバーヘッドであり、実際、十数命令に 1 パケットの転送を行なっている。

4.2 Complete Exchange

Complete Exchange は各 PE にあるデータをすべての PE に送る all-to-all の通信操作である。この操作はマトリクスの転置、ADI integration、2 次元 FTT などに用いられる。プログラムは、各 PE に全 PE 分のバッファを作り、すべての PE のデータが格納されるまでの時間を、以下のアルゴリズムを使って実行し、測定した。付録に各アルゴリズムの概要を示す。

Linear Exchange (lex) — 各ステップで一斉に、一か所に送りだし、それを N ステップ繰り返す。他のプロセッサからのメッセージを受けとる時に、メッセージタイプを使って、multiplex することもできるが、EM-4 の場合は直接書き込み、メッセージはカウントのためにつかっている。

Pairwise Exchange (pex) — 各ステップで、2 つずつペアを組みながら交信して、データを交換していく。これを N ステップ繰り返す。

Recursive Exchange(rex) — バタフライネットワークを作って、データを交換する。$log_2 N$ ステップで済むが、従来のメッセージ通信ではデータを pack/unpack する必要がある。EM-4 では相手のバッファに直接書き込むため、この操作は必要ない。

Random Write Exchange (wex) — すべての PE で、自 PE のデータを隣の PE から順に、一斉に非同期にブロック転送し、最後に barrier 同期をつかって、同期をおこなう。これは EM-4 のリモートメモリ機能を共有メモリ的に使った方法である[2]。

[1] CM-5 では Active Message レイヤーを使えばこのレーテンシは短縮されるであろう

[2] あとに述べるように、barrier 同期終了時にすべての remote write が終了しているかの証明は、残念ながらされていない。現

Program	Size(words)	Time (μs)
lex	1	1634
lex	10	2754
lex	100	16639
pex	1	1079
pex	10	1990
pex	100	12632
rex	1	916
rex	10	1823
rex	100	12437
wex	1	458
wex	10	1018
wex	100	12672

表 2: Complete Exchange の実行時間

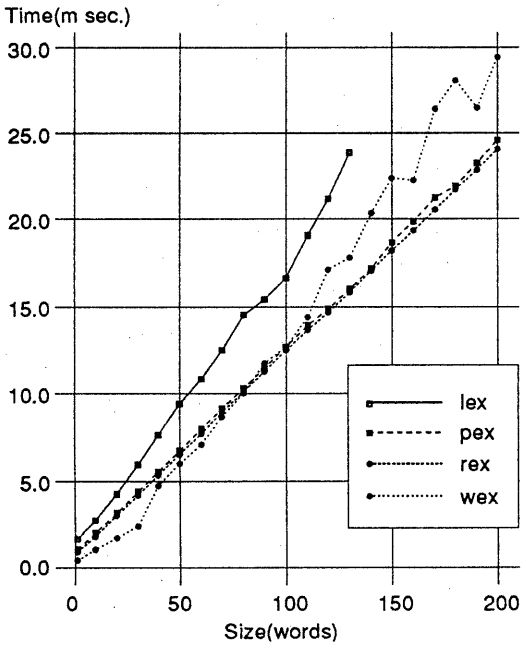

図 3: Complete Exchange のメッセージサイズによる変化

表 2に結果を示す[3]。図 3にメッセージサイズによる変化を示す。lex の場合は、メッセージが 1 つのプロセッサに集中し、性能は低下する。さらに、EM-4 はメモリ上に 4K パケットまでのパケットを保持できるが、130 ワード以上ではこのバッファが溢れ実行ができなかった。pack/unpack のオーバーヘッドがないため、pex と rex はほとんど差がなかった。wex は同期のオーバヘッドが少ないため、メッセージサイズが小さい領域では速いが、ランダム性があるため、メッセージサイズが大きくなるとネットワークの衝突を起こし易くなる。pex と lex は通信が規則だだしくおこるため、そのような性能の低下は起こらない。

4.3 Broadcast

ブロードキャストは、2 つのアルゴリズムで行なった。

Linear Broadcast (lib) — 単に、マスタが順に送りだすアルゴリズム。

Recursive Broadcast (reb) — $log_2 N$ オーダーで、tree の形にブロードキャストするアルゴリズム。

表 3に実行結果を示す。サイズが pkt である項は、パケット通信のみの時間である。図 4にメッセージサイズによる変化を示す。明らかに、reb の方がよいことがわかる。reb は、1 msec 以下であり、十分に速い。また、両アルゴリズムともネットワークの衝突は見られない。

Program	Size(words)	Time (μs)
lib	pkt	410
lib	1	864
lib	10	1453
lib	100	6171
reb	pkt	27
reb	1	65
reb	10	108
reb	100	618

表 3: Broadcast の実行時間

4.4 データ並列の基本操作

データ並列プログラムの基本操作の性能として、表 4に 64PE における reduction と send&recv(shift)、

在、検討中であるが、経験的に不都合は確認されていない。

[3]pex と rex については、パケット通信を使って同期をしている。この場合はデータの転送が均質的であり、バッファリングをしなくとも不都合はないが、安全のためにはワードメッセージ転送を使わなくてはならない。しかし、そのオーバーヘッドは十分に小さいと思われる。

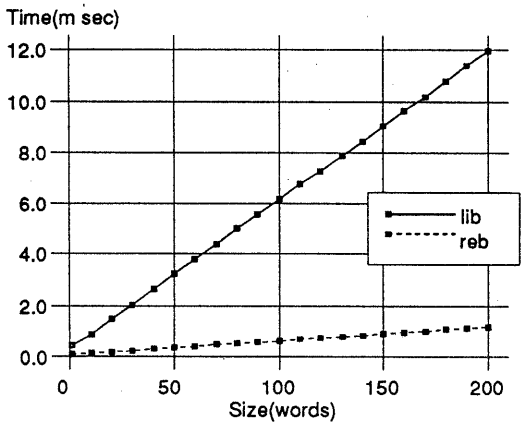

図 4: broadcast のメッセージサイズによる変化

scan の各基本操作について示す。演算はすべて整数和である。reduction はバリア同期としてつかわれており、パケット通信によって、ソフトウエア的にバタフライネットワークを作り、実現している。

Operation	Size(words)	Time (μs)
reduction		15
shift	1	8
shift	10	17
shift	100	118
scan	1	205
scan	10	218
scan	100	348

表 4: データ並列基本操作の実行時間

4.5 考察・問題点

まず、単独のリンク当たりの実効転送性能は、4Mbytes/sec 程度になるが、これは実際のハードウエアの転送容量の 1/6 程度である。しかし、複数のプロセッサが同時に通信し合う場合には、以上の転送能力が必要となる。EM-4 の場合、ネットワーク容量自体は大きいので、実際には十分間に合う。たとえ、一つの PE からの転送速度がネットワーク容量一杯に転送できたとしても、複数の PE からの転送により、実際の性能は低下してしまうであろう。

次に、メッセージデータのブロック転送では明示的に数ワード（現在は 10 ワード）ごとにスレッドを中断している。これは、現在の EM-4 ではブロックを中断しなければ、ネットワークから PE へのパケットを取り込むことができないからである。これにより、ネットワークにパケットが滞留し、ネットワークの性能を低下させてしまう要因になる。また、取り込むことができたとしても、オンチップのパケットバッファからメモリに書き込み（パケットオーバーフロー）が起き、性能の低下となる。スレッド中断によるオーバーヘッドは 10% 程度である。

EM-X ではこの点を改良し、メッセージのブロック転送などで多く使われるリモートメモリの書き込みのパケットは命令実行パイプラインでなく、パケットのバッファ部でバイパスして実行させることができる。また、ネットワークからパケットを取り込む動作の優先度を命令実行よりも高くし、ネットワークへのプロセッサの実行状態からの影響を低減している。

また、先に指摘したとおり、ランダムに任意のプロセッサに転送する場合にはリモートメモリ書き込みを行なった後、barrier で同期する方法が有効である。しかし、現在の barrier はバタフライネットワーク状に全プロセッサの同期をとるもので、ネットワークの全リンクは通過するものの、すべての非同期の書き込みパケットが実行されているか証明されていない。少なくとも、まだそのような事態は観測されていないが、論理的に保証することが必要であると考えている。

最後に、本方式はプロセッサの Point-to-Point FIFO の性質を非常に強く、利用した方式である。将来、大規模なマルチプロセッサシステムでは adaptive な routing が必要となるとおもわれるが、この性質を保持するためには到着時の re-ordering の機構が必要となる。この性質を用いないとすると、カウンタによる同期をする方法が考えられる。

5 結論

本稿では、EM-4/X での RICA による細粒度パケットによるメッセージ通信の実現とその性能について述べた。EM-4/X は、細粒度パケットによる通信が効率的に処理できるように、パケット処理がパイプラインに融合されるように設計されている。その結果、従来のメッセージ通信型プロセッサに比べて、極めて低レーテンシの通信が可能になるだけでなく、相手のメモリに対し直接書き込めることにより、柔軟なプログラミングが可能になる。さらに、細粒度のパケットであっても、1PE からのブロック転送における実効転送バンド幅はプログラムのオー

バヘッドにより 4MBytes/sec と、CM-5 などと比べれば低いが、ネットワークの容量自体は大きいため、all-to-all 通信、broadcast 通信など複数のプロセッサが同時に通信を行なう場合には十分が性能が得られていることがわかった。

我々は現在、EM-4 の後継機である EM-X を開発中である。EM-X では書き込みパケットがパケットバッファ部でバイパスして処理されるため、さらに本方式の効率化が見込めることがわかっている。以上から、EM-X が従来のメッセージ通信型のプログラムにおいても十分な性能を達成できる見込みを得た。

謝辞　本研究を遂行するにあたり御指導、御討論いただいた太田情報アーキテクチャ部長、ならびに計算機方式研究室の同僚諸氏に感謝いたします。

参考文献

[1] Y. Kodama, Y. Koumuara, M. Sato, H. Sakane, S. Sakai, and Y. Yamaguchi. EMC-Y: Parallel Processing Element Optimizing Communication and Computation. In *Proc. of 1993 ACM International Conference on Supercomputing*, pages 167–174, 1993.

[2] D. Kranz, K. Johnson, A. Agarwal, J. Kubiatowicz, and B-H. Lim. Integrating Message-Passing and Shared Memory: Eearly Experience. In *Proc. of 4th ACM PPoPP*, pages 54–63, 1993.

[3] R. Ponnusamy, A. Choudhary, and G. Fox. Commuincation Overhead on CM5: A Experimental Performance Evaluation. In *Proc. of Frontiers '92*, pages 108–115, 1992.

[4] S. Sakai, Y. Yamaguchi, K. Hiraki, Y. Kodama, and T. Yuba. An Architecture of a Dataflow Single Chip Processor. In *Proc. of the 16th Annual International Symposium on Computer Architecture*, pages 46–53, June 1989.

[5] M. Sato, Y. Kodama, S. Sakai, Y. Yamaguchi, and Y. Koumuara. Thread-based Programming for the EM-4 Hybrid Dataflow Machine. In *Proc. of of the 19th Annual International Symposium on Computer Architecture*, pages 146–155, May 1992.

[6] 佐藤、児玉、坂井、山口. 並列計算機 EM-4 上におけるマルチスレッドプログラミング モデル. In 信学技報 *CPSY91-36*, pages 15–22, 1991.

[7] 佐藤、児玉、坂井、山口. 高並列計算機 EM-4 上での共有メモリベンチマークの実行 — 並列スタンダードセルルータ LocusRoute の検討 —. In 信学技報 *CPSY92-61,FTS92-34*, pages 49–56, 1992.

[8] 児玉、甲村、佐藤、坂井、山口. 高並列処理向け要素プロセッサ EMC-Y の設計. In *JSPP '92* 論文集, pages 329–336, 1992.

[9] 児玉、坂井、山口. データ駆動型シングルチッププロセッサ EMC-R の動作原理と実装. 情報処理学会論文誌, 32(7), 1991.

付録：Complete Exchange のアルゴリズム

Linear Exchange —

```
for(j = 0; j < N_PE; j++){
    if(me == j){
        for(k = 0; k < N_PE; k++)
                if(me != k) receive(k);
    } else send(j);
}
```

Pairwise Exchange —

```
for(j = 1; j < N_PE; j++){
    node = j ^ me;
    if(me < node){
        receive(node);
send(node);
    } else {
        send(node);
        receive(node);
    }
}
```

Recursive Exchange —

```
for(i = 0 ; i < log(N_PE); i++){
    k = N_PE/(2**i);
    if((me%k) < k/2) node = me + k/2;
    else  node = me - k/2;
    if(me < node){
        ... pack message ...
        receive(node);
        send(node);
        ... unpack message ...
    } else {
        receive(node);
        ... unpack message ...
        ... pack message ...
        send(node);
       }
}
```