

## 並列計算機 EM-4 における 分散データ構造を用いたマルチスレッドプログラミング

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)

データ駆動計算機 EM-4 は、逐次実行を基本とするマルチスレッドプログラミングモデルでは分散メモリのマルチプロセッサと見ることができる。本稿では、スレッド間で共有する分散データ構造について述べる。各スレッドは、分散データ構造にアクセスすることで、通信・協調を行なうことができる。分散データ構造として、I structure を拡張し、待ち合わせるスレッドとデータの双方向のキューを持つ Q structure を提案する。さらに、これらの分散データ構造が Linda の Generative 通信の通信の性質を持つことを示す。EM-4 はそのデータ駆動計算機としての特徴から、遠隔操作の起動が高速に行なうことができるため、分散データ構造を効率的に実現できる。

## Distributed Data Structure in Multi-thread Programming Model for a Highly Parallel Dataflow Machine EM-4

SATO Mitsuhsa

KODAMA Yuetsu SAKAI Shuichi YAMAGUTI Yoshinori

Electrotechnical Laboratory

1-1-4 Umezono, Tsukuba, Ibalaki 305, Japan

From the viewpoint of an executing sequential thread, the hybrid dataflow machine EM-4 can be thought of as a distributed-memory processor. In this paper, we describe some distributed data structure shared among threads in different PEs; threads may communicate and coordinate by leaving data in shared data. A new distributed data structure called Q structure is introduced which can be used as shared queue. We show that the distributed data structure has properties of generative communication proposed in Linda. The inherited nature from dataflow architecture allows very efficient remote operation invocation to access distributed data in different PEs.

## 1 はじめに

EM-4 は、強連結枝モデルに基づくデータ駆動計算機である。EM-4 では、強連結枝で結ばれたコードを一つのブロックとし同期をコードブロックの先頭に制限して、発火されたコードブロックをレジスタベースの RISC アーキテクチャで効率に逐次実行することで強連結枝モデルを実現している。このコードブロックを強連結枝ブロックと呼ぶ。強連結枝ブロック内は逐次実行されているので、この強連結枝ブロックを連鎖してパケットで実効することによって、任意の長さの命令列でも逐次実行することが可能となる。我々は、EM-4 の要素プロセッサ EMC-R の逐次 C コンパイラを開発した。これによって、各要素プロセッサ上で、C 言語で記述された逐次プログラムを実行することができる。この逐次実行を基本とするプログラミングでは、各要素プロセッサごとに一つあるいは複数の逐次実行 (thread) を記述し、これらの間で同期を行って並列計算を行う。このようなプログラミングモデルを、データ駆動計算機本来のデータフロー計算モデルに対して、ここではマルチスレッドプログラミングモデルと呼ぶことにする。このプログラミングモデルでは、EM-4 はそのデータ駆動計算機としての特徴から、スレッドの生成、同期が非常に高速な分散メモリのマルチプロセッサとして見る事ができる。

分散メモリプロセッサ上のスレッドの同期の方法の一つとして、メッセージ通信による同期が考えられる。我々は、[8]において、EM-4でのメッセージ通信のための機構が効率的に実現されることを示した。

本稿では、EM-4でのマルチスレッドプログラミングモデルにおける分散データ構造のための操作とその実現について述べる。分散データ構造とは、多くのスレッド (あるいはプロセス) から同時に直接アクセスできるデータ構造をいう。本来、データフロー言語で導入されている Istructure メモリ [5] も分散データ構造の一つであるが、本稿では Istructure メモリを拡張し、待ち合わせるスレッドとデータの双方向のキュー構造をもつ Q structure を提案する。さらに、キーを加えた Q structure は Linda で提案された generative 通信 [3] の性質を持つことを示す。これらの分散データ構造は、Linda の tuple 空間による分散データ構造の subset であるが、同期に必要な機能に従って使い分けることによって、必要な分散データ構造の効率的な実現が可能になる。2 章では、EM-4 の概要について説明し、3 章で分散データ構造について述べる。4 章では、分散デ

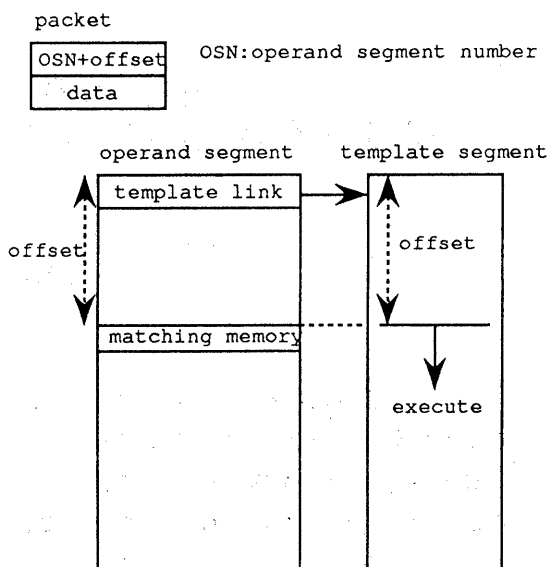


図 1: 関数の実行とセグメント

タ構造を用いたプログラミングについて考察を行う。

## 2 EM-4 でのマルチスレッド実行

スレッド実行の方法を中心に EM-4 について概要を説明する。詳しくは、[6] [7] を参照していただきたい。

### 2.1 パケットと強連結枝ブロックの実行

EM-4 での強連結枝ブロック (単にブロックと呼ぶ) は、レジスタベースの RISC アーキテクチャで逐次実行される命令コード列である。PE でのブロックの実行は、PE に対してパケットを送信することで起動する。関数を実行する場合は、オペランドセグメントと呼ぶ関数フレームを割り当て、オペランドセグメントの先頭に関数の命令コードのアドレスをリンクしておく。関数の命令コードのセグメントをテンプレートセグメントと呼ぶ。

パケットには、以下のデータが含まれている。

- 起動する関数のフレーム、すなわちオペランドセグメントの番号。
- 起動される強連結ブロックのテンプレートセグメント内オフセット。

- マッチングの属性。1 オペランド入力、2 オペランド入力など。
- 強連結ブロックの先頭の命令に対するオペランドデータ。
- バケットタイプ (通常 0)

バケットで指定されているオペランドセグメントにリンクされているテンプレートセグメントにオフセットを加算して、先頭命令アドレスを計算する。マッチングの属性が、1 オペランドの場合は、単にブロックの実行を起動する。スレッドの実行には、この1オペランドの属性のバケットを用いて、ブロックを逐次に連鎖させて、任意の長さの命令列を逐次実行することができる。図1に関数の実行とそれぞれのセグメントの関係について示す。

ブロックが実行される際には、バケットのオペランドデータは所定のレジスタにロードされている。オペランドセグメントレジスタには、そのバケットで起動されたオペランドセグメントのアドレスが入る。EM-4は、16個のレジスタを持ち、そのうち10レジスタは汎用レジスタとして使うことができる。ただし、ブロックの実行の終了後はレジスタの内容が保証されないが、PE内のメモリに対するアクセス命令によって、ローカルメモリに退避しておくことができる。

## 2.2 EMC-Rの逐次Cコンパイラとスレッド生成

Cコンパイラでは、関数のインスタンスごとにオペランドセグメントを割り当てるようにコンパイルし、このオペランドセグメントを引数、ローカル変数やレジスタの退避領域に使っている。関数呼び出しにはセグメントを割り当て、引数を書き込み、バケットによって起動する。バケットのデータとして、呼び出し側のcontinuationを渡し、呼び出し側の関数はブロックの実行を中断する。関数のリターン時にcontinuationに対して、リターン値をバケットで送り、呼び出し側の関数を再開させる。

データ参照はすべて、ローカルメモリに対しておこなわれる。データフローモデルとは異なり、スレッド実行では逐次実行されるため、メモリに対する副作用による計算が可能である。

スレッドを生成するには、オペランドセグメントを割り当て、現在実行中のスレッドを中断せずに、そのオペランドセグメントを起動する。他のPEにスレッドを生成するには、他のPEのオペランドセ

グメントに対して同様のことを行なえばよい。これらの操作は、関数forkで行なうことができる。

```
fork(PE_addr, func, nargs, arg1, ..., argn)
```

PE\_addrは、スレッドを生成するPE番号を含むアドレスである。この関数によって、PE\_addrで指定されたPEに、関数funcから開始するスレッドを生成する。この関数は、値を返さない。スレッドは、関数funcが終了した時点で、終了する。

また、関数remote\_callは、forkと同じ引数を取り、remote procedure callを行なうものである。forkと同様に、指定されたPEにおいて任意の関数を実行するが、指定された関数が終了するまで、呼び出し側のスレッドは中断され、リターン値を受け取り実行を再開する。

PEに到着したバケットは、ハードウェアのFIFOキューに保持され、実行されているブロックが終了すると、キューからバケットを取り出され、対応するブロックの実行が行なわれる。スレッドから見れば、バケットは中断しているスレッドの内部状態であり、バケットキューによって、ハードウェアによってスケジューリングされていると見ることができる。

## 2.3 特殊バケットとリモートメモリアクセス

0以外のバケットタイプをもつバケットを特殊バケットという。このバケットが受信された場合には、バケットタイプに対応するシステムルーチンが実行される。この場合、バケットタイプでルーチンを指定するため、データ部だけでなく、アドレス部も引数として用いることができる。そのルーチンには、以下のものがある：

- バケットによるPEのメモリの書き込み、読みだし
- バケットによるPEのオペランドセグメントなどの資源の割り当て

他のPEのデータをアクセスする場合には、PE番号を含むデータのアドレス(グローバルアドレス)をバケットのアドレス部で指定し、その操作はバケットタイプで指定する。ブロックの実行は途中で中断することはないため、オペランドセグメントなどシステム資源に関する管理については、1つのブロック内で行なうことで排他制御を行なうことができる。

Cプログラムからはmem\_read/mem\_writeを用いて、グローバルアドレスg\_addrで指定されたメモリに対してアクセスすることができる。

```
mem_write(g_addr, word)
word = mem_read(g_addr)
```

これらの関数は、コンパイル時に inline 展開され、パケットを送信する数命令にコンパイルされる。EM-4 は、分散メモリのマルチプロセッサであるが、パケットにより他の PE での実行を高速に起動することによって、グローバルアドレス空間でのリモートのメモリアクセスを効率的に行なうことができる。

### 3 マルチスレッドプログラミングモデルにおける分散データ構造

マルチスレッドプログラミングモデルでは、各スレッドは個々の PE のローカルメモリ上で実行され、EM-4 は分散メモリのマルチプロセッサとして見ることができる。

分散環境での計算では、何らかの同期機構を持たなくてはならない。現在、多くの分散メモリシステムではメッセージ通信を用いているが、これは共有されるデータ構造をプロセス（スレッド）の構造に吸収しようというものである。分散データ構造は、スレッドとそれらが共有するデータオブジェクトを区別し、スレッドはそのデータオブジェクトをアクセスすることによって、通信・協調を行なおうというものである。

#### 3.1 分散データ構造に関する操作

##### 3.1.1 I structure メモリ

I-structure メモリは、グローバルアドレスで指定されたメモリのみをバッファとするデータ構造である。

```
I_write(g_addr, word)
word = I_read(g_addr)
```

I\_read は、I\_write で書き込みが行われるまで、スレッドはブロックされる。I\_write は、I\_read でブロックされているスレッドを再開する。もし、前に書き込んだデータが読まれる前にさらに書き込みがあった場合、あるいは複数のスレッドが同じメモリを同時に読む場合にはエラーとなる。

##### 3.1.2 Q structure メモリ

Q structure は、指定されたグローバルアドレスにキュー構造をもつデータ構造である。

```
Q_out(g_addr, word)
word = Q_in(g_addr)
```

Q\_in は、Q\_out によってデータが書き込まれるまでブロックされる。Q\_in が複数のスレッドによって同じ g\_addr に行われたときには、それらのスレッドはすべてブロックされ、Q\_out によってデータが書き込まれるごとに 1 つずつ再開される。同時に Q\_out によって、書き込まれたデータは g\_addr で指定されるキューに入れられる。Q\_in はこのキューから、データを取り出す。

```
word = Q_read(g_addr)
```

Q\_read は、g\_addr からデータを取り除かずにデータを読みだす。データが書き込まれていない場合は Q\_in と同様に、ブロックする。

これと似たデータ構造として、M structure [4] があるが、M structure の場合は同時に複数の書き込みを許していない。

##### 3.1.3 キー付き Q structure メモリ

Q\_in\_with\_key と Q\_out\_with\_key は、Q\_in と Q\_out にキューのデータに関して、キーによるデータ選択の機能を付け加えたものである。

```
Q_out_with_key(g_addr, key, word)
word = Q_in_with_key(g_addr, key)
word = Q_read_with_key(g_addr, key)
```

Q\_in\_with\_key は、現在書き込まれているデータの中に指定された key を持つデータを選択、取り出し、その値を返す。該当するデータがない場合は Q\_out\_with\_key で指定された key を持つデータが書き込まれるまでブロックされる。Q\_out\_with\_key は、指定された key を持つデータを書き込む。

Q\_read\_with\_key は、Q\_read と同様、データを取り除かない。

```
Q_out_with_any(g_addr, word)
word = Q_in_with_any(g_addr, &key)
word = Q_read_with_any(g_addr, &key)
```

これらは、任意のキー付きのデータに対して、同様の操作を行なう。Q\_in\_with\_any と Q\_read\_with\_any は、任意のキーのデータを取り出し、その値と共にキーの値を返す。データがなければ、ブロックする。Q\_out\_with\_any は、どの key でもマッチするデータを書き込む。

### 3.2 分散データ構造に対するスレッドの生成

以下の関数は、g\_addr にある分散データ構造に関して、指定された関数 func のリターン値を書

き込むスレッドを生成するものである。スレッドは、データ構造がある PE で実行され、書き込みが終了した時点で消滅する。

```
I_fork(g_addr, func, nargs, arg1, ...)
Q_fork(g_addr, func, nargs, arg1, ...)
Q_fork_with_key(g_addr, key, func, ...)
```

### 3.3 EM-4 の分散データ構造の実現

EM-4 は、データ駆動計算機としての機構から、次のような特質をもつ：

- 他のプロセッサに対して、バケットにより遠隔操作を高速に起動することができる。
- スレッドの生成が高速に行なうことができる。
- 停止しているスレッドは実行すべきブロックを示す continuation という形で保存される。実行を再開するには、その continuation に対してバケットを送ればよい。

EM-4 では、遠隔操作の実現に関して、特殊バケットを用いる方法とルーチンを起動する方法がある。リモートメモリアクセスや I structure など簡単な操作は、特殊バケットの機能を使ってインプリメントされている。例えば、I\_read は以下の動作を行なう：

1. アドレス部にはアクセスするメモリのグローバルアドレス、データ部には現在のスレッドの continuation をもつ I structure の読みだしの特殊バケットを送り出す。現在のスレッドは、ブロックを終了し、停止する。
2. 特殊バケットルーチンでは、アドレス部で指定されているメモリを検査し、そこにデータがすでにセットされていれば、そのデータをバケットのデータ部にある continuation に対してデータを送る。その際、もとのメモリはクリアしておく。
3. もし、まだメモリにデータがない場合はそのメモリにバケットのデータ部にある continuation を書き込んでおく。

一方、I\_write は：

1. アドレス部にはアクセスするメモリのグローバルアドレス、データ部にはデータを持つ、バケットタイプが I structure の書き込みのバケットを送り出す。現在のスレッドは、続行される。

2. 特殊バケットルーチンでは、アドレス部で指定されているメモリを読みだし、そこに continuation が入っていれば、バケットのデータをその continuation に対して送り出す。その際、メモリの内容をクリアしておく。

3. もし、メモリに continuation がない場合はそのメモリにデータを書き込んでおく。

Q structure に対する操作は、キューに伴うメモリ管理が伴うので特殊バケットを使わずに、グローバルアドレスで示されるデータがある PE に対して、操作に対応するルーチンを起動することで行なう。例えば、Q\_in は：

1. アクセスするデータ構造のグローバルアドレスで示される PE に対して、Q structure からの読み込みのルーチンにリンクしたオペランドセグメントを確保する。この操作は、特殊バケットルーチンで行なわれる。
2. 確保したオペランドセグメントに対して、アクセスするデータのアドレスと現在の continuation を送り、呼びだし側のスレッドはブロックを終了し、停止する。
3. ルーチン側では、アドレスで示されるキューを検査し、データがある場合はそのデータを取り出して continuation に送り、ない場合はキューにその continuation を入れておく。その後、ルーチンのオペランドセグメントを解放し、終了する。

Q\_out では：

1. Q\_in と同様に、データ構造のある PE に Q structure の書き込みのルーチンをリンクしたオペランドセグメントを確保し、アクセスするデータのアドレスと書き込むデータを送る。呼び出し側のスレッドは中断しない。
2. ルーチン側では、指定されたキューに continuation があれば、そのデータをその continuation に送りだし、そうでなければ、データをキューに入れておく。その後、ルーチンのオペランドセグメントを解放し、終了する。

これらの操作で起動されるルーチンは、引数の待ち合わせをデータフロー的に行なうことによって、効率的に実行される。

スレッドを生成する fork では、生成されたスレッドで実行される関数のリターンアドレスは、通

常単に何もしないで終了するブロックになっているが、I\_fork、Q\_forkなどの場合は、データ構造にリターン値を書き込むルーチンが設定される。このスレッドは、データ構造のあるPEと同一であるので、書き込み操作はローカルに行なわれる。

## 4 分散データ構造を用いたプログラミング

### 4.1 Generative 通信の性質

これまで提案した分散データ構造は、Lindaのtuple空間のsubsetであり、Lindaの特徴であるGenerative 通信 [3] の性質を満たす：

**Space uncoupling** — 分散データがアクセスするプロセスの位置に依存しない。Q structureでは、PEに渡るグローバルアドレスを用いてアクセスを行なうため、データにアクセスするスレッドがどのPEで実行されていても、グローバルアドレス空間の任意のデータに対して書き込みあるいは読み込みができる。この性質は送り手側が受け取り手側を陽に知らなくともよいことを意味する。例えば、message passingでは送り先のプロセスが明示されていないとはならない。

**Time uncoupling** — 送信されたデータが送信したプロセスとは無関係に存在する。Q structureでは、Q\_outではスレッドがブロックされず（非同期通信）、この性質をもつ。CSPでは送り側のプロセスはそのデータが受け取られるまで、ブロックされており、CSPのchannelはこの性質を満たしてはいない。

**Distributed Sharing** — データは任意のPEのプロセスから共有され、そのデータに対する操作は排他的に行なわれる。Q structureでも、任意のPEのスレッドの操作に関して、同様に排他的に行なわれる。いくつかの言語に見られるように、共有変数をプロセスやモジュールでインプリメントする必要はない。

**Support for continuation passing** — ここでの“continuation passing”とは、あるプロセスがデータを他のプロセスに送り、そのリブライを待つ場合、リブライを待つ名前を値として送りだし、最終的にリブライするプロセスがその値を名前して用いることができることをいう。Q structureでは、名前とはグローバルアドレスそのものであるから、値として扱うことができる。

**Structured naming** — 分散データ構造の中から、あるデータを選択し、そのデータから関係す

る分散データ構造を生成したり、取り出すことで構造的に分散データ構造を構成できる。Q structureでは、キーを用いることによって、データを選択できるが、パターンマッチングによる選択は行なっていない。

### 4.2 分散データ構造の例

[2]で示されている分散並列プログラミングの基本的な問題について、分散データ構造を用いたプログラミング例を示す。

#### 4.2.1 remote procedure

メッセージ通信を含む基本的な通信機構は、Q structureを用いて実現することができる。例えば、remote procedure callは以下のように実現することができる：

```
struct {
    g_addr_t ret_addr;
    word_t args[N_ARGS]
} x,y;

g_addr_t me;
g_addr_t proc_port;

/* caller */
x.ret_addr = me;
x.args[0] = ...; /* set args */
Q_out_block(proc_port,&x,sizeof(x));
result = I_read(me);

/* callee */
Q_in_block(proc_port,&y,sizeof(y));
... body of "proc" ...
I_write(y.ret_addr,result);
```

ここで、proc\_portとmeはグローバルアドレスであり、あらかじめ設定されているものとする。Q\_out\_block/Q\_in\_blockは、ブロック単位でデータを転送する操作である。この場合、リターン値を受け取るのに一つの値しか待ち合わせしないため、I structureを用いることによって簡単化を行なっている。

この例は、そのprocedure自身が別のスレッドで実行されるもので、しばしば“active procedure”と呼ばれる構造である。したがって、reentrantでなく、再帰呼び出しはできない。

#### 4.2.2 shared variable

Q structureに関する操作は排他的に行なわれるため、shared variableを容易に実現することが

できる。

shared variableを更新する場合は：

```
Q_in(var_addr); /* discard old value */
Q_out(var_addr,new_value);
```

ここで、var\_addrはshared variableのグローバルアドレスである。shared variableを読み込む場合は、Q\_readで読むことができる。

また、Q structureでは、Pオペレーションとして、Q\_inを使い、Vオペレーションとして、Q\_outを用いることによって、セマフォを実現できる。初期値がNのセマフォを作るには、初期化時にQ\_outをN回繰り返せばよい。これによって、critical sectionは以下の様に実現できる：

```
g_addr_t sem;
Q_out(sem); /* initialize */
....
Q_in(sem); /* P operation */
... body of critical section ...
Q_out(sem); /* V operation */
```

グローバルアドレス変数semは、各PEにおいて、あらかじめ所定のPEのアドレスに初期化しておかななくてはならない。どのPEで実行されても、semで示されたQ structureに対して行なわれる。

#### 4.2.3 bag 構造

並列プログラミングでは、複数の同じプログラムを実行するスレッドをワーカーとして、それぞれのワーカーが分散データ構造に対して必要なデータを取り出し、そのデータに関して計算させるのが便利場合がある。

最も単純な場合を考えてみる。各PEにおいて：

```
g_addr_t job_bag;

j = Q_in(job_bag)
... execute job 'j' ...
```

あるjob kを実行させるには、Q\_outで、job kをこのQ structureに入ればよい。job\_bagで待っているスレッドがあれば、このスレッドがjob kをとりだし実行する。

このようなbag構造は、Space Uncouplingの性質によるものである。データを書き込む方は、誰がそのデータを必要とするかを気にする必要はなく、またデータを必要とする方はデータが分散データ構造に置かれており、それを必要とするどのスレッドでも得られるようになる。

#### 4.2.4 delayed statement

指定された時間の間、あるスレッドを停止させておくプログラムを考える。クロックごとに起動されるスレッドは、キー付きQ structureを使って、時間をキーとして、時間を更新する：

```
g_addr_t clock;

Q_in_with_any(clock,&now);
Q_out_with_key(clock,now+1);
```

現在の時間を読むには：

```
Q_read_with_any(clock,&now);
```

また、ある時間dだけスレッドを停止する関数delayは、以下のように書くことができる：

```
delay(d){ int now;
  Q_in_read_any(clock,&now);
  Q_read_with_key(clock,now+d);
}
```

### 4.3 Linda との比較

Lindaでのtuple空間は疑似的な共有メモリであるのに対して、ここで提案した分散データ構造は、構造をPEにまたがるグローバルアドレス空間に直接マッピングしている。これによって、ユーザが通信と負荷の分散を意識してプログラムすることになる。

また、Lindaではtupleデータのマッチングによって、いろいろな構造を作れるようにしているが、我々はその機能を制限し、単純化している。ユーザは、必要なデータ操作を使い分けることによってプログラムを効率化できる。

また、分散環境では、Lindaのtuple空間を実現するためには、いくつかのソフトウェアとハードウェア[1]による実現方式が提案されているが、多くの場合、tupleデータをcacheして効率化を行っているため、複雑なプロトコルが必要となっている。EM-4における分散データ構造の実現では、通信とそれをつかった遠隔操作が十分に高速であることから、分散データに対して直接アクセスしている。

### 4.4 データフロー計算モデルとの比較

データフロー計算モデルでの問題点の一つは、配列などのデータ構造と履歴を利用する計算である。データフロー計算モデルは関数的であり、配列の要素の更新は新たな配列を用意することでおこなわれ

る。I structureはその様な関数的な枠組の中で、配列要素に関する同期構造として提案された。しかし、ヒストグラム計算など実際の計算では履歴を必要となることがある。そのため、Id [4]ではM structureを導入し、複数のトークンによる同時アクセスを許している。複数データの同時書き込みはないため、タグ・データ駆動計算機のマッチングのハードウェアでの実現が可能になる利点がある。

これまで、データ駆動計算機ではデータフローグラフを直接記述するデータフロー言語が使われてきた。これによって、データ駆動計算機では命令レベルから関数レベルまでの幅広い並列性を引き出すことができる。しかし、それらの並列性はデータ駆動的にimplicitに取り出されるため、実際の計算機では、それらの並列性の制御と負荷分散が問題になる。分散データ構造を用いたスレッドプログラミングでは、並列性をユーザは陽に制御・分散できる。また、分散メモリシステムではデータの局所性を考慮したプログラミングは効率化の基本であり、本質的に静的なデータ構造をもたないデータフロー計算モデルはこのことについての枠組がない。

## 5 結論

本稿では、EM-4におけるマルチスレッドプログラミングモデルとその分散データ構造について述べた。分散データ構造は、スレッド間で共有するデータオブジェクトであり、スレッドはそのデータオブジェクトにアクセスすることによって、通信・協調を行なう。本稿で提案したQ structureはLindaのtuple空間のsubsetであり、generative通信の性質を持つ。分散データ構造を用いることによって、柔軟な並列構造が可能になる。

EM-4では、分散データ構造は全PEメモリ空間に直接マッピングすること実現され、データに対する操作はデータのあるPEでその操作を起動することで行なう。EM-4は、そのデータ駆動計算機としての性質から、遠隔操作の起動が高速に行なうことができるため、分散データ構造を効率的に実現できる。

EM-4はマルチスレッドプログラミングモデルから見れば、スレッドを生成するのが非常に高速な分散メモリシステムとしてみる事ができる。分散環境では局所性を考慮したプログラミングは基本であり、共有される分散データ構造とローカルなデータ構造を区別し、分散データ構造とスレッドの配置を決めることによって、ユーザは通信と負荷の分散を考慮してプログラムを効率化することができる。

## 謝辞

本研究を遂行するにあたり御指導、御討論いただいた弓場情報アーキテクチャ部長、島田計算機方式研究室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] S. Ahuja, N. Carriero, D. Gelernter and V. Krishnaswamy, "Matching Language and Hardware for Parallel Computation in the Linda Machine", *IEEE Trans. on Computers*, Vol. 37, No. 8, Aug. 1988, pp. 921-929.
- [2] N. Carriero and David Gelernter, "How to Write Parallel Programs: A guide to the Perplexed", *ACM Computing Surveys*, Vol. 21, No. 3, 1989, pp. 323-357.
- [3] David Gelernter, "Generative Communication in Linda", *ACM Trans. on. Prog. Lang. and Sys.*, Vol. 7, No. 1, 1985, pp. 80-112.
- [4] R. S. Nikhil and Arivid, "ID Language Reference Manual", *Computation Structures Group Memo 284-2*, MIT, Jul. 1991.
- [5] R. S. Nikhil and K. K. Pingali, "I-Structure: Data Structures for Parallel Computing", *ACM Trans. on Prog. Lang. and Sys.*, Vol. 11, No. 4, Oct. 1989, pp. 598-639.
- [6] 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, pp 849-858, 1991.
- [7] S. Sakai, Y. Yamaguti, K. Hiraki, Y. Kodama and T. Yuba, "An Architecture of a Dataflow Single Chip Processor", *Proc. of the 16th Annual International Symposium on Computer Architecture*, pp. 46-53, June 1989.
- [8] 佐藤、山口、児玉、並列計算機 EM-4 におけるマルチスレッドプログラミングモデル、電子情報通信学会コンピュータシステム研究会、CPSY 91-36, 1991, pp. 15-22.