並列計算機 EM-4 の並列プログラミング言語 EM-C

佐藤三久[†] 児玉祐悦[†]
坂井修一[††] 山口喜教[†]

EM-4 は，高速な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．EM-4 の並列プログラミングのために C 言語を拡張した EM-C を開発した．EM-C では EM-4 のデータ駆動機構をプロセッサ間において高速なスレッド起動，同期の機構として用いている．さらに，データ駆動機構による簡便なリモートメモリアクセス機能を利用して，EM-C では各プロセッサのローカルメモリを分散グローバルアドレス空間として，データ分散・参照をできるようにし，共有メモリ並列プログラムを実行可能にしている．リモートメモリアクセスやリモート操作に関するレーテンシを隠すためにマルチスレッドプログラミングのための並列構文を提供し，EM-C コンパイラはデータフロー待ち合わせ機構を用いた効率的なコードを生成する．

EM-C: A Parallel Programming Language for the EM-4 Multiprocessor

Mistuhisa Sato,[†] Yuetsu Kodama,[†] Shuichi Sakai[††]
and Yoshinori Yamaguchi[†]

We present a parallel programming language, EM-C, designed for the EM-4 multiprocessor whose dataflow architecture efficiently integrates communication into computation. EM-C allows a programmer to use dataflow mechanism as a set of basic operations for fast thread creation and synchronization. We introduce the notion of a *distributed global address space* to distribute data structures and access them using remote memory operation by the dataflow mechanism. New parallel constructs are provided to support for executing threads dependent on the data distribution and exploiting relatively coarse-grain parallelism to tolerate the data-dependent remote operation latency.

1. はじめに

EM-4[4]のデータ駆動機構は，データフローパケットによる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-C では，データ駆動機構を逐次実行を行なうスレッドの起動あるいは同期の機構として用いる．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]．このモデルでは，EM-4 はプロセッサ間でのスレッド生成やリモートプロシージャコールが非常に早いプロセッサと見ることができる．

† 電子技術総合研究所
Electrotechnical Laboratory
†† RWC つくば研究センタ
RWC Tsukuba Research Center

EM-4 では，このデータ駆動機構をデータフロー実行だけでなく，遠隔のメモリの読みだし・書き込みにも拡張している．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，他のプロセッサのメモリに対してアクセスすることが可能である．

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いる．すべてのプロセッサのローカルメモリをアドレスづけできる空間であり，これを分散グローバルアドレス空間（distributed global address space）と呼んでいる．プログラマは，この空間上にデータを分散配置し，共有メモリとして直接アクセスすることができる．従来のメモリ分散型マルチプロセッサではノードごとに一つあるは複数の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ため，プログラムを 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い．EM-4では分散メモリでは困難だった共有メモリ並列プログラムを実現することができる．

ローカルメモリとリモートのメモリを使い分けることは，EM-4に限らず従来のマルチプロセッサでも，性能を得るためのキーとなる．プログラマは参照が局所的になるように工夫することが必要となる．スレッドの起動が高速なEM-4では，あるスレッドの計算に必要なデータをデータを通信によって移動するよりも，スレッドをデータのある場所に制御する方が効率的な場合がある．EM-Cではデータあるプロセッサにスレッドを制御する *where* 構文を導入した．これは，リモートプロシジャーコールに似ているが，Cの構文して埋め込まれており，通常のローカルコールのほんの数倍の程度のオーバヘッドで実行できる．また，配列など分散グローバルアドレス空間に分散配置されたデータ構造に対しては各プロセッサに同一なスレッドを生成する *everywhere* 構文を使って，データ並列操作などを容易にプログラミングできる．さらに，EM-Cでは，他のプロセッサに対するリモートメモリアクセスやリモートオペレーションのレーテンシをプロセッサ内の並列性で隠蔽し効率化するために，粗粒度の並列性を記述する *forkwith* 構文を導入した．

EM-Cは，EM-4 nativeな言語として，プログラマに対して，並列プログラムの構造的な記述を提供することを目的としている．我々はすでに，分散メモリ型のプロセッサとして，EM-Cにおいてメッセージ通信のための機構を提供しており[5]，また，EM-4のデータ並列言語コンパイラのターゲット言語としても用いられている[6]．2章において，EM-4でのプログラムのスレッドへのコンパイルの方法と実行時の構造について述べ，3章で，EM-Cの構文の概要について述べる．4章において，共有メモリベンチマークSPLASHの一つであるスタンダードセルのルータLocus RouteのEM-4での実現について報告する．

2. EM-Cプログラムの実行時構造

2.1 EM-4マルチプロセッサ

EM-4プロトタイプは，80の要素プロセッサ（PE）からなるマルチプロセッサシステムであり，1990年4月から稼働している．EM-4の要素プロセッサEMC-Rは，それぞれローカルなメモリをもっており，オメガネットワークで結合されている．

EMC-RはRISCアーキテクチャをとっており，パイプラインは逐次実行とデータ駆動機構による通信と同期処理が融合されるように設計されている．ネットワーク上のメッセージは，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．**図1**にパケットのフォーマットを示す．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．起動するスレッドは，アドレス部にあるオペランドセグメント番号（OSN）とオフセットによって指定される．その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．

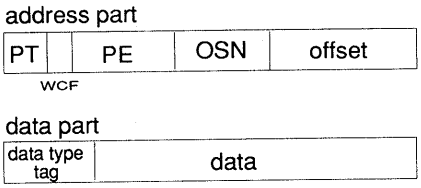


図1 EM-4のパケットフォーマット
Fig. 1 EM-4 packet format.

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．WCF（wait condition flag）はデータフローでのマッチングの属性，すなわちパケットが左オペランドトークン，あるいは右オペランドトークンであるかを指定するフィールドである．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（PT）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．PTが指定された（すなわち，0以外の）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関数フレームなどのリソース

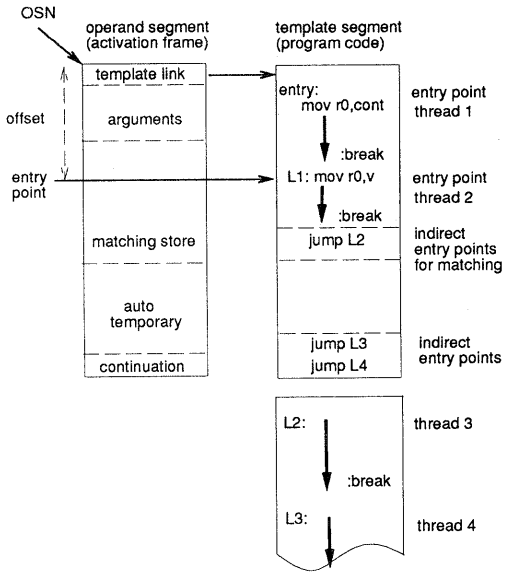


図 2 EM-C の関数実行時のデータ構造
Fig. 2 Runtime structure of EM-C program.

の割り当てにも用いられている.

2.2 EM-C の関数の実行

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.

図 2 に EM-C での関数実行時のデータ構造を示す. 関数フレーム（オペランドセグメント）の先頭は，呼び出す関数のプログラムコード（テンプレートセグメント）にリンクしておく. 関数内のスレッドを起動するには，オペランドセグメントの，テンプレートセグメント上のスレッドのエントリーポイントに対応するアドレスに対して，パケットを送出する. スレッドが実行される時には，セグメントレジスタにパケットで指定されたオペランドセグメントのアドレスがセットされる. オペランドセグメントは，マッチング時の待ち合わせ領域として使われる他，コンパイラによって，一時変数の退避領域として用いられる.

関数呼びだしは，ローカルにスレッド間の通信を使って行なっている. **図 3** に関数呼びだしの手順を示す. 逐次に呼び出す場合には，呼び出す関数をリンクしたテンプレートセグメントを獲得し（GETOSN），引数を書き込み，関数の先頭を起動して，呼びだし側はスレッドを終了する. 呼び出す関数の起動に際して，

```
   r0 = GETOSN($FOO)   ; allocate activation frame
   r0(1)=ARG1          ; set arguments
   R0(2)=ARG2
     ...
   GET C0,r0(entry)  :break
C0:  ...                    ; result in r0

FOO.entry:   ; continuation is in r0
   cont = r0
     function body
   MKPKT result,cont
     reclaim continuation and switch
```

図 3 関数呼びだしの手順
Fig. 3 Calling sequence of EM-C.

呼びだし側の continuation をパケットのデータとして，送る. GET 命令は，指定されたエントリポイントの continuation をデータとして，パケットを送出する命令である. ":break" は，スレッドの終了をさせる. 呼び出された関数では，パケットのデータとして渡された呼びだし側の continuation を保持しておき，関数終了時にこの continuation で関数のリターン値と共に呼びだし側を起動し，フレームを回収してスレッドを終了する. MKPKT 命令は，データとアドレスをオペランドに指定してパケットを送出する.

2.3 Remote Call と Fork

関数呼びだしは他の PE の関数の呼びだしに容易に拡張できる. `remote_call` は，関数 `func` を `pe_addr` で指定される PE で実行する操作である.

```
remote_call(pe_addr,func,narg,arg1,...)
```

図 4 にリモート関数起動の操作 `remote_call` の手順を示す. パケットに対してパケットタイプ `PT` を加えることによって，システム定義のハンドラを起動する. 呼ばれる側のコードは変わらない.

スレッドは，`fork` によって関数ごとに生成することができる. `fork` は，関数 `func(arg1,..)` を実行するスレッドを `pe_addr` で指定される PE に生成する.

```
fork(pe_addr,func,narg,arg1,....)
```

呼び出し側の命令実行を中断せずに，パケットを送出することによって，スレッドが生成される. continuation として，何もパケットを送出しないスレッドを与え，関数終了時に消滅するようにする. 通常，スレッドはその実行を終了する前に他のスレッドに対して，同期操作を用いて，実行の終了を通知する.

関数は，fork 操作により，複数の関数を起動することができるから，オペランドセグメントは tree 状に生成されることになる. これらの操作は，コンパイラ

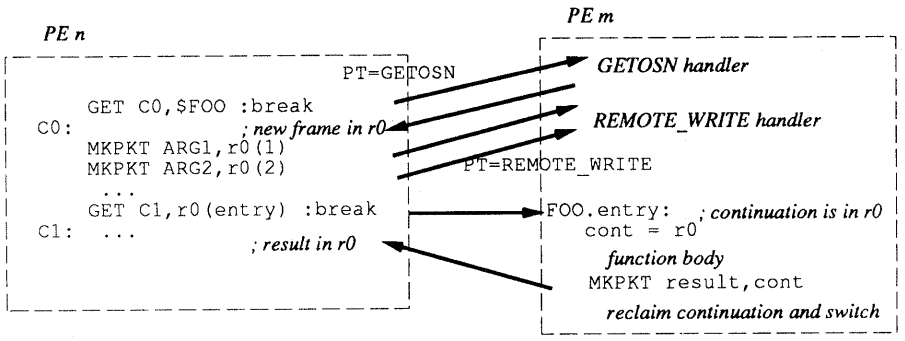


図4 remote call の手順
Fig. 4 Remote calling sequence.

によって in-line 展開される.

2.4 同期オペレーション

同期には，同期メモリの他，バリア同期もライブラリとして提供されている.

同期メモリ — I-structure 操作[2]を任意のアドレスに対して可能である．我々は，これを複数 reader / writer を許すように拡張している．我々はこれを Q structure[8]と呼んでいる．この同期構造体を用いることによって，読みだし側と書き込み側を1対1に対応させることができ，これを双方向のチャネルとして用いることができる.

バリア同期 — fork によってすべてのプロセッサにスレッドを生成し，その間でバリア同期する関数が提供されている[10]．この操作はパケット通信によって，ソフトウエア的にネットワークを作り，実現している．同期時に sum などの reduction 演算を行なうことができる.

また，通常のロック操作もライブラリとして提供されているが，スレッドの実行は関数コールまで中断することはないので，これを利用して，簡単な排他制御を行なうことができる.

2.5 基本操作の性能

表1 に EM-C での基本操作の実行時間を示す．**表2** には，基本操作の一つとして，リモートメモリアクセスの時間を示す．リモート操作は，80PE に対する平均実行時間である．remote call は，空な関数を実行した．なお，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ポート当たり 60.9 Mbytes/sec である.

表1 EM-C でのスレッド操作の実行時間 (μs)
Table 1 Execution time of EM-C thread operations(μs).

Operation	Time (μs)			
Number of arguments	0	1	2	3
local call/fork	1.68	2.16	2.64	3.12
remote fork	2.10	2.51	2.98	3.46
remote call	4.75	5.16	5.64	6.12

表2 EM-C でのリモートメモリのアクセス時間 (μs)
Table 2 Execution time of EM-C remote memory operations(μs).

Operation	Time (μs)
remote read	1.85
remote write	0.40

2.6 スケジューリング

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 resched 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も FIFO でスケジュールされる.

3. EM-C の並列プログラミング構文

3.1 分散グローバルアドレス空間でのデータ分散と参照

同じ PE で実行されるスレッドは，同じローカルメモリ空間を共有する．プログラム中で宣言されるデータは，各プロセッサ上のローカルメモリの同一アドレ

```
int global A[N_ROW]:[N_COL];
int global *p;          /* intra-PE addressing */
int shared *q[N_ROW]; /* inter-PE addressing */
```

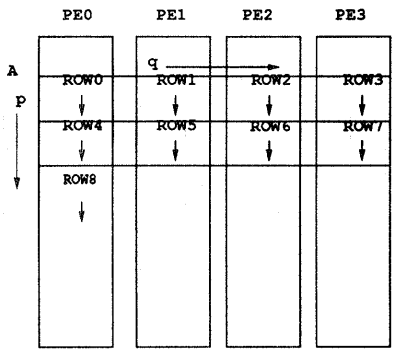


図5 グローバルデータの配置とアドレッシング
Fig. 5 Distribution and addressing of global data.

スに配置される．EM-C では，デフォールトではすべてのデータオブジェクトはローカルである．

データの宣言に対して，global キーワードを付け加えることによって，プログラマは分散グローバルアドレス空間上に配列などのデータを分散配置し，直接アクセスすることができる．**図5** に例を示す．分散配置に使われるプロセッサの配列と数は，コンパイル時に与える．global で指定されたポインタは，同じ PE 内をローカルアドレスが変化する方向にアクセスする (intra-PE addressing) ポインタである．以下の例では，他のプロセッサにある row にアクセスする:

```
int global *p;
p = A[k];
for(i = 0; i < N_COL; i++) s += *p++;
```

A[k] では，配列に対するグローバルなアドレスを計算する．コンパイラは，グローバルポインタの参照に関して，リモートメモリアクセスのコードを生成する．また，shared キーワードによって，インターリーブするように (inter-PE addressing) アクセスするポインタを用いることもできる．

ローカルなポインタとの代入操作で，グローバルなアドレスを明示的に作り出して，他の PE で参照することができる．

3.2 タスクブロック: *Forkwith* と *Dowith*

forkwith 構文は，一連のコード（これをタスクブロックと呼ぶ）を実行するスレッドを生成する．

forkwith(*parameters for task body*){
task body ... }

task body は，作られたスレッドで実行されるコードである．パラメータリストとして，タスクブロックから参照する変数を指定する．これらの変数は，タスク生成時に新しく作られたスレッドのコンテキスト（オペランドセグメント）にコピーされ，実行される．*forkwith* では，元のスレッドはブロックされず，次の文から実行を続行する．生成されたタスクのスレッドは，前章で述べた同期操作を使って，他のスレッドと同期・通信する．

forkwith の代わりに *dowith* を使うことによって，元のスレッドはタスクブロックが終了するまで，ブロックされる．*update* 指定子で，終了時に変数の値を元のコンテキストに反映させることができる．

dowith(*parameters for task body*){
task body ...
} [update(*update variable list*)]

この構文は，単独では単に *task body* のコードを実行するだけだが，以下に述べる *iterate* 構文や *where* 構文と共に用いて，スレッドを制御する単位を指定する．

3.3 データ配置によるスレッド制御: *Where*

where 構文は，データのアドレスによって，スレッドを制御する．指定されたグローバルポインタのアドレスから，データのある PE でタスクブロックが実行される．

where(*global pointer expression*)
task block statement

これにより，一旦，分散配置されたデータやグローバルポインタでリンクされたデータに関して，PE を意識せずにプログラムできる．

例えば，リモートプロシージャコールは，以下のように実現できる．

```
struct item global *p;/* global parameter */
int a;                /* local parameter */
....
where(p) dowith(a){
     v = foo(p,a,1);
} update(v);
```

これは，グローバルポインタ p のさすデータのある PE において関数 foo を変数 a と共に実行し，リターン値は変数 v で反映する．タスクブロック内では，p はローカルポインタとして解釈され，代わりに他のローカルポインタはグローバルなものになる．グローバルポインタで一連のブロックにおいて，多くの参照を行

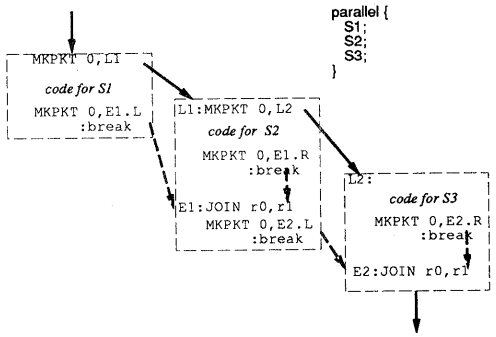


図6 Parallel 構文のコンパイル
Fig. 6 Parallel construct code.

なう場合，*where* 構文でそれらのアクセスをグループ化することによって，効率化することができる．

3.4 並列化構文

静的な並列性記述には cobegin/coend 型の parallel 構文を提供している．**図6** にコードを示す．それぞれの文を実行するスレッドは，MKPKT 命令により，fork され，同期はデータフローのマッチングを用いて，各文の実行の終了で生成されるトークンと他の文からのトークンをマッチングすることにより，実現している．

iterate 構文は同じタスクブロックを複数生成するもので，これはループなどの並列実行に使うことができる．

`iterate(`*#threads*`)`
task block statement [*reduction operation*]

最後の *reductionsum* 指定子は各タスクブロック終了時に変数についてのリダクション操作をデータフローのマッチングを使って行なうものである．ループ中にリモートレーテンシを含む場合に，複数のスレッドを使ってループを並列実行することにより，そのレーテンシを隠す場合に用いる．例えば，以下の例では，ループを N_TH 個のタスクブロックで pre-schedule 並列実行する．組み込み関数 *iterate_id* は並列ループの ID を得ることができる．

```
iterate(N_TH) dowith(){
    for(s=0, i=iterate_id(); i<N; i+=N_TH)
        s += A[i];
} reductionsum(s);
```

everywhere 構文は，すべての PE に同じタスクブロックを実行するスレッドを生成する．

`everywhere(`*global object list*`)`

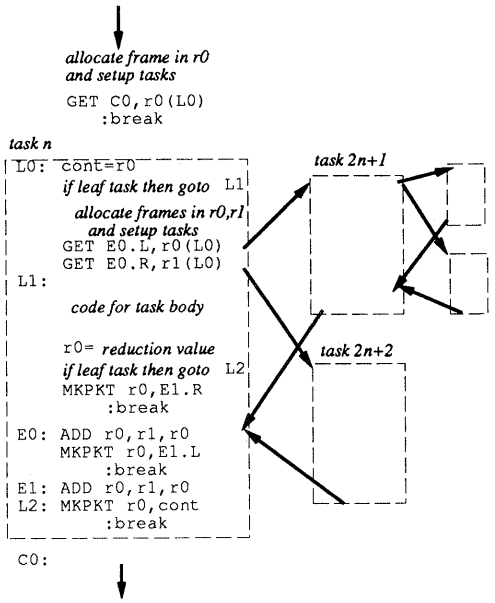


図7 並列ループのコンパイル
Fig. 7 Parallel loop code.

task block statement [*reduction operation*]

例えば，以下のコードは分散グローバルアドレス空間に配置された配列の内積を計算するものである．

```
data_t global A[N], global B[N], s;
  ....
everywhere(A,B) dowith() {
   for(s = 0, i = 0; i < N/N_PE; i++)
        s += A[i]*B[i];
} reductionsum(s);
```

指定された配列 A，B は，ブロック内では，ローカルにアクセスされる．この構文は，データ並列プログラミングなどのバリア同期としても使うことができるだけでなく，複数スレッドグループ間の複数バリアを可能とする．**図7** で示すように，スレッドの生成は tree 状に行なわれる．dowith のタスクブロックの同期を必要する場合はデータフローのマッチングを使って行なわれる．そのため，各タスクブロックは終了時に同期を行なうポイントに対する continuation をデータとして起動される．leaf でない場合には，各タスクブロックの先頭において，マッチングポイントである E0，E1，E2 に対する continuation を生成し，子のタスクブロックを起動する．

図8 に everywhere 構文の例と現在のインプリメン

```
everywhere dowith(){ /* null */ }
```
(a) 実行時間 execution time 59μs

```
everywhere dowith(p,x){ *p = x;/* write */}
```
(b) 実行時間 execution time 80μs

```
everywhere dowith(p)
  { s = *p;/* read */} reductionsum(s);
```
(c) 実行時間 execution time 74μs

図8 everywhere 並列ループの実行時間
Fig. 8 Execution time of everywhere parallel loop.

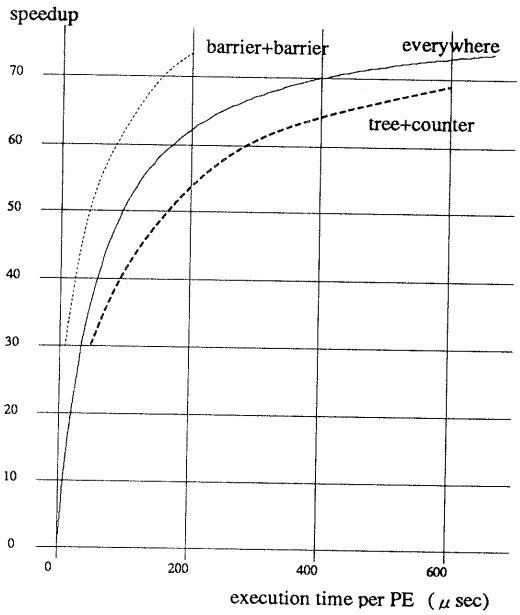


図9 everywhere 構文の性能向上比
Fig. 9 Speedup of everywhere loops.

トによる 80PE での簡単なタスクの実行時間を示す．(b) はすべての PE の同じアドレスに値を書き込むブロードキャスト操作，(c) はすべての PE の同じアドレスの値の総和を求める操作である．ブロードキャストしなくてはならない変数の数によって，実行時間は変化する．図9 に 64PE での everywhere で実行されるタスク時間と速度向上率について示す．タスクはそれぞれローカルに実行できるものとする．例えば，50 倍の速度向上率を得るには 95μsec のタスクであればよい．参考のために，すべての PE ですでに実行されているスレッド間で同期するのにバリアを用いることができる場合（barrier+barrier）と基本操作のみをつかってループを tree 状に分散，カウンタで同期する場合（tree+counter）を示す．everywhere 構文は単純な barrier よりも遅いが，タスクのパラメータをブロードキャストすることができ，構文にして埋め込むことにより，基本操作を使って書くよりも効率がよい．

3.5 並列構文のコンパイラ最適化

冗長なパケットの生成や関数フレームの生成を取り除くことにより最適化が可能である．たとえば，*where* 構文を用いて，リモートプロシジャコールをした場合，まず，タスクブロックのためのフレームとそこから関数コールのフレームの 2 つのフレームが作られることになるが，タスクブロックですぐに関数コールする場合には，直接，関数のフレームを割り当て，そこに引数をリモートメモリアクセスで書き込むことによって，冗長なフレームを省くことができる．また，関数呼びだしのあとで，すぐにタスクブロックを終了する場合には呼び出された関数から直接タスクブロックの同期を行なうようにすることによって，関数呼びだし時にタスクブロックに対するフレームを回収することができる．

スレッド間の同期に関して，単に空なデータを持つデータフローパケットで行なっているが，データの通信が必要な場合には，このデータを同期のためのデータに組み込むことによって，効率化できる．これによって，最終的に得られるコードは，データフロー言語から得られるコードに極めて近いものになる．

4. EM-C による共有メモリプログラム

分散グローバルメモリを用いることによって，共有メモリ並列プログラムを実現することが容易になる．その一つの例として，SPLASH ベンチマーク[7]の中から並列スタンダードセルルータ LocusRoute[3] を EM-C で並列化し，EM-4 上に実現した．

4.1 EM-C によるプログラミング

共有メモリ並列プログラムを EM-4 上で実現する場合，分散グローバルメモリ空間上に配置されたデータは，共有メモリとして，アクセスすることができるが，これを過度に多用すると多くのリモートメモリアクセスによって効率が低下する．したがって，共有されるデータと局所的にアクセスできるデータを区別しなくてはならない．また，データの参照が集中する場合はそれを避けるように（インターリーブするなどして）配置する必要がある．また，スレッドは，リモートメモリを読み出す場合にはレーテンシを生ずるため，リモートレーテンシを隠すためにマルチスレッド化する

```
ワイヤーデータを読み込み，PE に分散
for (N 回繰り返す){
  everywhere() dowith(){ /*全 PE で実行開始*/
   iterate(ワイヤレベルのスレッド数 (#w))
    dowith(){
     一つワイヤをとる（前回の結果を取り消す）
     どのピンを結ぶかを決定
     while(すべてのペアについて){
         iterate(ルートレベルのスレッド数 (#r))
             dowith(){  一つのルートをとり，
                 コスト配列にリモートアクセス
                 コストの計算}
           最小コストのルートを選択}
       コスト配列に選択されたルートを反映}
  /* 負荷分散 */
  他の PE のワイヤをコピーして，処理}}
```

図 10 LocusRoute の EM-C プログラムの概略
Fig. 10 Outline of LocusRoute in EM-C.

ことが必要となる．そのレーテンシを隠すためには，PE 内でも並列化を行ない，並列動作ができるようにする．

LocusRoute は，チャネル内の各配線格子点に対応した要素からなるコスト表を用いて，同じ格子点上に配線されているルートの数をコストとして，なるべく多くのルートが重ならないようにルーテングを行なうプログラムである．データとして，接続するピン・グループ（セルの上下の 2 つのセットの場合がある）の集合がワイヤのデータとして与えられる．まず，master PE がワイヤのデータを読み込み，ワイヤごとに各 PE に分散する．コスト表は，ポジションごとに各 PE に分散配置した．アドレス計算を簡単にするために，分散配置されたポジションごとのデータの 1 次元配列のアドレスを格納する配列を各 PE に格納するようにした．

図 10 に大体の EM-C プログラムの構造を示す．データ読み込み後，各 PE にスレッドを生成し，その PE に配置されたワイヤについて，ローカルにアクセスをして，ルーテングを行う．分散配置されたコスト表に対しては，リモートメモリアクセスで参照する．LocusRoute では，ワイヤ間の並列性，ピングループのペア（セグメント）間の並列性，各ルートの並列性の 3 つのレベルの並列性を持つが，PE 内において，リモートメモリによるレーテンシを隠すために，ワイヤ間，ルート間で並列に実行した．最後に，ロードバランスを行うために，PE 内のワイヤの処理が終了したら，他の PE のワイヤでまだルーテングされていないものを捜し，remote call を用いてデータを移動し，ルーテングを行なう．

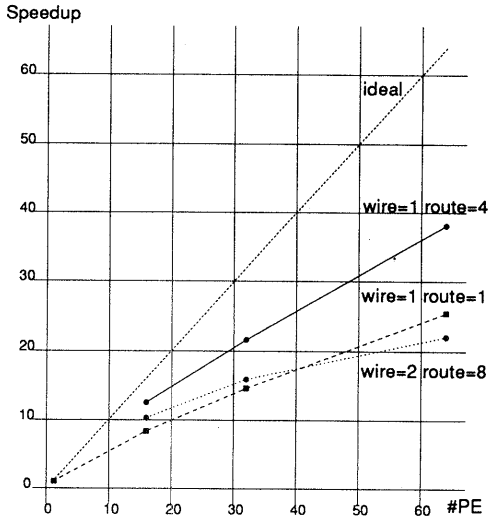


図 11 EM-4 での LocusRoute の性能向上比
Fig. 11 Speedup of LocusRoute in EM-4.

コスト表を更新する場合には，リモートメモリアクセスと同様に，特殊パケットを用いて，メモリを更新するスレッドで行なうため，ロックの必要はない．また，PE 内で並列実行する場合には EM-4 のスレッド生成は十分に早いため，元のプログラムのようにタスクキューを作らず，随時，実行時にスレッドの生成を行なうようにした．もとのプログラムでは，1 つの PE では 1 スレッドしか実行されないため，大域変数が多用されていたが，マルチスレッド化するために，いくつかの変数が局所的になるように書き直さなくてはならなかった．

4.2 実行結果

図 11 に，1PE 内において，ワイヤ，ルーテングを複数のスレッドで実行した場合のプロセッサ数性能向上比を示す．例えば，ワイヤを 2 スレッド，ルーテングを 8 スレッドで実行した場合，最大 16 スレッドが並列に実行されていることになる．データは，Primary2 (3817wire, 1920×20 チャネル) である．1PE のデータはメモリの制限のために実行できないため，実行時間は部分実行して推測して求めた．最初の 1 回のみでの比較である．ちなみに，回路の品質はプロセッサ

数が多くなると多少悪くなるが，数パーセント以内である．

1PE のデータは，完全にローカルアクセスで行なうプログラムで，並列化のためのオーバヘッドはない．比較のベースラインが多少異なるが，共有メモリプロセッサと比較して，同程度のスピードアップが得られている．(Encore Multimax では，15 プロセッサで，13.6 倍[3]．DASH では，16 プロセッサで，9.9 倍.[1] いずれも Primary2.) 共有メモリのプロセッサでは，すべてのプロセッサがコスト表をアクセスしあうために，キャッシュミスで 16 プロセッサからの速度向上が頭打ちになっている。この点では，リモートメモリを使っている EM-4 の方がスケーラビリティが良いことが確かめられた．

PE 内を並列処理することによって，リモートメモリのレーテンシを隠すことを狙ったが，Primary2 を見る限り，過度の並列性の抽出は返って性能を低下させる．これは，EM-4 では 16 パケット以上のパケットはメモリに退避する必要があり，このためにプロセッサの実行が阻害されるためである．また，ネットワークを混雑させる要因にもなる．EM-C では，スレッド数を調整し，プログラムを最適化することが可能である．

EM-4 ではすべてのスレッドが，FIFO でスケジューリングされるため，長い間スレッド中断されない場合があると，リモートメモリアクセスが待たされることになる．このようなスレッドを陽に中断させるようにプログラムをチューニングして，前報告[9]より，10%から 20%の性能向上をさせることができた．

5. おわりに

EM-C はリモートメモリアクセスの機能を用いて，EM-4 の分散グローバルアドレス空間上で仮想的な共有メモリを提供し，局所性を活用するためにスレッドをデータの配置にしたがって制御する構文を持つ．また，通信・同期のためのリモートメモリアクセスやリモートオペレーションのレーテンシを隠蔽するため，プログラマが粗粒度の並列性を記述する構文を提供している．これらは，すべて EM-4 のデータ駆動機構の枠組内で効率的に実現されている．

分散グローバルメモリを用いることによって，適用範囲の広い共有メモリ並列プログラムを実現することが容易になる．その一つの例として，並列スタンダードセルルータ LocusRoute[3] を EM-4 上に実現し，共有メモリプロセッサと同等以上の性能向上をすることが分かった．

この言語の目的は，EM-4 での並列プログラミングのための構造的な基盤を与えることにある．EM-C はすでに，データ並列言語コンパイラのターゲット言語として使われており，high level な SISAL などの関数型言語の EM-C の上に実現することを計画している．さらに，EM-4 に近い細粒度かつ低レーテンシの通信が可能な CM-5 などの計算機でも実現を検討している．

謝辞 本研究を遂行するにあたり御指導，御討論いただいた電子技術総合研究所 弓場前情報アーキテクチャ部長，島田前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) Lenoski, D., Laudon, J., Joe, T., Nakahira, D., Stevens, L., Gupta, A. and Hennessy, J.: The DASH Prototype: Implmentation and Performance, *Proc. of the 19th Annual International Symposium on Computer Architecture*, pp. 92–103 (1992).

2) Nikhil, R. S. and Pingali, K. K.: I-Structure: Data Structures for Parallel Computing, *ACM Trans. on Prog. Lang. and Syst.*, Vol. 11, No. 4, pp. 598–639 (1989).

3) Rose, J.: The Parallel Decomposition and Implmentation of an Itegrated Circuit Global Router, *Proc. of PPEARS 88*, pp. 138–145 (1988).

4) Sakai, S., Yamaguchi, Y., Hiraki, K., Kodama, Y. and Yuba, T.: An Architecture of a Dataflow Single Chip Processor, *Proc. of the 16th Annual International Symposium on Computer Architecture*, pp. 46–53 (1989).

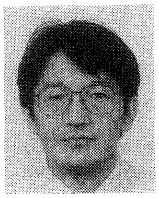
5) Sato, M., Kodama, Y., Sakai, S., Yamaguchi, Y. and Koumuara, Y.: Thread-based Programming for the EM-4 Hybrid Dataflow Machine, *Proc. of the 19th Annual International Symposium on Computer Architecture*, pp. 146–155 (1992).

6) Shaw, A., Kodama, Y., Sato, M., Sakai, S. and Yamaguchi, Y.: Data-Parallel Programming on the EM-4 Dataflow Parallel Supercomputer, *Proc. of Frontiers '92*, pp. 302–309 (1992).

7) Singh, J. P., Weber, W. D. and Gupta, A.: SPLASH: Stanford Parallel Applications for Shared Memory, Technical Report CSL-TR-92-505, Stanford University (1991).

8) 佐藤，児玉，坂井，山口: 高並列計算機 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, 情報処理学会研究報告 92-ARC-92, pp. 92–7 (1992).
9) 佐藤，児玉，坂井，山口: 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 —, 電子通信情報学会コンピュータシステム研究会 CPSY92-61, pp. 49–56 (1992).
10) 児玉，佐藤，坂井，山口: EM-C によるアクティビティ分散方式の検討, 情報処理学会第 46 回全国大会 6M-2 (1993).

(平成 5 年 9 月 16 日受付)
(平成 5 年 12 月 9 日採録)



佐藤 三久 (正会員)

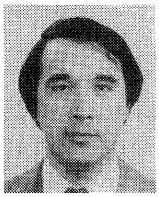
昭和 34 年生．昭和 57 年東京大学理学部情報科学科卒業．昭和 61 年同大学院理学系研究科博士課程中退．同年新技術事業団後藤磁束量子情報プロジェクトに参加．平成 3 年より，通産省電子技術総合研究所勤務．現在，同所情報アーキテクチャ部計算機方式研究室主任研究官．理学博士．並列処理アーキテクチャ，言語およびコンパイラ，計算機性能評価技術等の研究に従事．日本応用数理学会会員．

児玉 祐悦 (正会員)

昭和 37 年生．昭和 61 年東京大学工学部計数工学科卒業．昭和 63 年同大学院工学系研究科情報工学専門課程修士課程修了．同年通商産業省工業技術院電子技術総合研究所入所．以来，データ駆動計算機などの並列計算機システムの研究に従事．特に，プロセッサアーキテクチャ，並列性制御，動的負荷分散などに興味あり．現在，情報アーキテクチャ部計算機方式研究室に所属．

坂井 修一 (正会員)

昭和 33 年生．昭和 56 年東京大学理学部情報科学科卒業．昭和 61 年同大学院情報工学専門課程修了．工学博士．同年，電子技術総合研究所入所．平成 3 年 4 月より 1 年間米国 MIT 招聘研究員．平成 5 年 3 月より RWC 超並列アーキテクチャ研究室室長，現在に至る．計算機システム一般，特にアーキテクチャ，並列処理，スケジューリング問題などの研究に従事．情報処理学会研究賞（平成元年），同論文賞（平成 2 年度），元岡記念賞（平成 3 年），日本 IBM 科学賞（平成 3 年）各受賞．



山口 喜教 (正会員)

1972 年東京大学工学部電子工学科卒業．同年通商産業省工業技術院電子技術総合研究所入所．以来，高級言語計算機，データフロー計算機などの研究に従事．現在，情報アーキテクチャ部計算機方式研究室長．工学博士，1991 年情報処理学会論文賞．著書「データ駆動型並列計算機」(共著)．電子情報通信学会会員．