プログラミング－言語・基礎・実践－ 13－11
（1993. 8. 20）

超並列プログラミング言語 MPC++ の概要

石川 裕、小中 裕喜、前田 宗則、友清 孝志、堀 敦史
技術研究組合 新情報処理開発機構 つくば研究センタ

本稿では、現在我々が設計を進めている分散メモリを持つメッセージ駆動型超並列計算機上で実装されるプログラミング言語 MPC++ について述べる。MPC++ は、メッセージ駆動型計算機が提供する実行モデルをプログラミング言語に導入した言語である。MPC++ の特徴は以下の通りである。

1. メッセージパッシングによる関数起動を抽象化した関数インスタンスが提供されている。
2. 実行スレッドとのメッセージによる通信手段を抽象化したメッセージエントリとメッセージトークンが提供されている。
3. 分散メモリ上の論理的空間を規定するトポロジおよび空間および空間上での実行スレッドをプロセッサにマッピングするための論理プロセッサ群が提供されている。

An Overview of Massively Parallel Programming Language : MPC++

Yutaka Ishikawa, Hiroki Konaka, Munenori Maeda, Takashi Tomokiyo, Atsushi Hori
Tsukuba Research Center
Real World Computing Partership
1-6-1 Takezono, Tsukuba,
Ibaraki, 305, JAPAN

In this paper, we present an overview of a new programming language called MPC++ which is implemented on message-driven massively parallel machines. MPC++ introduces an execution model provided by a message driven architecture. The key features are to provide i) a *function instance* which is the abstraction of function invocation by message passing, ii) a *message entry and token* which is the abstraction of communication among threads, and iii) the notion of *topology* and *processor group* which gives us the abstraction of a logical memory space and processors.

1 はじめに

我々はRWC-1[3] 等のメッセージ駆動型並列計算機上でオペレーティングシステムカーネルおよびデータ並列処理を含むシステムソフトウエアを実現するための実装用プログラミング言語としてC++を拡張したプログラミング言語MPC++を設計している。MPC++では、分散メモリを持つメッセージ駆動型並列計算機を抽象化したプログラミングモデルを導入している。メッセージ駆動型計算機では、1) メッセージ駆動による実行実体（スレッド）の起動、2) メッセージの待ち合わせ機構、のための基本機能がハードウエアによって支援されている。また、メッセージ駆動型並列計算機の各PEはローカルなメモリを持ち、多くの場合、メモリを共有するためにはソフトウエアで共有機能を実現する必要がある。

このような計算機が提供している実行モデルをプログラミング言語に導入する方法として、MPC++では以下のような設計方針をとる。

1. 低レベル言語機能として実行モデルを最小限の抽象化により提供する。ただし、メッセージのタイプ検査が可能となるような言語枠組を提供する。
2. 分散メモリ上の変数配置のための言語機能およびそれら変数上の基本操作を提供する。
3. 以上の言語機能を組み合わせて構造化機能を提供する。

このような設計方針に基づきMPC++では、1) メッセージパッシングによる関数の起動を抽象化した関数インスタンス、2) メッセージによる通信手段を抽象化したメッセージエントリとメッセージトークン、3) 分散メモリ上への変数割り当ておよびアドレッシングを抽象化したトポロジおよびトポロジ上の変数およびデータ並列のための実行スレッドをマッピングするための論理プロセッサグループ、4) 強力なマクロ機能、をプログラミング言語に導入する。

本稿では、まず、第2章で分散メモリ上への変数割り当てについて例を用いて説明した後、第3章で関数インスタンスについて概説する。第4章ではメッセージエントリとトークンについて概要を述べる。第5章では、メッセージエントリとトークンを用いたプログラミング例として、クラスによって実現したI構造体[2] の例を示す。並列構文については第6章で概説する。本稿では、MPC++のマクロ機能については述べない。

2 分散記憶領域における変数

分散記憶領域における変数の扱い方は概ね以下のようにまとめることができるであろう。

1. scalar 変数記憶領域を全PE内ローカルにとる。
2. scalar 変数記憶領域をPE間グローバル領域にとる。
   (a) その記憶領域の配置を決める。
   (b) その記憶領域の配置を変更する。
   (c) その記憶領域の配置は処理系依存。
3. parallel 変数記憶領域をPE間に跨ってとる。
   (a) 配列の分散配置方法を制御する。
   (b) その配置方法を変更する。

MPC++では、これらの変数領域の宣言および配置制御を実現する。分散記憶領域記述は、単なるストレージクラスというだけの意味ではなく、変数に対するアクセス方法および操作についても規定する概念である。そこで、MPC++ではストレージクラスに対する修飾子ではなく、タイプに対する修飾子として扱い、あらたなるタイプ宣言時にのみ使えるものとする。

MPC++では、以下に述べるトポロジおよびプロセッサグループにより、変数の分散記憶領域配置およびそれら変数に対する演算を規定する。

- トポロジ
  論理的分散記憶領域およびアクセス方法を抽象化した概念。
- プロセッサグループ
  トポロジによって定義された分散記憶領域およびデータ並列による実行実体を実行するための論理プロセッサ群。

2.1 簡単な型宣言

以下のようにCやC++のシンタックスに準拠した変数は全てのプロセッサにローカルに存在する変数として扱われる。s1はint型を持つPE毎にローカル変数であり、全てのPEに配置される。

```
extern int          s1;
```

以下がトポロジの定義例である。

```
1 topology scalar          glb;
2 topology array[100]      top1;
3 topology array[10][10]   top2;
```

scalar領域として、glbという名前を定義している。scalar領域を持つ変数は全PE上に唯一存在する。

一次元配列で100要素もつ領域として、top1というトポロジ名を定義している。top2というトポロジは二次元配列である。scalar, arrayの他にbinaryTree, list, setも提供する。トポロジを使った型宣言例および変数宣言例を以下に示す。

```
1 typedef int@glb          intglb;
2 typedef int@top1         inttop1;
3 typedef int@top2         inttop2;
4 extern intglb            s1;
5 extern inttop1           s2;
6 extern inttop2           s3;
```

型intglbは、glbトポロジを持つint型として定義されている。s1はintglb型を持つ変数である。intglb型はglbトポロジ上の型であるため、s1変数は全PE上に一つしか存在しない。型inttop1は、top1トポロジ上にint型として定義されている。s2はinttop1型を持つ変数である。inttop1型はtop1トポロジ上の型なので、s2変数はtop1トポロジ上に分散される。

変数の参照方法は以下の通りである。scalarトポロジを持つ変数は、CおよびC++同様の参照方法である。排他制御方法については6.1節を参照のこと。配列トポロジを持つ変数は@の後に添字をつけることにより参照可能となる。

```
1 sub()
2 {
3     s1 = 1;
4     s2@[10] = 2;
5     s3@[4][2] = 3;
6 }
```

2.2 ポインタを含む型宣言

先に宣言したintglb、inttop1へのポインタ型宣言例を以下に示す。

```
1 typedef int@glb          *intgp;
2 typedef int@top1         *inttp1p;
3 extern intgp             s1p;
4 extern inttp1p           s2p;
```

ここでは、s1p, s2pともに全てのプロセッサにローカルに存在するポインタ変数として定義されていることに注意されたし。s1p, s2pは以下のように記述しても良い。

```
1 extern intglb            *s1p;
2 extern inttop1           *s2p;
```

あるトポロジ上のポインタ変数は、以下のように型として宣言する必要がある。

```
1 typedef int              *gpint@glb;
2 typedef int              *tp1pint@top1;
3 extern gpint             s3p;
4 extern tp1pint           s4p;
```

上記例において、gpintはglbトポロジ上のint型変数を指すポインタ型として宣言されている。tp1pintはtop1トポロジ上のint型変数を指すポインタ型として宣言されている。より複雑な宣言例を以下に示す。

```
1 typedef int@glb          *gpintgp@glb;
2 typedef int@top2         *tp1pinttp1@top1;
```

型gintgpは、glbトポロジ上に存在しglbトポロジ上のint型変数を指すポインタ型である。型tp1pinttp1は、top1トポロジ上に存在しtop2トポロジ上のint型変数を指すポインタ型である。

3 関数インスタンス

実行実体としてのスレッドは関数インスタンスによって実現される。関数インスタンスとは、関数にメッセージを送ることによって生成されるスレッドのことである。以下に関数インスタンス生成方法を示す。

7行目では、現在実行されているプロセッサ上でhoo関数インスタンスを作り実行している。hoo関数インスタンスからの返答メッセージが到着するまで、sub1関数の実行は待たされる。8行目では、プロセッサ番号3上でgoo関数インスタンスを作り実行している。goo関数インスタンスからの返値をiに代入している。

```
1 void     hoo(int, int);
2 int      goo(int);
3 sub1()
4 {
5     int          i;
6
7     hoo(10, 20) @ [];
8     i = goo(10) @ [3];
9 }
```

図 1: 関数インスタンスの例

3.1 関数インスタンスのパラメータ

分散メモリ型計算機上のプログラミング言語において、PE間に跨ってアドレス渡しが行なわれる場合は、共有仮想記憶または等価な機能を実現する必要がある。共有仮想記憶が提供されない場合、例えば、リモートメモリのread/writeの度にメッセージを送るような手段もある。分散システム記述用言語のようにパラメータをマーシャリング／アンマーシャリングして、データのコピーのみしか許さない言語もある。

MPC++では、PEに跨ったアドレス渡しをする場合には何らかのトポロジ上の変数へのポインタでなけれ

```
 1 void
 2 p1(token(int) ll)
 3 {
 4     ...;
 5     ll <- 10;
 6     ...;
 7 }
 8 sub2()
 9 {
10     entry(int)          ll;
11
12     for (i = 0; i < 10; i++) {
13         p1(ll) @ (null)[]; // コンティニュエーションがnullなので、
14                            // forkと同等（詳細は第4.2節参照のこと）
15         ...;               // ここでは、llのR-Valueは UNDEFである。
16       ll(int k):/* receive */
17         ...;               // 受信した後は、llのR-Valueはトークン値である。
18     }
19 }
```

図 2: メッセージエントリおよびトークン型使用例

ばならない。以下のプログラム中、他のプロセッサ上での関数インスタンス生成はエラーとなる。

```
 1 swap(int *ip)
 2 {
 3     int          i = *ip;
 4     *ip = *(ip + 1); *(ip + 1) = i;
 5 }
 6 main()
 7 {
 8     int          ia[20];
 9     swap(ia) @ [10];     // error
10     swap(ia) @ [];       // ok
11     swap(ia);            // ok
12 }
```

以下のプログラムでは、関数 swap の引数がトポロジ上のデータを指すポインタ変数なので、正しい。

```
 1 swap(intglb* ip)
 2 {
 3     int          i = *ip;
 4     *ip = *(ip + 1); *(ip + 1) = i;
 5 }
 6 main()
 7 {
 8     int          ia[20];
 9     swap(ia) @ [10];     // ok
10     swap(ia) @ [];       // ok
11     swap(ia);            // ok
12 }
```

構造体は call by value で構造体がコピーされる。MPC++ では、構造体のメンバ変数のうち全てのポインターが何らかのトポロジ上の変数へのポインターである場合に限り、その構造体をリモート関数生成における関数引数として渡せることにする。

4 メッセージエントリとトークン

関数インスタンス間でのメッセージ交信を実現するためにメッセージエントリとトークン型を導入する。図2に例を示す。図2の10行目は、メッセージエントリ宣言例である。メッセージエントリは、エントリ名とそのエントリに到着するメッセージの型から構成される。2行目では、トークン型の変数 ll を関数 p1 のパラメータとして宣言している。トークン変数は送信点、メッセージエントリは受信点として使われる。メッセージエントリに送られるメッセージを唯一のメッセージとするために、メッセージエントリの単一参照性を保証する。メッセージエントリおよびトークンのセマンティックスは以下のように規定される。

1. メッセージエントリ変数を R-Value として参照するとトークンが取り出せて、変数には UNDEF が設定される。このトークンは token という型を持つ。
2. メッセージエントリ変数にメッセージが到着した後、その変数の R-Value を取り出すとトークンが取り出せる（トークンの再利用性）。
3. token 型変数の R-Value はトークンであり、R-Value として参照すると変数には UNDEF が設定される。
4. token 型変数に対して可能な操作は <- だけである。
5. token 型変数あるいは entry 型変数に束縛されているメッセージトークンに対し <- 操作が一回だけ可能である。<- 操作により、メッセージが送信さ

れる。

6. token 型変数あるいは entry 型変数に <- 操作を行なった後、その変数の R-Value は UNDEF となる。
7. メッセージエントリにメッセージが到着しないうちにエントリが消滅する場合（関数の実行終了時）、メッセージエントリ消滅例外事象が起きる。

なお、以下のようにメッセージエントリに送るメッセージのアーギュメントは 2 つ以上であっても構わない。

```
 1 extern void ff(entry(int, int));
 2 aaa()
 3 {
 4     int                 i, j;
 5     entry(int, int)     l1;
 6
 7     ff(l1);
 8     ...;
 9  l1(i, j):
10     ...;
11 }
```

4.1 コンティニュエーションとしてのメッセージエントリ

関数インスタンスの実行時における関数からの返値授受はメッセージエントリおよびトークンで表現される。これはコンティニュエーションの表現方法であると解釈できる。図1のプログラムは以下のプログラムと等価である。

```
 1 void    hoo(int, int);
 2 int     goo(int);
 3 sub1()
 4 {
 5     int          i;
 6     entry(void) l1;
 7     entry(int)  l2;
 8
 9     hoo(10, 20) @(l1)[];
10   l1: // entry point
11     goo(10) @(l2)[3];
12   l2(i): // entry point
13 }
```

9行目では、hoo 関数インスタンスを生成し、hoo 関数インスタンスからの返答メッセージを l1 メッセージエントリで受け付ける。11行目では、goo 関数インスタンスを生成し、goo 関数インスタンスからの返答メッセージを l2 メッセージエントリで受け付け、値を i に格納する。関数インスタンスからの返答値送出方法は、return 文または reply 文によって可能である。reply 文は MPC++ で導入された構文である。

関数側は返値を返すための token を以下のように拾うことが可能である。関数のコンティニュエーションがメッセージエントリ（token 型）として実現されていると考えることが出来る。

```
1 int
2 goo(int a1, int a2)@(cont)
3 {
4     cont <- a1 + a2;
5 }
```

上記プログラム中 cont 変数は token(int) 型で定義されているのと同等であり、cont 変数に関数インスタンス呼び出し側のラベルが束縛される。当然、cont 変数の内容を他の関数に渡すことが可能である。

メッセージ合流ラベルに null メッセージが送れる必要がある。そのために null という予約語を使う。

```
1 void
2 hoo(int a1, int a2)@(cont)
3 {
4     cont <- null;
5 }
```

4.2 スレッドフォーク

関数インスタンスの生成はスレッド生成とみなすことが出来る。前節で述べたコンティニュエーション表現により、関数インスタンスによるスレッドフォークと同等の記述が可能となる。例えば、以下のプログラムでは f1 および f2 関数インスタンスを生成する。f1、f2、ff の実行は並列に実行され、10行目および 11行目で同期する。

```
 1 void    f1(), f2();
 2 ff()
 3 {
 4     entry(void) l1;
 5     entry(void) l2;
 6
 7     f1() @(l1)[7]; // fork
 8     f2() @(l2)[9]; // fork
 9     ...;
10   l1: // join
11   l2(i): // join
12 }
```

以下のように、関数インスタンス生成時に返値を受けとるメッセージエントリの代わりに nullcont を記述すると、join なしの fork と同様のセマンティックスとなる。

```
1 extern int      foo();
2 main()
3 {
4     foo() @ (nullcont)[1];
5     ...;
6 }
```

4.3 同期機構としてのメッセージエントリ

以下は、複数のメッセージエントリで待機する方法である。

```
 1 extern int       p1();
 2 extern int       p2();
 3 abc()
 4 {
 5     entry(int)   l1;
 6     entry(int)   l2;
 7     p1() @ (l1)[10];
 8     p2() @ (l2)[20];
 9     wait {
10     l1(int i):
11         ...;
12         continue;
13     l2(int i):
14         ...;
15         continue;
16     }
17 }
```

メッセージエントリにメッセージが到着しないうちにエントリが消滅する場合は、メッセージエントリ消滅例外事象が起きる。

4.4 メッセージエントリおよびトークンについての考察

4.4.1 プリミティブ性

メッセージエントリはI構造体[2]の制限的使用方法と考えることも可能である。すなわち、entry型によるI構造体の宣言、I構造体のread操作だけを制限したtoken型変数の導入であると考えられる。I構造体をMPC++言語のBuiltin Typeとして導入しなかった理由は以下の通りである。

1. I構造体のライフタイムの問題
   I構造体を関数実行時と同じライフタイムで生成した場合、関数実行が終了してもread/writeが完了するまでフレーム消滅を待たなければならない。関数実行が終了しても関数実行フレーム中に消滅することの出来ないI構造体が存在していると、関数実行フレームの管理が複雑になる。このような場合、I構造体はヒープ領域に作成する必要がある。I構造体の使い方は自由であるが、なるべく関数実行フレームにI構造体が確保されるようなプログラミングを強制すべきである。
2. 単一read/write保証の問題
   I構造体の参照を無闇に渡すと単一read/write性が失われる可能性がある。実行時にこれを検出するのは効率が悪い。
3. メッセージエントリおよびトークンを用いて、C++のクラスとしてI構造体は実現できる。実現方法を第5章に示す。

4.4.2 制限

メッセージエントリおよびトークンは、i) メッセージエントリを宣言しているスレッドがメッセージ受信を、ii) メッセージエントリのトークンを受けとった側がメッセージ送信を、それぞれ行なうという制限がある。メッセージエントリを宣言し、受信ポイントを他のスレッドに渡すことは許されない。しかし、第5章に示すようにC++クラスによって、そのような機能は容易に実現可能である。

5 プログラミング例

メッセージエントリおよびトークンによるプログラミング例としてI構造体クラス定義例を示す。

```
 1 class Istructure {
 2 private:
 3     entry(Object *)      cont;
 4 public:
 5     void          write(Object *v)
 6                   { cont <- v; }
 7     Object        *read()
 8                   { cont(Object *val):
 9                     return val; }
10 };
```

read関数を呼び出したスレッドは、メッセージエントリcontで待つことになる。write関数を呼び出したスレッドは、contメッセージエントリのトークンを使って値を送出する。Istructureクラスの使用例を以下に示す。

```
1 test()
2 {
3     Istructure  *pi1 = new Istructure;
4     bar(pi1)@(null)[];
5     pi1->read();
6 }
```

このようにC++のクラス定義機能を用いることにより、様々な有用な言語機能を構築していくことが出来る。例えば、Q構造体[4]は、I構造体リストを作りそれを管理するクラスとして定義すれば良い。このようなクラス実現では、リスト操作時等の処理においてクリティカルリージョンを実現する必要がある。MPC++では、クリティカルリージョンのための言語機能としてatomic文を提供している。また、相互排除のためのMutexクラスやバイトストリームのためのStreamクラス等がメッセージエントリおよびトークンを使って容易に実現できる。

6 並列制御構文

MIMD 的プログラミングのためのプリミティブは、関数インスタンス、メッセージエントリおよびトークンによって実現できることを第3、4章で示した。本章では、まず、複数スレッドからの排他的変数参照方法について述べた後、データ並列のための言語機能について説明する。

6.1 複数スレッドからの排他的変数参照

複数のスレッドからの参照による相互排除問題はプログラマの責任である。相互排除機能は、メッセージエントリおよびトークンにより実現される Mutex クラスによって提供される。相互排除記述を容易にするために、**mutex** 文を提供する。以下に **mutex** 文の例を示す。

```
 1 Mutex   rr;
 2 sub1()
 3 {
 4     S10;
 5     mutex (rr) {
 6         S11;
 7     }
 8     S12;
 9 }
10 sub2()
11 {
12     S20;
13     mutex (rr) {
14         S21;
15     }
16     S22;
17 }
```

プログラム中、mutex 文で囲まれている S11 および S21 は排他的に実行される。

6.2 with, where 文

プリミティブデータ型の並列変数に対しては、C*[1] 同様の with 文および where 文を提供する。以下に例を示す。

```
 1 topology array[10][10]  top2;
 2 typedef int@top2        inttop2;
 3 mul(inttop2 a1, inttop2 a2)
 4 {
 5   with (top2) {
 6     int i, sum = 0;
 7     for (i = 0; i < 10; i++) {
 8       sum +=
 9         a1@[pos(0)][i]*a2@[i][pos(0)];
10     }
11     a1 = sum;
12   }
13 }
```

上記プログラム中、pos 関数は top2 トポロジにおける with 文中で参照できる疑似関数である。pos 関数の引数は配列の次元である。pos(0) はトポロジ上の各要素上でデータ並列実行される時の X 座標上の位置を表現する。9行目の例のように、トポロジ上の変数の要素を参照するために、@ の後に添字を記述する。

with 文中からの関数呼び出しは、以下のように同一トポロジ上の関数として定義されている必要がある。

```
1 extern foo(int i, inttop2 j) @ top2;
2 sub2(inttop2 a1)
3 {
4   with (top2) {
5     int i;
6     foo(i, a1);
7   }
8 }
```

1行目は、top2 トポロジ上に foo 関数を宣言している。top2 トポロジ上の各要素に foo 関数が定義されていると解釈する。5行目は、top2 トポロジ上の各要素が持つローカル変数を定義している。6行目で、foo 関数を呼び出している。foo 関数呼び出しの別の方法は、以下のように foo 関数の要素を特定する方法である。

```
foo(10, a1) @ [1][2];
```

6.3 プロセッサ制御

with 文中、データ並列を実行するプロセッサグループを指示することが可能である。

```
 1 procgroup                p1, p2;
 2 topology array[10][10]   top2 on p1;
 3 typedef int@top2         inttop2;
 4 inttop2                  a1, a2;
 5 mobile inttop2           a3;
 6 sub()
 7 {
 8   par {
 9     with (top2) {
10       int        i, sum = 0;
11       for (i = 0; i < 10; i++) {
12         sum +=
13           a1@[pos(0)][i]*a2@[i][pos(0)];
14       }
15       a1 = sum;
16     }
17     with (top2) on (p2) {
18       int        i, sum = 0;
19       for (i = 0; i < 10; i++) {
20         sum +=
21           a3@[pos(0)][i]*a2@[i][pos(0)];
22       }
23       a3 = sum;
24     }
25   }
26 }
```

上記プログラムにおいて、1行目でプロセッサグループ p1 および p2 を規定している。MPC++ では、実行時に獲得できた物理プロセッサに応じて各プロセッサグ

ループにプロセッサを割り当てることが可能である。2行目では、top2 トポロジに関するデータ配列処理は暗黙ではp1 上で実行することを宣言している。17行目では、with 文をp2 プロセッサ上で実行するよう指示している。5行目のa3 変数宣言の例のように、変数をプロセッサ間で移動可能とするmobile ストレージ修飾子がある。a3 変数が移動可能変数なので、17行目のwith 文をp2 プロセッサ上で実行することが可能となる。

6.4 並列クラス

C++ におけるクラスのメンバ関数に、新たにparallel public、parallel private というアクセス方法を導入する。parallel public/private のメンバ関数が呼び出されるとデータ並列処理が行なわれる。parallel public/private 中に定義されているメンバ関数を並列メンバ関数と呼ぶことにする。並列メンバ関数で、クラスと同一のトポロジを持つ並列変数が定義されていた場合、各要素上の関数は、その要素上に存在する並列変数が参照できる。並列メンバ関数の返り値がintやcharのようなscalar データタイプの場合は、そのメンバ関数はリダクション操作のための関数として扱われ、リダクション関数と呼ばれる。以下にプログラム例を示す。

```
 1 class c1 @ top1 {
 2 private:
 3   int   b;
 4   int   c;
 5 parallel public:
 6   c1(int a1, int a2) { b = a1; c = a2; }
 7   c1 &operator+(c1 &a) {
 8     return c1(b + a.b, c + a.c);
 9   };
10   int aReduction() { return b + c; }
11   int !aReduction(int lv, int rv)
12         { return lv + rv; }
13 parallel private:
14   c1 &operator-(c1 &a) {
15     return self->b - a.b;
16   };
17 };
18
19 c1       o1, o2, o3;
20 cex()
21 {
22    int   i;
23    o1 = o2 + o3;
24    i = o1.aReduction();
25 }
```

上記プログラム例において、23行目の＋演算は、7行目で定義されている＋並列演算が呼び出される。24行目はaReduction リダクション関数呼びだし例である。10行目および11行目がreduction 操作定義例である。10行目の関数は、トポロジ上の各々の要素がreduction 時に使う値を計算する関数である。11行目に示すようにリダクション関数名の先頭に!をつけた関数を定義する必要がある。本関数は、2つの要素がaReduction 関数を実行した結果の値を使ってリダクションする時の処理を規定する関数である。

7 おわりに

本稿では、現在設計しているMPC++ について現在の仕様を基に概要を述べてきた。MPC++ では、トポロジおよびトポロジ上の操作によるSIMD 的実行記述、関数インスタンス、メッセージエントリとトークンによるMIMD 的実行記述が実現されている。MPC++ はRWC-1 等のメッセージ駆動型超並列計算機を想定して設計を進めているが、それらの計算機専用言語ではない。CM-5 やParagon 等の商用高並列計算機上でも効率良く実装できる汎用ポータブル言語として検討を進めている。MPC++ は設計段階にあるために、本稿で述べた仕様は流動的である。今後、カーネルやシステムソフトウエア記述およびポータビリティの検討を通して、MPC++ の言語機能を見直していく予定でいる。

謝辞

日頃から、御討論頂いているRWC 超並列プログラミング言語ワーキンググループおよび超並列ソフトウエアワークショップグループのメンバの方々に感謝致します。また、本研究を御支援して頂いているRWC プロジェクト関係者および議論に参加して頂いている全ての方々に感謝致します。

参考文献

[1] Thinking Machines Corporation. *C* Programming Guide.* 1993.

[2] R.S. Nikhil and K.K. Pingali. I–Structure: Data Structures for Parallel Computing. *ACM Trans. on Programming Languages and Systems*, Vol. 11, No. 4, pp. 598–639, 1989.

[3] 坂井, 岡本, 松岡, 広野, 児玉, 佐藤, 横田. 超並列計算機RWC-1 の基本構想. 並列処理シンポジウムJSPP'93, pp. 87–94, 1993.

[4] 佐藤三久, 児玉祐悦, 坂井修一, 山口喜教. 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. 情報処理学会計算機アーキテクチャ研究会, 1992.