プログラミング－言語・基礎・実践－ 13－12
（1993. 8. 20）

超並列オブジェクトベース言語 *OCore* の概要

小中 裕喜　石川 裕　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

超並列プログラミングモデルの研究基盤として超並列オブジェクトベース言語 *OCore* を提案する。*OCore* では従来の並列オブジェクト指向言語の基本部分に加え、オブジェクトの集合の構造化とメッセージの分散処理、効率的な実装などを目的として「共同体」という概念を導入している。また例外やイベントを処理するメタアーキテクチャや強い型づけなどにより、柔軟かつ効率的な並列プログラムの記述を可能とする。

An Overview of *OCore* : A Massively Parallel Object-based Language

Hiroki Konaka, Yutaka Ishikawa
Munenori Maeda, Takashi Tomokiyo, Atsushi Hori

Tsukuba Research Center, Real World Computing Partnership
Tsukuba Mitsui Building 16F, 1-6-1 Takezono
Tsukuba-shi, Ibaraki 305, Japan

In this paper we propose a massively parallel object-based language *OCore* as a research vehicle for massively parallel computation models. In addition to the fundamentals of existing parallel object-oriented languages, *OCore* has introduced a notion of "community", a structured set of objects that enables distributed processing of messages while it can be implemented efficiently. *OCore* is strongly-typed and provides meta architecutures for handling exceptions and events. These features help programmers write flexible and efficient parallel programs.

1 はじめに

超並列プログラミングモデルの研究基盤として超並列オブジェクトベース言語 *OCore* を提案する。*OCore* は従来の並列オブジェクト指向言語の基本部分に加え、超並列計算のための基礎となる部分を提供することをめざしている。超並列計算モデル研究者用の Research Vehicle とするため、*OCore* は簡素で拡張性の高い並列 Object-based 言語として設計されている。

OCore の特徴を以下にまとめる:

- オブジェクトの集合の構造化とメッセージの分散処理、効率的な実装などを目的として「共同体」という概念を導入している。
- 例外やイベントを処理するメタアーキテクチャを有する。
- カラーによるアクセス制御により、並列プログラムの記述性/安全性を向上させている。
- 同期構造体を用いた通信により柔軟な同期を可能とする。
- 強い型づけと簡素な言語セマンティクスによりプログラムの安全性と実行効率の向上を図る。
- グローバル GC を有する [3]。

OCore は RWC で開発される超並列マシン RWC-1[7] やその他の疎結合並列計算機を主なターゲットとする。

2 *OCore* の概要

2.1 *OCore* におけるオブジェクト

オブジェクトはメッセージパッシングなどを行ないながら処理を進める計算主体である。*OCore* におけるオブジェクトの処理はベースレベルとメタレベルにわかれ、通常のメッセージ処理はベースレベルで行なわれる。ベースレベルにおけるメッセージの処理は逐次的であり、メッセージ処理の並列性およびメソッド内の並列性は考慮していない。

オブジェクトのふるまいはクラスによって記述される。クラスにはインスタンス変数、メソッド、ローカル関数、メタ変数、イベントハンドラ、例外ハンドラなどが定義される。イベントハンドラはオブジェクトの生成やメッセージ処理の終了など特定のイベントのタイミングで起動されるものであり、例外ハンドラは例外事象の発生によって起動される。イベントハンドラと例外ハンドラの実行はメタレベルで行なわれ、そこでは特別な擬変数、メタ変数、オペレータを実行することが可能となっている。詳細に関しては [5] を参照されたい。

オブジェクトはベースレベル及びイベントハンドラを実行するスレッドと、例外ハンドラを実行するスレッドを各々1 つ有する。

インヘリタンスは現状では扱っていないが、非常に重要な概念であり導入する方向で検討中である。

2.2 共同体

前節で述べたように例外処理を除いてはオブジェクト内の並列性は失われている。従って、オブジェクトというアブストラクションが失った並列性を別のアブストラクションで提供する必要が生じる。われわれはそのために「共同体」という概念を導入する。

共同体はオブジェクトの集合を、インデックスによってはられた多次元空間の中で構造化するものである。共同体のふるまいは共同体の要素となるオブジェクトのクラスと、共同体テンプレートと呼ばれるものによって記述される。共同体については節をあらためて述べる。

2.3 タイプ

OCore では強い型づけを行なうことにより、プログラムの安全性と実行効率の向上を図っている。

ビルトインのデータタイプには整数 (`Int`)、浮動小数 (`Float`)、アトム (`Atom`)、ストリング (`String`) といった基本データタイプのものの他に、リスト (`List-of`)、タプル (`{...}`)、配列 (`Array-of`) などがある。また *OCore* において特に重要なビルトインデータタイプとして、I 構造体 (`Istr-of`)[4]、Q 構造体 (`Qstr-of`)[6] などの同期機構をもった同期構造体がある。

クラス名はそのままタイプ名として扱う。また、クラス、共同体テンプレートはパラメトリックに定義することが可能であるが、これらの名前もパラメトリックなタイプのコンストラクタとなる[1]。またタイプのユニオンも許す。

メタレベルや特定のオペレータでのみ扱えるタイプとしてはメッセージ型 (Message)、オブジェクト型 (Object) などがある。

タイプを修飾するものとしてアクセス制限のためのカラーや参照を 1 回だけ利用することが可能であるこ

[1] *OCore* ではタイプやタイプコンストラクタの名前は英大文字ではじめなければならない。従ってクラス名等もそれに準ずる。

とを示す単一利用性に関するものがあるが、これらについては後述する。

2.4 シンタックス

OCore のプログラム要素としては、タイプ定義、パレット定義、グローバル/ ローカル変数定義、関数定義、クラス定義、共同体テンプレート定義、プロセッサ定義、メイン関数定義など、そしてそれらのプロトタイプ宣言がある。

シンタックスはS式をベースとしており、メッセージ式はセレクタと引数に対応する式の並びを[...] でくくって記述する。タプルも式の並びを{...}でくくって表現する。また制御構造などはCommon Lisp のものを一部援用している (do, if, cond など)。

OCore では強い型づけを行なうため、変数や関数の引数、戻り値などでは型指定が必要となる。

以下に *OCore* のシンタックスを簡単な例で示す。まずタイプ定義の例を以下に示す:

```
(typedef Q-Int (Qstr-of Int))
```

ここでは整数型のデータを格納するQ構造体のタイプを定義している。次にグローバル変数の定義を示す:

```
1 (globals (Int a
2               b :init 0
3               c :single-assign)
4         (Q-Int q))
```

グローバル変数は全PEで1つしかない変数であり、ここでは整数型のグローバル変数3つと整数型のデータを格納するQ構造体型のグローバル変数1つを定義している。またbは0に初期化している。cのあとの:single-assign は変数が単一代入則に従うことを示している。各PEに1つずつ存在するローカル変数の定義も (locals ...) を用いて同様に定義する。

関数定義も次のように複数の関数定義をまとめた形で定義される:

```
1 (functions (add1 ((Int x)) (returns Int)
2                  (1+ x))
3            (sub1 ((Int y)) (returns Int)
4                  (local ((Int z))
5                         (set z (1- y))
6                         (return z)))
7 )
```

1つの関数定義は引数のタイプ宣言のならび、戻り値のタイプ宣言、そして本体で構成される。local は式のならびからなるブロックを定義し、そのスコープでの変数定義を許す。set は変数やタプルへの代入を行うものであり、return はCにおけるものと同様である。return がなければ関数は本体の式の値を返す。

メイン関数はCの main() と同様にプログラムの開始時に実行される式を書くものであり、*OCore* ではPE 0でのみ実行される:

(main *expression*)

パレット、クラス、共同体テンプレートなどの定義については後述する。

3 オブジェクト

3.1 クラス定義

クラスにはインスタンス変数、メソッド、ローカル関数、メタ変数、イベントハンドラ、例外ハンドラ、利用パレットなどが定義される。クラス変数は存在しない。またメタ変数はイベントハンドラおよび例外ハンドラでのみ参照される。パレットに関しては3.4節で述べる。

簡単なクラス定義の例を以下に示す:

```
1 (class Foo
2   (vars (Int c))
3   (methods
4     ([:clear] (set c 0))
5     ([:add (Int x)] (set c (+ c x)))
6     ([:set (Q-Int q)] (set c (qread q)))
7     ([:read (Q-int q)] (qwrite q c))
8    ))
```

ここでは1つのインスタンス変数と4つのメソッドをもつクラス Foo を定義している。各メソッドの定義は[...] で囲まれたメッセージパターンと、カラーや参照利用に関する記述 (なくてもよい)、そしてメソッド本体の式から構成される。メッセージパターンはセレクタ[2]と引数の宣言からなる。*OCore* では同一クラス内で同じセレクタを持ちながら引数の数やタイプの異なるメソッドを持つことを許さない。

なお、ここで qread, qwrite はQ構造体に対するオペレータである。Q構造体では qread のリクエストと qwrite のリクエストは1対1に対応づけられ、一方のリクエストばかりが複数到着した場合にはキューイングされる。qwrite はQ構造体に対する書き込み要求であり、呼び出し側のスレッドはそのまま実行を継続する。一方 qread はQ構造体に対する読み出し要求であり、呼び出し側のスレッドはサスペンドする。対応する書き込み要求がすでに存在するか、あるいは到着すれば、呼び出し側のスレッドは復帰し、その書き込み要求の値を戻り値として返す。Q構造体に対する操作としては他に、書き込み要求が複数個キューイ

[2] *OCore* ではセレクタは:のあとに英小文字で始まる名前をもつ。

ングされるまで待つqwaitや、キューイングされている書き込み要求の値をすべて取り出してリストにして返すqread-allなどがある。

3.2 パラメトリッククラス

あるクラスが扱うタイプをパラメトリックにすることが可能である。以下の例ではパラメトリックなクラスを定義し、そこから整数型を扱うクラスを得ている:

```
1 (parametric-class P-C (T-Var)
2   (vars (T-Var d))
3   (methods ([:set (T-Var x)] (set d x)))
4 )
5
6 (class C-Int (P-C Int))
```

3.3 メッセージパッシング

オブジェクト間のメッセージパッシングは非同期送信のみが可能である。メッセージの返答やパラメータの遅延送信については同期構造体を用いてプログラマが陽に記述する。

さきほど定義したクラスFooのインスタンスを用いるクラスの定義例を示す:

```
1 (class Bar
2   (vars (Foo foo1 :init (new Foo)
3              foo2 :init (new Foo)))
4   (methods
5     ([:start]
6       (local ((Int x)
7               (Q-Int q :init (new Q-Int)))
8         (send foo1 [:clear])
9         (send foo1 [:add 7])
10        (send foo1 [:read q])
11        (set x (qread q))))
12    ([:next]
13      (local ((Q-Int q :init (new Q-Int)))
14        (send foo1 [:read q])
15        (send foo2 [:set q])))
16   ))
```

あるクラスのインスタンスはnewを用いて生成する。Barのインスタンス変数には初期化時にFooのインスタンスが2つ設定される。メッセージ送信式はsendのあとに受信オブジェクトとメッセージを指定したものである。

ここでメッセージ送信式の戻り値はいわゆるメッセージの返答ではなく、単に受信オブジェクトの参照が返ってくる[3]だけであるという点に注意する必要がある。*OCore*ではメッセージは送信するだけであり、その返答がもし必要ならば別のメッセージで受け取るか、もしくは同期構造体を用いなければならない。

10〜11行目は同期構造体による返答の例であり、10行目では返答を受け取るためのQ構造体をメッセージに入れて送っている。そしてその返答を11行目のqreadによって待っている。メッセージを受け取ったほうでは:readメソッドの中(クラスFooの定義の7行目)でQ構造体に返答を書き込んでいる。

14〜15行目ではさらにQ構造体を他のオブジェクトに送信しておくことにより自分を介さずに直接返答を他のオブジェクトが利用できるようにした例である。このような同期構造体はメッセージ中に複数あってもよく、またそれぞれどのオブジェクトが書き込み、どのオブジェクトが読み出すかが自由にプログラミングできる。例えば同期構造体を含むメッセージが受信オブジェクトからさらにフォワードされ、フォワード先で返答が決定される場合でも、中継オブジェクトを介さず送信オブジェクトが直接返答を受け取ることが可能である。このように同期構造体を用いることにより、柔軟かつ効率的な通信/同期の記述が可能となる。

3.4 カラー

*OCore*ではオブジェクトの参照にカラーをつけることを可能とする。カラーは参照するオブジェクトに対するアクセスの制限を加えるものである。カラーの集合をパレットと呼ぶ。パレットをもったクラスはカラーによってメソッドにアクセス制限を行なうことが可能である。そのクラスのインスタンスに対する参照をもったオブジェクトが、参照のカラーと異なるカラーをもつメソッドを呼び出すことは許されない。

```
1 (palette :RedBlack {:Red :Black})
2
3 (class C-Foo
4   (use-palette :RedBlack)
5   (methods
6     ([:clear] (color :Red) (set c 0))
7     ([:add (Int x)] (set c (+ c x)))
8     ([:read (Q-int q)] (color :Black)
9       (qwrite q c))
10   ))
```

1行目はパレットの定義である[4]。ここでは:Red, :Blackの2つのカラーからなるパレット:RedBlackを定義している。

6行目でメソッドのパターンの後にカラーの指定を行っている。これにより:Red以外のカラーをもった参

[3]後述するように参照の単一利用が指定されていれば、UNDEFが返ってくる。

[4]パレット、カラーともコロン(:)のあとに英大文字で始まる名前をもつ。

照で [:clear] というメッセージを送信することを禁止している。一般に1つのメソッドに複数のカラーを持たせることも可能である。

カラー付の参照の生成は次のように行う:

```
1  (local (((C-Foo :Red) red)
2          ((C-Foo :Black) black))
3         (set {red black} (colored-new C-Foo)))
```

ここでカラー付の参照を表わすタイプは通常のタイプ式にカラーを修飾子として付記する。また、colored-new はパレットをもったクラスを引数とし、そのカラー付参照をパレットの各カラーに対応したタプルで返す。一方、通常の new をパレット付のクラスに対して行った場合は new の返す参照は透明であると考えられ、どのカラーのメソッドでも起動することが可能である。透明な参照をカラー付の参照をとる変数に代入することは可能であるが、その逆は許されない。あるカラーを持った参照を別のカラーの参照をとる変数に代入することも当然許されない。

あるオブジェクトを複数のオブジェクトで共有するとき、特定のオブジェクトにのみある種のメソッドの起動を許したいことがしばしばある。このような場合は、共有オブジェクトのクラスにパレットを持たせ、特定のメソッドにカラーをつけるとともに、そのメソッドの起動を許可したいオブジェクトに同じカラーの参照をもたせ、他のオブジェクトには別のカラーの参照を持たせることにより実現することができる。

このようなケースは特に同期構造体などの場合しばしば生じるものである。そのため *OCore* では同期構造体に:In と:Out の2色からなるパレット:IO をもたせ、同期構造体に対する操作にもカラー指定を行なうことにより、誤まった操作を未然に防ぐことを可能としている。

```
1  (local (((Q-Int :In) q-in)
2          ((Q-Int :Out) q-out))
3         (set {q-in q-out} (colored-new Q-Int))
4         (qwrite q-out 7) ;; OK
5         (qread q-out))   ;; compilation error
```

3.5 単一利用性

一般にオブジェクトの参照はいったん生成されると何度でも複製可能で、何度でもメッセージを送信することが可能である。しかしながら、オブジェクトの種類によっては参照の利用回数を制限したい場合も少なくない。これには一般に重みつき参照カウントを利用することも可能であるが、ここでは参照の単一利用性とカラーの組み合わせによる部分的解決を図る。

変数のタイプ宣言に修飾子として:Use-Once をつけることにより、その変数に代入される参照の単一利用が宣言される。その参照が利用されるか、もしくは他に渡されると変数の値は UNDEF になる。参照が渡される側でも同様に単一利用の宣言がなければエラーとなる。

```
1  (local (((Foo :Use-Once) foo))
2         (set foo (new Foo))
3         (send foo [:clear])  ;; foo = UNDEF
4         (send foo [:add 7])) ;; error
```

また単にテストするだけのようなメッセージ送信では参照を利用したといえない場合もある。以下のようにメソッドを定義すれば、そのメソッドの呼び出しで参照を消費することはない:

```
1   (methods
2     ([:test (Q-Int x)] (no-ref-use) (...))
3     ...)
```

返答に用いる Q 構造体の場合にも、多くの場合返答の書き手、読み手ともに1つずつである。メッセージのフォワードなどにより Q 構造体がコピーされて、誤って複数の人が返答を読み書きすることを防ぐには例えばカラー付の参照をそれぞれ単一利用すればよい:

```
1  (local (((Q-Int :In :Use-Once) q-in)
2          ((Q-Int :Out :Use-Once) q-out))
3         (set {q-in q-out} (colored-new Q-Int))
4         (send foo [:read2 q-out])
5         (qread q-in))
```

なお単一利用を宣言した変数に代入された参照が用いられずに捨てられてしまう場合[5]は、そのオブジェクトで例外が発生する。故意に参照を捨てる場合には discard を用いる。

4 共同体

4.1 共同体とは?

共同体はオブジェクトの集合の構造化とメッセージの分散処理、効率的な実装などを目的に導入された概念である。共同体はオブジェクトの集合を、インデックスによってはられた多次元空間の中で構造化する。

共同体の実装は、オブジェクトの集合の論理構造が動的に変化するか、要素となるオブジェクトが共同体生成時に生成されるかなどで大きく異なる。例えば要素オブジェクトの集合が共同体生成時に決定・生成され、論理構造が変化しない場合は比較的単純に実装可

[5] ローカル変数に代入されたままメソッドの処理を終える場合や、変数に代入されていた参照が上書きされる場合、インスタンス変数に代入されたままそのオブジェクトがガーベッジとなったことが判明した場合など。

能である。本稿ではプログラミングの観点からみた共同体について述べるにとどめ、実装に関しては稿を改めることとする。また、以下の説明ではある要素オブジェクトが複数の共同体に属することはないものとしている。

4.2 共同体テンプレート

共同体のふるまいは共同体の要素となるオブジェクトのクラスと、共同体テンプレートと呼ばれるものによって記述される。共同体テンプレートには要素オブジェクトの集合体の論理構造と、その実プロセッサへのマッピングを決定する手続き、及び共同体生成時の初期化のためのイベントハンドラなどが記述される。

共同体テンプレートは要素となるオブジェクトのクラスをパラメトリックにして定義することが可能である。以下にその例を示す:

```
 1 (parametric-com PCom (Element)
 2   (dimension 2)
 3   (event-handlers
 4     (created ((Int sx sy))
 5       (returns (PCom Element))
 6       (...))
 7   (procedures
 8    (:element ((Int x y))
 9              (returns Element)
10              (...))
11    (:left ((Int x y))
12           (returns Element)
13           (...))
14    ...))
```

1行目は要素オブジェクトのクラスがパラメトリックで、それをタイプ変数 Element で記述することを表し、2行目ではこれが 2 次元に構造化された共同体を作るためのテンプレートであることを示している。created は共同体のインスタンスを生成するときに全 PE の上で実行されるイベントハンドラであり、共同体の各次元のサイズを引数にとって、共同体のマッピングのための情報の設定や、必要ならば要素オブジェクトの生成・初期化を行なう。一方、procedures の中で定義しているのは共同体手続きと呼ばれるものである。共同体インスタンスを知る任意のオブジェクトは共同体手続きを自らのスレッドで呼び出すことが可能である。:element は共同体の要素オブジェクトの参照を返すものであり、その他、共同体の論理構造に応じて:left,:right などといった類の共同体手続きを用意する。

共同体テンプレートの記述は共同体の実装に大きく依存する。しかしながら、共同体テンプレートは要素オブジェクトのクラスをパラメトリックに定義できるので、多くの場合プログラマはライブラリ化されたパラメトリックな共同体テンプレートを利用するだけですむと考えられる。パラメトリックな共同体テンプレートを具体化して共同体テンプレートを得るには以下のように行なう:

```
(com ComGoo (PCom Goo))
```

4.3 共同体の要素オブジェクトの定義

共同体の要素オブジェクトのクラス定義では通常のクラス定義に加えて、どの共同体テンプレートを利用するかを記述する。

```
1 (class Goo
2   (belongs-to ComGoo)
3   ...
4 )
```

これによってそのクラス定義中で以下の擬変数、オペレータが利用できるようになる:

com 属する共同体インスタンス

dimension 属する共同体の次元数

(size-of-dim *n*) 属する共同体の *n* 次元目のサイズ

(index *n*) 共同体の中での *n* 次元目のインデックス

(barrier) 同一共同体に属する (すでに生成された) すべての要素オブジェクトの間でバリア同期をとる

(global-*op* *expression*) 同一共同体に属する (すでに生成された) すべての要素オブジェクトの間でグローバルオペレーションを行なう (ex. global-add)

(com-get *instance-variable-name* *element*) 同一共同体に属する他の要素オブジェクトのインスタンス変数の値を読み出す[6]

また、共同体テンプレートで定義されている共同体手続きはクラス内からは以下のようにして呼び出すことができる:

```
(call com (:left (index 0) (index 1)))
```

[6] これを通常のメッセージパッシングで行なおうとすると、グローバルオペレーションでサスペンドしている要素オブジェクトに読みだし要求のメッセージが到着することによるデッドロックの可能性がある。

共同体の要素オブジェクトクラスの定義例とその利用例を以下に示す:

```
 1 (class Goo
 2   (belongs-to ComGoo)
 3   (vars (Int v :init 0 sum))
 4   (methods
 5     ([:add (Int x)] (set v (+ v x)))
 6     ([:sum] (set sum (global-add v)))
 7     ([:read-sum (Q-Int q)] (qwrite q sum))
 8     ([:shift]
 9       (local ((Int new-v))
10         (set new-v
11           (com-get v
12             (call com
13               (:left (index 0) (index 1)))))
14         (barrier)
15         (set v new-v)))
16     ...)
17 )
18
19 (globals (ComGoo cg :init (new ComGoo 7 7)))
20
21 (main
22   (local ()
23     (send (call cg (:element 1 2)) [:add 5])
24     (broadcast cg [:shift])
25    ))
```

この例では、19行目で共同体のインスタンスを生成し、23行目では共同体手続きを利用して、その共同体のある要素オブジェクトに対しメッセージを送信している。また、共同体に対しては24行目のように、全要素オブジェクトに対して同一メッセージをbroadcastすることも可能である。このように共同体に属さないオブジェクトやスレッドからも共同体手続きを利用することにより、共同体の要素オブジェクトにメッセージ送信が可能である点が共同体の大きな特徴である。

共同体ではブロードキャストと、バリアやその他のグローバルオペレーションなどを用いることにより、データパラレル的なプログラミングを行なうことも可能である。また、共同体の要素オブジェクトを他のオブジェクト同士のメッセージ交換の場として用いることにより、抽象化、分散化、階層化などさまざまな特徴をもった通信モデルの記述も可能となる [8]。

5 検討

5.1 共同体

共同体の効率的な実装には以下のような点が密接に関係する:

i) 要素オブジェクトの集合が共同体生成時に決定されるか?

ii) 共同体生成時に要素オブジェクトがeagerに生成されるか?

iii) ある要素オブジェクトが複数の共同体に属することが可能か?

iv) 要素オブジェクトの動的再構成が可能か?

1番目はオブジェクトが共同体に自由に出入りすることを許すかどうかである。しかしながら、共同体は構造化された集合体であり、単なるグループとは自ずと用途が異なると考えられるので、要素オブジェクトの集合は共同体生成時に決定されると考えるのが妥当であろう。

2番目は共同体生成時に全要素オブジェクトを生成・初期化するか、それともメッセージが初めて到着した時点で受信オブジェクトを生成・初期化するか、という問題である。共同体全体に対してブロードキャストなどをはじめとして偏りのないメッセージパッシングが行なわれる場合は、eagerに生成した方がよいが、非常に大きい論理空間を共同体で表現し、個々の要素オブジェクトがスパースに偏りをもってアクセスされる場合はlazyに生成すべきである。lazyに生成する場合、各メッセージパッシングにおいて受信オブジェクトがすでに存在するかどうかを常にチェックする必要があり、直接メッセージを送信することはできない。

3番目は共同体のメンバーがオーバーラップする場合である。完全に一致する場合は2つを組み合わせた新たな共同体テンプレートを用いればよい。一方、部分的にオーバーラップする場合、例えば両方のグローバルオペレーションが同時に進行してデッドロックするのをどのように防ぐか、などの問題が生じる。

最後は要素オブジェクトの集合体の論理構造あるいは実プロセッサへのマッピングを動的に変更することを許すかどうかである。論理構造の動的な変更は、例えば共同体手続きで用いられるデータをグローバルに変更することで実現可能である。一方、実プロセッサへのマッピングの動的な変更は、要素オブジェクトのグローバルなマイグレーションを伴うことになり、相当のオーバヘッドが予想される。また、動的再構成を行なっている間に送信されたメッセージを再構成後の適切な要素オブジェクトに送り届けるための中継機構も必要となる。共同体全体の負荷分散の手段として、それほどのオーバヘッドに見合う効果があるかどうかはアプリケーションに依存する。

共同体の負荷分散の手段としては他に、負荷の集中している範囲に対して新たな共同体を生成し、処理を

委託しようという方法が考えられる。新たな共同体を負荷の軽いプロセッサにマッピングすれば、毎回メッセージをフォワードするオーバヘッドはあるものの、全体的な再構成と比較してデータの移動を抑えることが可能である。

5.2 関連研究

共同体と似た概念をもつ言語にCAとpC++がある。

CA(Concurrent Aggregates)[1]では通常のオブジェクトの他にrepresentativeと呼ばれる同種のオブジェクトの集合aggregateを扱う。aggregateには通常のオブジェクトと全く同様にメッセージを送ることが可能であり、それはランタイムによって適当なrepresentativeに送られた後、そのrepresentative自身がさらに適切なrepresentativeへとメッセージをフォワードする。aggregateのこのような性質は律速段階となる任意のオブジェクトをaggregateに置換していくことにより、さらに効率的なプログラムを得るというプログラミングスタイルを可能とする。しかし、逆に受信representativeが特定される場合でも常にメッセージをフォワードするオーバヘッドが入る。また、aggregate全体へのブロードキャストやグローバルオペレーションなどは存在しない。

pC++[2]はC++にcollectionというクラスを導入し、collectionの中のオブジェクトに対するデータパラレルな操作を可能としている。オブジェクトの実プロセッサへのマッピングの記述にHPF的なtemplate,alignmentの指定方法を導入し、継承をも利用した柔軟な構造化、マッピングの指定を可能としている。しかしながら、collection中の個々のオブジェクトの制御はグローバルなスレッドのもとで行なわれるだけであり、*OCore*が要素オブジェクトに制御をもたせ、必要に応じてグローバルオペレーションで同期をとるのと比較して、処理の柔軟さに欠ける。

6 おわりに

超並列オブジェクトベース言語*OCore*の概要について述べた。*OCore*では従来の並列オブジェクト指向言語の基本部分に加え、オブジェクトの集合の構造化とメッセージの分散処理、効率的な実装などを目的として共同体という概念を導入した。また、強い型づけと簡素なセマンティクス、カラーや単一利用性などの概念の導入により、並列プログラムの安全性と実行効率の向上を図っている。

今後はさらに実装面あるいはアプリケーションフィールドからのフィードバックに基づき、言語仕様を洗練するとともに、各種並列マシンへの実装にとりかかる予定である。

謝辞

本研究を行なうにあたり、多くの非常に有益なアドバイスを頂いたRWC超並列ソフトウェアワークショップおよびRWC超並列言語ワーキンググループのメンバーの方々に感謝致します。

参考文献

[1] A. A. Chien. *Concurrent Aggregates.* The MIT Press, 1993.

[2] D. Gannon. Libraries and tools for object parallel programming. In *Proc. of CNRS-NSF Workshop on Environments and Tools For Parallel Scientific Computing*, Sept. 1992.

[3] M. Maeda, H. Konaka, Y. Ishikawa, T. Tomokiyo, and A. Hori. An incremental, weighted, cyclic reference counting for object-based languages. 情報処理学会プログラミング-言語・基礎・実践-研究会, Aug. 1993.

[4] R. Nikhil and K. Pingali. I-Structure: Data Structures for Parallel Computing. *ACM Trans. on Programming Languages and Systems*, 11(4):598–639, 1989.

[5] 友清, 石川, 小中, 前田, 堀. 超並列オブジェクトベース言語OCoreにおけるリフレクション機能. 情報処理学会プログラミング-言語・基礎・実践-研究会, Aug. 1993.

[6] 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. 情報処理学会計算機アーキテクチャ研究会, 1992.

[7] 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. *JSPP'93*, pp. 87–94, May 1993.

[8] 小中, 横田, 瀬尾. 並列計算機上のオブジェクト間の間接的通信の実現について. 情報処理学会プログラミング-言語・基礎・実践-, 9(3):17–24, 1992.