プログラミング 2－8
（1995. 8. 24）

並列オブジェクト指向言語 *OCore*における共同体の拡張

小中 裕喜　友清 孝志　前田 宗則
石川 裕　堀 敦史　Jörg Nolte

技術研究組合 新情報処理開発機構 つくば研究センタ

並列オブジェクト指向言語 *OCore*で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として共同体という概念を導入し、データ並列計算やマルチアクセスデータの記述をサポートしている。しかしながら、これまではオブジェクトの集合が共同体生成時に固定されるなどの制限があった。本論文では共同体の一つの拡張として、オブジェクトの集合の動的な変化を可能とする動的共同体を提案し、その実装について報告する。

An Extention of a Community in the Parallel Object-Oriented Language *OCore*

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

Tsukuba Research Center, Real World Computing Partnership

Tsukuba Mitsui Building 16F, 1-6-1 Takezono
Tsukuba-shi, Ibaraki 305, Japan

The parallel object-oriented language *OCore* provides the notion of a community that structures a set of objects and enables distributed message handling while it can be implemented efficiently. Communities support data parallel computation as well as multi-access data. However, there were some restrictions on the set of objects in a community; for example, the set was fixed upon community creation. In this paper, we propose an extension of a community, called a *dynamic community*, that enables the set of objects to change dynamically. We also describe the implementation of dynamic communities.

1 はじめに

われわれは並列アプリケーションのプログラマビリティの向上をめざして、並列オブジェクト指向言語*OCore*を提案している [9]。*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、効率的な実装をサポートする共同体
- アルゴリズムの記述と最適化や資源管理などに関する記述との分離を容易にするメタレベルアーキテクチャ
- **on-the-fly グローバル GC**[4]

などが導入されている。

特に共同体はデータ並列計算やマルチアクセスデータの記述のためのアブストラクションを提供するものであり、プログラマは共同体を用いて並列性を encapsulate しながら、マルチコンピュータ上で効率的に実行される大規模な並列プログラムを構築していくことが可能となる。しかしながら、これまではオブジェクトの集合が共同体生成時に固定されるなどの制限があった。本論文では共同体をより柔軟な枠組とするために、オブジェクトの集合の動的な変化を可能とする一つの拡張を提案する。

2節では*OCore*の概要について述べる。また、従来の共同体における制限と実装の概要について説明する。3節では拡張された共同体について述べる。また、Paragon XP/S[1], CM-5[7] などをターゲットとした*OCore*のプロトタイプ言語処理系の上での実現とその予備評価についても言及する。

2 *OCore*の概要

プログラムの debuggability と実行効率の向上のため、*OCore*は静的に型づけされた言語となっている。ユーザはシステムが提供する整数、アトムなどのアトミックデータや、配列、同期構造体などの構造データに加えて、ユーザ定義のオブジェクトや共同体を用いてプログラミングを行なう。

2.1 オブジェクト

*OCore*におけるオブジェクトは、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り、そのふるまいはクラスによって記述される。クラスにはスロット、メソッド、ブロードキャストハンドラ、ローカル関数およびメタレベルのプログラム要素が定義される。単一継承およびパラメトリック定義がサポートされており、クラス定義の再利用性が向上している。メソッドやローカル関数，及び 2.2節で述べるブロードキャストハンドラの処理はベースレベルで行なわれる。メタレベルの定義ではオブジェクトの状態遷移や例外発生などに対応して起動されるハンドラなどが記述される。メタレベルアーキテクチャの詳細に関しては [8] を参照されたい。

オブジェクト内の concurrency はベースレベルでは存在しない。1つのオブジェクトにおけるメッセージ処理は逐次的である。メソッドには同期、非同期の2種類があり、オブジェクト間のメッセージパッシングも、対応するメソッドによって同期か非同期かが決定される。同期型のメッセージパッシングでは送信オブジェクトの処理はリプライが戻るまで中断されるが、非同期型の場合は送信オブジェクトの処理はそのまま続行される。また同期構造体 [5, 6] を用いた柔軟な同期・通信も可能である。

2.2 静的共同体

共同体はオブジェクトの集合を多次元論理空間の中で構造化するものであり、データ並列計算やマルチアクセスデータの記述とその並列計算機上での効率的な実行をサポートする。共同体のインスタンスに属するオブジェクトはメンバオブジェクトと呼ばれ、そのクラス定義ではメンバオブジェクト特有の操作を利用できる。

共同体の実装はメンバオブジェクトの集合の性質により大きく異なる。従来の共同体は以下の制限のもとで実現されていた:

R1 メンバオブジェクトの集合は共同体生成時に生成され、固定される。

R2 メンバオブジェクトは同一クラスのオブジェクトでなければならない。

R3 メンバオブジェクトは各実行ノードに少なくとも1つ以上存在しなければならない。従ってメンバオブジェクトの総数は実行ノード数以上となる。

R4 メンバオブジェクトの集合は共同体の論理空間の上に1対1に対応していなければならない。

R5 各メンバオブジェクトは同時に複数の共同体に属することはできない。

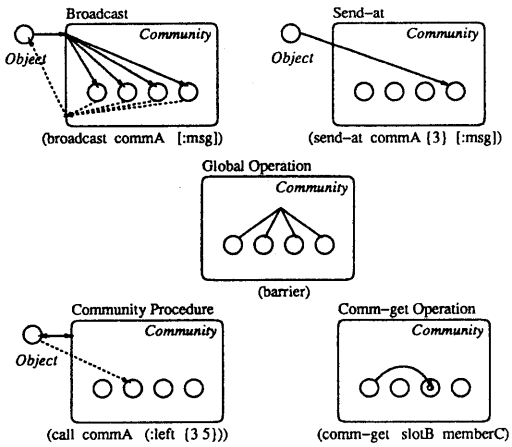

図 1: 共同体に関する操作

以上のような制限をもつ従来の共同体を本論文では**静的共同体**と呼ぶことにする。これに対し、3節で拡張される共同体を**動的共同体**と呼ぶことにする。

図 1に共同体に関するいくつかの操作の概念図と記述例を示す。これらは動的共同体にも共通しているものである。以下に、共同体に関する記述と各操作の概要を説明する。

2.2.1 共同体テンプレート

共同体インスタンスのふるまいはメンバ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のベースレベルの記述には、メンバオブジェクトのクラス、論理空間の次元数、共同体手続きなどがある。共同体手続きはメンバオブジェクトはもとより、共同体インスタンスを知る任意のオブジェクトが利用可能な手続きであり、主に共同体の論理空間を記述するのに用いられる。

またメタレベルでは、共同体の論理空間と実行ノードの間のマッピング関数や共同体インスタンス生成時の初期化のためのハンドラなどが記述される。それぞれ省略時にはデフォルトのものが用いられる。

共同体テンプレートにおけるメンバオブジェクトのクラスはパラメトリックに定義することが可能である。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が、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ことになる。

2.2.2 共同体に対する操作

共同体インスタンスを知る任意のオブジェクトは以下の操作を行なうことができる:

- 共同体インスタンスの全メンバオブジェクトにメッセージをブロードキャスト (`broadcast`) する。
- あるインデックスに対応したメンバオブジェクトに対しメッセージを送信 (`send-at`) する。
- 共同体インスタンスの提供する共同体手続きを呼び出す (`call`)。

2.2.3 メンバオブジェクトにおける操作

各メンバオブジェクトは自分の属する共同体インスタンスや、その次元数、サイズ、自分のインデックス、あるインデックスに対応したメンバオブジェクトを知ることが可能である。また共同体における唯一の代表メンバオブジェクト、ノードごとの代表メンバオブジェクトを知ることができる。

同一の共同体インスタンスに属するメンバオブジェクトの間ではバリア同期やリダクションなどの大域的操作が可能である。リダクションでは各メンバオブジェクトからの値の総和や最大値などを求めることができる。また、明示的なメッセージパッシングを行なうことなく他のメンバオブジェクトのスロットに効率的にアクセスできる `comm-get` という操作も利用可能である。

`send-at` を利用したメンバオブジェクト単体へのメッセージパッシングは、通常のオブジェクト間のメッセージパッシングと同様に行われる。一方、`broadcast` されたメッセージは各メンバオブジェクトにおいて、メソッドではなくブロードキャストハンドラによって処理される。ブロードキャストハンドラにも同期、非同期の 2 種類がある。各オブジェクトにおける処理はメソッドの処理と類似しているが、同期通信におけるリプライの処理は大きく異なる。同期型のブロードキャストハンドラでは、リプライを行なう前に全メンバオブジェクトの間でバリア同期が暗黙的に行なわれ、その後ある 1 つのメンバオブジェクトだけがリプライを行なう。また、バリア同期の代わりにリダクションを行ない、その結果をリプライすることも可能である。

2.2.4 静的共同体の実装の概要

静的共同体の生成は以下のように行なう。まずノードごとのメンバオブジェクトの生成や大域的操作のサポートを行なうローカルマネージャを各ノードに生成

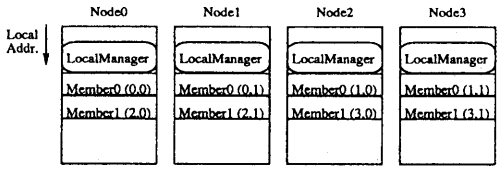

図 2: 静的共同体におけるアロケーション

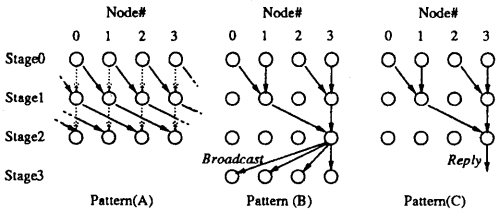

図 3: 静的共同体における大域的操作用通信パターン

する。各ローカルマネージャは、そのノードにマッピングされる共同体の部分論理空間に対応するメンバオブジェクトの生成と初期化を行なう。このとき各メンバオブジェクトにはインデックスとローカルマネージャのアドレスが渡される。最後に大域的操作のために必要な初期化を行なって各ノードの処理を終了する。

大域的操作や comm-get などを効率的に処理するため、静的共同体においてはローカルマネージャ、および同じローカルインデックスをもつメンバオブジェクトは、図2のように、各ノードにおいて同じローカルアドレスをもつようにアロケートされる。このようなメモリアロケーションの実現のため、各ノード上のメモリ空間の一部はグローバルに管理されている。しかしながら、この実装は静的共同体の制限 **R2** をもたらしている。

共同体における大域的操作では、まずノードごとにローカルな処理を行なった後、ノード間の大域的操作を行ない、必要ならば結果を各メンバオブジェクトに返す。ノード間の大域的操作には図3に示す3つの通信パターンが用意されている。一般には (A) または (B) が用いられ、大域的操作のレイテンシが隠蔽可能かどうかに応じてメタレベルの記述により選択される。また共同体インスタンスにおいて同時に実行されることがない同期型ブロードキャストハンドラにおけるリプライでは、リプライするメンバオブジェクト以外は同期の成立を知る必要がないので、(C) を用いることができる。制限 **R3** によって大域的操作に関するローカルマネージャの初期化は大いに簡略化されている。

broadcast においては、メッセージがある程度大きい場合、それを全メンバオブジェクトにコピーするのは非効率的である。そこで、ノードごとに 1 つだけコピーを置き、各メンバオブジェクトにはそのコピーの利用を示す短い特別なメッセージを渡している。コピーにはローカルなメンバオブジェクトの総数が参照カウントとしてセットされ、全メンバオブジェクトが処理を終えた時点で領域が解放される。

send-at や comm-get ではマッピング関数を用いてローカルにメンバオブジェクトのグローバルポインタを計算してからメッセージパッシングやローカル/リモートメモリアクセスを行なう。

2.2.5 共同体プログラミング

broadcast と大域的操作を用いることにより、データ並列プログラミングを行なうことが可能である [3]。また、共同体をオブジェクト間の通信媒体として用いることにより、場のような概念の記述 [2] や、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が可能となる [10]。

3 動的共同体

本節では従来の共同体の 1 つの拡張として動的共同体を導入する。2.2節に述べた制限がいかに緩和されるか、またそれがいかに実現されるかを述べる。

3.1 動的共同体の構成

メンバオブジェクトの集合の動的変化が問題となるのは、それが broadcast や大域的操作と相容れない点である。メンバオブジェクトの追加/削除が無秩序に行なわれると、broadcast の範囲が不明確となったり、大域的操作が不完全でデッドロックを生じたりする。

一方で、例えば保守的な分散シミュレーションなどではタイムステップごとにメンバオブジェクトの集合が変化するだけで、むしろ大域的操作が不備なく効率的に行なわれることが必要とされるケースが多い。そこで *OCore* ではメンバオブジェクトの集合をグローバルに規定する reorganize という操作を導入することにした。そして、メンバオブジェクトの追加/削除要求は随時行なうことができるが、それらは共同体に即時反映されるのではなく、reorganize が行なわれてはじめて実際に追加/削除が行なわれることとした。また、

```
1 (class S (belongs-to SC) ...)   ;; abbrev.
2 (community SC (consists-of S 1))
3
4 (class D (belongs-to DC) ...)   ;; abbrev.
5 (community DC (consists-of D 1 :dynamic)
6   (procedures
7     (:node ((Int i)) (returns Int)
8         (bit-and i max-pe))))
9
10 (main
11   (let ((SC sc) (DC dc) (Int i))
12     (set sc (new SC {100}))
13     (set dc (new DC {100}))
14     (dotimes (i 100)
15       (put-member dc {i}
16         (new D :on (call dc (:node i)))))
17     (reorganize dc)
18     ...))
```

図 4: 同型の 2 種類の共同体の構成

broadcast や大域的操作が reorganize とオーバラップしているかどうかをランタイムにチェックし、オーバラップしていなければそれらの操作が consistent な動的共同体に対して行なわれることを保証するようにした。

論理空間のサイズ、メンバオブジェクト数、およびマッピングが等しい、同型の 2 種類の共同体を構成するプログラム例を図 4に示す。

2行めおよび 5〜8行めはそれぞれ S,D をメンバクラスとする静的/動的共同体の 1 次元のテンプレート SC,DC を定義している。:dynamic は動的共同体のテンプレートであることを示す。DC では 16行めで用いられる、論理空間とノードとのマッピングを表す手続きが定義されている。

共同体の構成は 12行め以降である。静的共同体では 12行めの new だけでメンバオブジェクトの生成まで分散して行なわれる。一方、動的共同体を構成するにはまず 13行めで論理空間だけを生成し、続いてメンバオブジェクトの追加要求を行なう。15行めでは、共同体手続きを利用して同型となるように生成したオブジェクトの共同体への追加要求を行なっている。メンバオブジェクトの追加はそのオブジェクトの移動を伴うわけではないので、ユーザは必要に応じてこのようにメンバオブジェクトを分散して生成したり、あるいは負荷に応じて移動させてから追加することになる。そして最後に reorganize を行なう。

メンバオブジェクトの追加/削除に関する操作を除けば、ユーザは静的共同体と動的共同体を区別することなくプログラミングすることが可能である。

3.2 動的共同体の実現

動的共同体でも、メンバオブジェクトの集合は論理空間に 1 対 1 に対応しなければならないが、静的共同体における制限 **R4** のように論理空間の各要素にそれぞれメンバオブジェクトが対応していなければならない、というのは柔軟性に欠ける。同様に、制限 **R3** も動的共同体には強過ぎるものである。動的共同体の実現では、これらの制限を緩和しながらも、broadcast や大域的操作など共同体特有の処理の性能をできる限り低下させず、かつ reorganize 処理をスケーラブルに効率良く実装することが重要となる。以下ではプロトタイプ言語処理系で動的共同体がどのように実現されているかを説明する。

3.2.1 メンバオブジェクトの管理

まず、メンバオブジェクトの追加/削除要求の処理とその他の処理を独立して行なうため、各ノードのローカルマネージャには、部分論理空間に対応するされるメンバオブジェクトのグローバルポインタを格納する領域を 2 つ用意する。一方を**メンバ領域**、他方を**予備メンバ領域**と呼ぶ。メンバ領域に登録されたオブジェクトが共同体を構成する。メンバオブジェクトの追加/削除要求の処理は予備メンバ領域に対して行なう。この領域の最適な構造は、論理空間におけるメンバオブジェクトの密度や追加/削除の頻度などに依存するが、今回は単に部分空間と等しいサイズのグローバルポインタの配列とした。また、各ローカルマネージャにはローカルに存在するメンバオブジェクトのうち、リモートにマッピングされているものをリストにしておく。これを**輸出メンバリスト**と呼ぶ。図 5に動的共同体におけるメンバオブジェクトの管理と変更操作の概要を示す。

3.2.2 reorganize

reorganize 要求は、ノード 0 以外からであればノード 0 にフォワードされ、そこでオーバラップがチェックされる。続いてノード 0 から全ノードに送られる。その後の各ノードの処理の概要は以下の通りである:

i) メンバ領域および輸出メンバリストに登録されたローカルなメンバオブジェクトを共同体から外す。

ii) 輸出メンバリストをクリアする。

iii) バリア同期を行なう。

iv) 予備メンバ領域の内容をメンバ領域にコピーする。

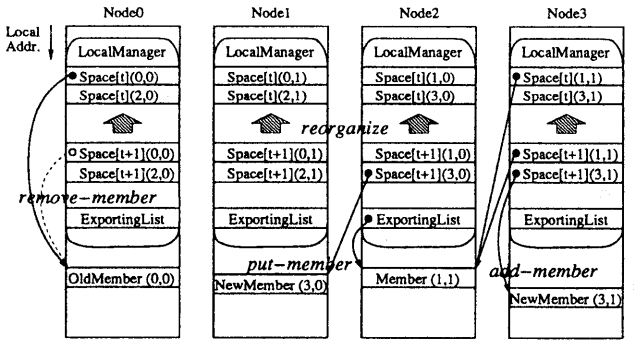

図 5: 動的共同体におけるメンバオブジェクト管理

v) メンバ領域の各メンバオブジェクトにインデックスとローカルマネージャのアドレスを伝える。

   (a) ローカルメンバであればそれをノードごとの代表オブジェクト候補とする。

   (b) リモートメンバに対しては非同期に通知する。通知されたノードではそのメンバを輸出メンバリストに登録して ACK を返す。

vi) 共同体内での上記通知の完了を知るため、リモートからのすべての ACK が集まるのを待ってからバリア同期を行なう。

vii) 大域的操作のための初期化を行ない、代表ノードが `reorganize` 完了をリプライする。

メンバオブジェクトの存在しないノードがある場合、図 3に示した通信パターンはそのままでは大域的操作に使えなくなるので、新たな通信パターンを構築しなければならない。vii)では、パターン (B) および (C) に対応した通信パターンの構築をスケーラブルに行なっている[1]。まずノード数 Nのシステムを C個のノードからなるクラスタの集合として、各クラスタで分散して通信パターンを構築し、次にそれら N/C個のクラスタをそれぞれ要素としてクラスタを構成し、分散して通信パターンを構築する、というように階層的に処理を行なっていく。要素数を Mとするとき、クラスタサイズ Cは以下のように定めている:

$$C = \begin{cases} C_{max} & \text{if } M \geq C_{max} \\ M & \text{if } M < C_{max} \end{cases} \qquad (1)$$

[1]パターン (A) に対しても同様の初期化が可能であるが、処理が複雑となるため現在はサポートしていない。

C_{max}としては現在 1 ワードのビット数 (Paragon XP/S などでは 32) を用いている。なお、図 3においてパターン (A) をステージ Sからはじめることにより、ノード番号が 2^Sを法とする剰余類に属するノードの間で大域的操作を行なうことが可能である。またパターン (B) を左右反転してステージ Sから行ない、結果をマルチキャストすることにより、隣接要素のノード番号が 2^Sはなれたクラスタにおける大域的操作を行なうことができる。これらの操作 ($G_A(S), G_B(S)$ と記すことにする) を利用して、vii)の処理は以下のように行なわれる:

1. ノード内の大域的操作完了を知るためのカウンタに、メンバ領域および輸出メンバリストに登録されたローカルに存在するメンバオブジェクトの総数をセットする。総数が 1 以上のノードをメンバ存在ノードと呼ぶことにする。

2. メンバ存在ノードにおいて、ノードごとの代表オブジェクトを決定する。v)(a)で候補があがっていればそれを用いる。なければ、輸出メンバの 1 つを代表とする。

3. クラスタレベル $L = 0$とし、ノードをクラスタ要素、ノード番号をクラスタ要素番号とする。

4. クラスタサイズを式 (1) で定める。クラスタ要素番号をクラスタサイズで割った余りをクラスタ内番号とする。

5. $G_A(L)$ でバリア同期を行なう。$L = 0$であれば、単なるバリアの代わりにリダクションによって最小メンバ存在ノード上の代表オブジェクトを求め、それを共同体全体の代表オブジェクトとする。

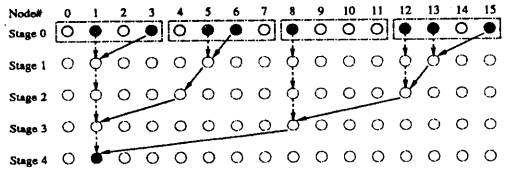

図 6: 動的共同体における大域的操作用通信パターン

6. クラスタ内番号が 0 以外であれば終了。0 であればクラスタ内のメンバ存在クラスタ要素の分布をビットパターンとしてクラスタ内の各要素にマルチキャストする。マルチキャストされた各クラスタ要素では:
   (a) 分散してクラスタ内の大域的通信パターンを構成する。
   (b) $G_B(L)$でクラスタ内のバリア同期を行なう。
7. $L = L + \log_2 C_{max}$
8. $L < \log_2 N$であればクラスタ内番号 0 のものを新たにクラスタ要素として 4へ。
9. ノード番号が 0 以外であれば終了。0 であれば:
   (a) 最小メンバ存在ノードが存在してそれが 0 でなければ、そのノードに同期成立ノードであることを通知する。通知されたノードは `reorganize` の完了をリプライする。
   (b) そうでなければ単に `reorganize` の完了をリプライする。

6(a)の詳細は省略するが、基本的にはクラスタ内のメンバ存在ノードを二分しながら、クラスタ内の同期成立がそのクラスタ中の最小ノードで行なわれるように、(B) および (C) のための通信パターンを構築していく。ただし大域的な同期成立は最小メンバ存在ノードで行なわれるようになっている。$N = 16, C_{max} = 4$のときの通信パターンの構築例を図 6 に示す。黒丸がメンバ存在ノードである。右半分に見られるように必ずしもバランスのとれたパターンを構築するわけではないが、分散して構築していくためノード数の増大にも対応することができる。

3.2.3 その他の動的共同体に対する操作の実現

メンバオブジェクトの追加/削除要求の処理は予備メンバ領域に対して行なうが、その高速化のためにビットテーブルが用いられている。静的共同体とはメンバオブジェクトの管理方法が異なることにより、**R2** の制限が:

R2' メンバオブジェクトはあるクラスもしくはそのサブクラスのオブジェクトでなければならない。

と緩和され、動的共同体では heterogeneous な集合を表現することが可能となっている。

`send-at` はインデックスに対応したメンバオブジェクトのグローバルポインタをローカルもしくはリモートのメンバ領域から得て、メッセージを送信する。`broadcast` に関しては各ノードにブロードキャストされたメッセージがまた他のノードにフォワードされるオーバヘッドを避けるため、各ローカルマネージャはメンバ領域および輸出メンバリストに登録されたローカルなメンバオブジェクトに対しメッセージをコピーする。コピーが非効率な場合にメッセージの共有が行なわれるのは静的共同体と同様である。各メンバオブジェクトにおける大域的操作もそのノードのローカルマネージャを利用することにより、処理の局所化を図っている。

3.3 予備評価

以上述べた実装方式による動的共同体の基本操作の実行性能を Paragon XP/S (クロック 50MHz, 各ノードの主記憶 16MB) を用いて測定した。

静的共同体 (SC)、メンバオブジェクトがそれぞれローカルにマッピングされた SC と同型の動的共同体 (LDC)、メンバオブジェクトがそれぞれ隣のノードにマッピングされた動的共同体 (RDC) のそれぞれについて、2048 のメンバオブジェクトからなる共同体を構成する時間と `reorganize` 処理の時間を図 7に示す。

共同体は図 4と同様にして構成されているが、LDC, RDC については `put-member` の部分を分散して行なっている。動的共同体の構成ではいずれも `reorganize` の処理がかなりの時間を占めているが、LDC では大域的操作のための通信パターンの構築が支配的でノード数に対しほぼ対数的に増大しているのに対し、RDC では v)(b)の輸出メンバの登録が支配的で、ノード数に反比例しているのがわかる。また、RDC の構成では `put-member` にともなう通信オーバヘッドがかなり大きいことが示されている。

一方 `broadcast` や大域的操作ではいずれの共同体でもほとんど実行時間に差が見られなかった。例えば 64 ノードでのバリア同期は 1.4ms 程度であった。しか

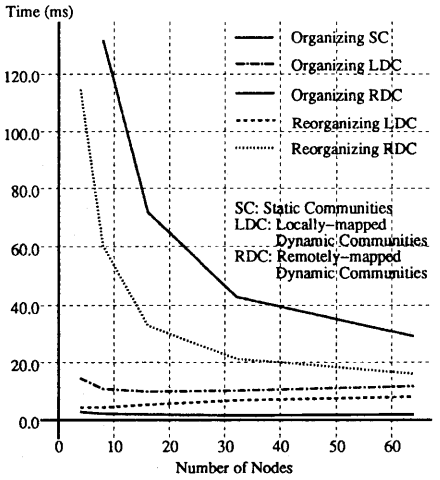

図 7: 共同体の構成および reorganize の実行時間

し、均等に分散されていない動的共同体では実行時間は当然長くなる。send-at においては静的共同体では常にローカルにメンバオブジェクトのグローバルポインタを計算するのに対し、動的共同体ではグローバルポインタを知るのにリモートノードとの通信が必要な場合があるため、同期型のメッセージ送信に倍程度の時間がかかることもある。

4 おわりに

共同体の一つの拡張として、オブジェクトの集合の動的な変化を可能とする動的共同体を提案し、その実装について述べた。動的共同体では従来の共同体における制限が緩和され、dynamic/heterogeneous なオブジェクトの集合を扱うことが可能となった。

今後の拡張/改良としては、メンバオブジェクトの密度が低い場合の実装の効率化、複数共同体への所属のサポート、などが考えられる。また、アプリケーション記述を通じた動的共同体の評価も行なっていきたい。

謝辞

本研究を行なうにあたり、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] Intel Supercomputer Systems Division. *Paragon OSF/1 C System Calls Reference Manual*, 1993.

[2] H. Konaka, T. Tomokiyo, M. Maeda, Y. Ishikawa, and A. Hori. A Parallel Object-Oriented Language OCore. In *Proc. of Intl. Workshop on Theory and Practice of Parallel Programming*, pp. 153–172, 1994.

[3] H. Konaka, T. Tomokiyo, M. Maeda, Y. Ishikawa, and A. Hori. Data Parallel Programming in the Parallel Object-Oriented Language OCore. In *Pre-proceedings of French-Japanese Workshop on Object-Based Parallel and Distributed Computation*, 1995.

[4] 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, 13(15):113–120, 1993.

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

[6] M. Sato, Y. Kodama, S. Sakai, Y. Yamaguchi, and S. Sekiguchi. Distributed Data Structure in Thread-based Programming for a Highly Parallel Dataflow Machine EM-4. *Proc. of ISCA 92 Dataflow Workshop*, 1992.

[7] Thinking Machines Corp. *CM5 Technical Summary.*

[8] T. Tomokiyo, H. Konaka, M. Maeda, A. Hori, and Y. Ishikawa. Meta-level Architecture in *OCore*. *OOPSLA '93 Object-Oriented Reflection Workshop*, 1993.

[9] 小中, 石川, 前田, 友清, 堀. 超並列オブジェクトベース言語 *OCore* の並列計算機上での実装. 情報処理学会論文誌, 36(7):1520–1528, 1995.

[10] 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-研究会, 9(3):17–24, 1992.