OCore における行列共同体の提案と実装

御手洗 潔[†]　　田中 二郎[‡]
[†] 筑波大学 理工学研究科　mitarai@softlab.is.tsukuba.ac.jp
[‡] 筑波大学 電子・情報工学系　jiro@is.tsukuba.ac.jp

OCore 共同体のデータ並列性の記述能力に注目した場合、粒度の大きさがメンバオブジェクト単位で決定されるので、行列演算を中心とした定型的な科学技術計算を複数のプロセッサを用いて記述する場合には、もう少し細かい粒度の並列性を実現したい。本論文では、行列演算の記述の容易性と実行の効率化をめざして、4つ組 +1 構造による行列共同体を提案し、実装及びその評価について報告する。

An Implementation of the Matrix Community in *OCore*

Kiyoshi MITARAI[†]　　Jiro TANAKA[‡]
[†]Master's Program in Science and Engineering
[‡]Institute of Information Sciences and Electronics

University of Tsukuba
Tennoudai 1-1-1, Tsukuba-shi, Ibaraki-ken, 305 Japan

In case of thinking about description capability of data-parallelism due to *OCore* community, we want to realize finer granularity of parallelism in describing scientific calculations, mainly involving matrix operations. In this paper, our aim is to easily describe matrix operations and make the execution more efficient. We propose the *Matrix Community using Quartet+1 Structure* and implement it in *OCore*.

1　はじめに

近年、VLSI 技術の発展とともに、数百 ~ 数千個の高性能なプロセッサを高速ネットワークで結合した、分散メモリ型マルチコンピュータシステムが商用化されつつある。こうしたシステムで実用的な並列アプリケーションを記述するために、多種多様の並列プログラミング言語が提案されている。

われわれは、簡素で拡張性の高い並列オブジェクト指向言語として現在、RWCP (新情報処理開発機構) で開発が進められている *OCore* に注目している [5, 6, 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て、以下の特徴などを導入している [9]。

- オブジェクトの集合の構造化と通信の分散、効率的な実装を目的として「共同体」という概念を有する
- アルゴリズムの記述と、資源管理、実行最適化、例外処理などの記述の分離を可能にする「メタレベルアーキテクチャ」を有する
- 「グローバル Garbage Collection (on-the-fly) [4]」を有する
- 同期構造体を用いた柔軟な同期、通信を可能とする

OCore によるプログラミングではアプリケーション内の定型的な並列性 (データ並列) を共同体で、非定型的な並列性 (コントロール並列) をオブジェクトにより記述することが可能である。

しかしながら、共同体のデータ並列性の記述能力に注目した場合、粒度の大きさがメンバオブジェクトの大きさによって決定されることになる。したがって、内在する並列性をアプリケーション毎に的確な

記述を行なうには共同体の個々のメンバオブジェクトを構造化する必要がある。

数学の分野では行列、計算機科学の分野では配列構造といった概念がある。これらの概念はオブジェクト指向計算に基づく言語 *OCore* では共同体及びメンバオブジェクトという形で実現している。

このような関係を考慮すると、アプリケーションとして行列演算を適用することはごく自然な流れであると考える。

本論文では、行列数値演算の記述の容易性と実行の効率化をめざして、共同体を用いた行列演算の並列実装のための行列共同体を提案し、その実装、評価について述べる。

2 共同体の概要

2.1 共同体

共同体は *OCore* のオブジェクトにはないメッセージ処理の並列性を提供する。共同体はオブジェクトの集合をインデックスの任意の n 個の組によって一意に決定可能な n 次元空間 ($n \geq 1$) に配置し、管理するための枠組みである。また、共同体と似た概念をもつ言語に CA (Concurrent Aggregates) [1] 等がある。いずれもメッセージの分散、その並列処理環境での効率的な実装を目的として導入された概念である。

現在、 *OCore* では静的共同体と動的共同体 [10] が実装されている。静的共同体はメンバオブジェクトが同種で固定されるなど、柔軟性に欠けるが処理速度に優れている。一方、静的共同体の制約を若干ゆるめた動的共同体は継承関係にある異種のメンバオブジェクトとの共存、メンバオブジェクトの動的な追加、削除などが可能であり、柔軟性に優れているが、処理速度に劣る面がある。

2.2 共同体のメンバオブジェクトへの操作

各メンバオブジェクトは各々の属している共同体インスタンス、その次元、サイズ、共同体内インデックスを知ることができる。また、同一共同体インスタンスに属する全メンバオブジェクト間でのバリア同期や各メンバオブジェクトのスロット値についてのリダクション (和、積、最大値、最小値等の計算) などの大域的操作が定義されている。

さらに、図 1のように同じ共同体インスタンスの任意のメンバオブジェクト間であれば明示的にメッセージパッシングを行なうことなく、 *comm-get* や *comm-get-at* という命令を使用して他のメンバオブジェクトのスロット値に対して効率的にアクセス可能である。

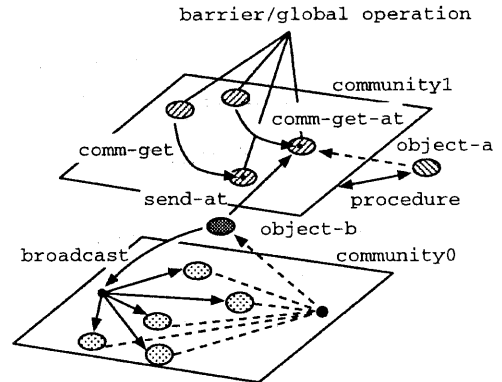

図 1: 共同体のメンバオブジェクトへの操作

3 従来の *OCore* での行列演算

3.1 *OCore* での配列の扱い

OCore において、行列の基本的な演算を行なうための構造データ型として、言語に組み込まれた **Array-of** と **Vector-of** を利用することになる。各々数値タイプに限れば **Int**、 **Float**、 **Double** の 3 種類が利用可能である。双方の主な特徴は以下の通りである [6]。

- **Array-of**
  - 1 次元の配列に限定
  - 多次元の配列を構成するには typedef などを使用して構造化が必要
  - グローバル変数、ローカル変数、インスタンス変数、テンポラリ変数として使用可能
  - 一般的に **Vector-of** よりアクセススピードが遅い
- **Vector-of**
  - 多次元の配列が使用可能
  - ローカル変数、インスタンス変数、テンポラリ変数として使用可能 (グローバルでは使用不可)
  - 一般的に **Array-of** よりアクセススピードが速い

3.2 配列を用いたプログラム例

2つの行列の乗算を行なうプログラムを考える。まず、構造データ型として **Array-of** と **Vector-of** の2種類が利用可能である。それから、行列自体をスロット(インスタンス変数)、ローカル変数、またはグローバル変数として定義するかで3種類から選択できる。すなわち、全部で5つの組み合わせが考えられる。本論文では、速さとデータの独立性を考慮してスロットを **Vector-of** とした例を図2に示す。

```
(class Matrix
 (vars ((Vector-of Float {1024 1024}) mat-a mat-b
                                           mat-c))
 (methods
  ([:mul (Int m k n)] (replies Void)
    (let ((Int i l j) (Float wrk))
      (dotimes (i m)
       (dotimes (j n)
        (set wrk 0.0f)
        (dotimes (l k)
         (set wrk (+ wrk (* (get-at mat-a {i l})
                            (get-at mat-b {l j})
                                          ))))
        (put-at mat-c {i j} wrk)))))))
```

図 2: 行列の乗算を行なうプログラム例

このプログラム例ではクラス Matrix が存在する PE (プロセッサエレメント) のみが2つの行列 (mat-a, mat-b) の乗算を行ない、結果を mat-c にセットする。したがって、クラス Matrix を共同体のメンバオブジェクトとした場合にも各々行列が独立に定義される。そして、個々に計算されることになり、複数の PE が協力して行列の乗算を行なうことはできない。

4 行列共同体の定義

われわれは、行列 (ベクトル) の基本演算等を並列に行なうために行列共同体を定義し、その実装のために4つ組 +1 構造を考案した。行列共同体は以下のように定義できる。

まず、与えられた正方行列 A を行方向、列方向ともに等しく等間隔になるように区切る。そして、それぞれの部分行列 (以後、これをブロック行列 [8] と呼ぶ) を成分とする元の行列のブロック化行列を考える。さらに、図3のようにブロック行列から2次元共同体のメンバオブジェクトへのマッピングを行列共同体と定義する。

このようにして行列と共同体の対応を考えることにより、行列演算に内在しているデータ並列性を共同体を使って記述できる。

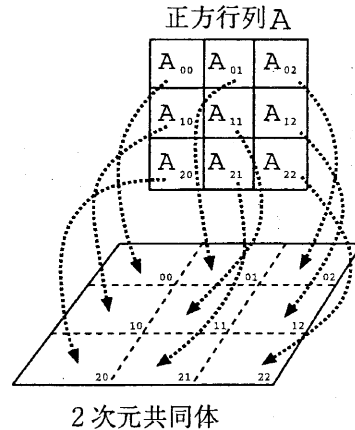

図 3: 正方行列 A の2次元共同体へのマッピング

5 行列共同体の実装の概要

行列共同体の本実装では、処理効率を考慮して静的共同体を使用することにした。以下に4つ組 +1 構造による行列共同体の実装の概要を述べる。

5.1 4つ組 +1 構造

4つ組 +1 構造の行列共同体での外観は図4に示すように、行列共同体の各メンバオブジェクトにブロック行列と同等のサイズ× 4+1 の領域を確保したものである。図5に行列共同体のメンバオブジェクトの (i j) 成分の4つ組 +1 構造を拡大したものを示し、行列の値がどのようにセットされるか説明する。

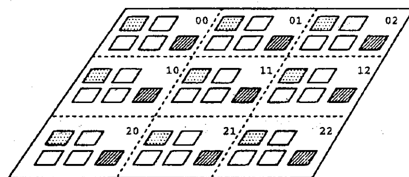

図 4: 行列共同体上の4つ組 +1 構造の外観

通常、行列共同体を生成し、データセットメソッドにより正方行列 A をセットしたとすると図5に示すように、4つ組の左上に正方行列 A の (i j) ブロック行列成分がセットされる。次に4つ組 +1 構造の中でセットされたデータ (左上の部分) 以外の構成要

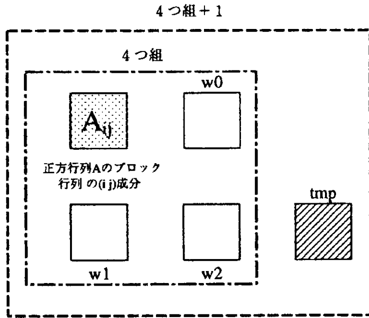

図 5: 行列共同体の (i j) 成分の 4 つ組 +1 構造

素 w0〜w2 は行列の乗算を行なうときに最低限必要な領域である。また、 tmp は他の行列共同体からのデータ転送、計算が同時に行なわれる場合データ転送のためのダブルバッファリングの領域として使用する。

さらに、加算などの行列演算の場合、計算前のデータを保存したいときに tmp 領域を以前のデータの待避領域として使用する。したがって、行列共同体の本実装では 4 つ組 +1 構造が必要となる。

5.2 4 つ組 +1 構造の操作

行列共同体のメンバオブジェクトの 4 つ組 +1 構造における操作は以下のように分類することができる。

- 他の行列共同体のメンバオブジェクトの 4 つ組 +1 構造間での操作
- 自分自身の行列共同体の他のメンバオブジェクトの 4 つ組 +1 構造間での操作
- 任意のメンバオブジェクトの 4 つ組 +1 構造内での操作

それぞれデータのコピー、スワップ、和、差、積などの行列の基本演算を行なうことができる。

5.3 行列共同体の実装の基本方針

行列共同体上の 4 つ組 +1 構造を用いて各種行列演算を記述する際には、ブロードキャストにより全てのメンバオブジェクトで定義されたブロードキャストハンドラが同時に起動されることが基本となる。

したがって、各々のメンバオブジェクのブロードキャストハンドラ内の共通したエラー処理や各メンバオブジェクトに共通した処理を全てのメンバオブジェクトで行なうことは無駄である。そこで、この無駄な処理をなくすためにプログラム中の全ての場所から呼び出せるユーザ定義可能なグローバル関数を使用することにした。グローバル関数はエラー処理などの共通処理部とブロードキャストハンドラの起動部から構成される。

このことにより、一つのブロードキャストハンドラだけでは記述できない処理の記述や API (アプリケーションプログラムインタフェース) に柔軟性を持たせることも可能になった。

5.4 行列共同体上の行列乗算アルゴリズム

行列共同体上の 4 つ組 +1 構造を用いた行列の乗算アルゴリズムを説明する。行列共同体を用いた行列の乗算は行列共同体の列ベクトル単位の積の計算を列ベクトルの個数分行なうことにより実現している。図 6に 3 × 3 の場合の行列共同体の最初の列ベクトル部分の積が求まる様子を示す。

ステップ 1 では行列共同体 B の最初の列ベクトルの値を行列共同体 A の各行ベクトルの w0 にコピーする。ステップ 2 では行列共同体 A の全ての成分で $w1_{ij} = A_{ij} \times w0_{ij}$ を行なう。ステップ 3 では行列共同体 A の第 0 列ベクトルの $w2_{i0}$ へ行方向に第 0〜第 2 列ベクトルの $w1_{ij}$ を足し込む操作を行なう。

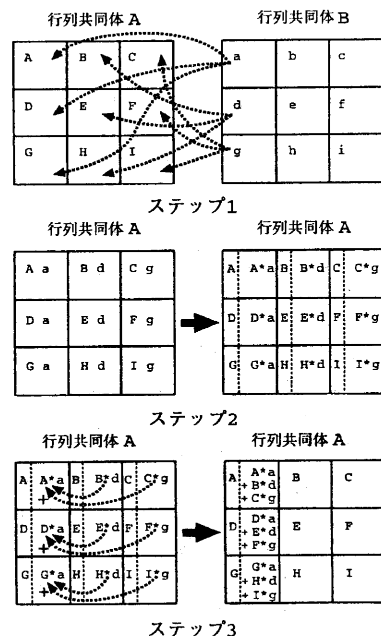

図 6: 行列共同体上の行列の乗算

6 行列共同体の性能評価

以上に述べた 4 つ組 +1 構造による行列共同体の実装に対する基本演算の一部の実行性能について、Intel Paragon XP/S [3] を用いて評価を行なった。

6.1 評価条件

本性能評価で使用したハードウエア、ソフトウエア環境は以下の通りである。

- Intel Paragon XP/S (RWCP つくば研究センター)
  - 計算ノードの最大使用可能数は 64 (現在 2 のべきしか使用できない)
  - 各計算ノードには計算用と通信用の 2 つの i860XP (50MHz) を搭載
  - 各計算ノードの主記憶の大きさは 16MB
  - 各計算ノード間の通信には NX ライブラリを使用
- OS は OSF/1 Release 1.0.4 Server 1.3 R1_3
- *OCore* の処理系は Release 0.9.9
- Garbage Collection は未使用

6.2 基本演算の評価

行列共同体の基本演算の評価は行列共同体間でデータの移動がない評価と行列共同体間でデータの移動をともなう評価に分けて行なった。それぞれ共同体にマッピングする行列の大きさ (1024×1024) を固定し、ブロック行列の大きさ及び実行時の計算ノードを変化させたときの実行時間を測定した。双方とも計算ノード数 8 の場合の数値を表 1、表 2にまとめた。

6.2.1 データの移動がない評価結果

行列共同体間でデータの移動がない基本演算として本実装では、単位行列生成、ゼロ行列生成、行列のスカラー積、行列のトレース等がある。本論文では行列のスカラー積の場合の結果を図 7に示す。

本評価ではブロック行列の大きさ (16×16, 32×32, 64×64) を変えながら計算ノード数 2~64 の実行時間を各 3 回測定し、平均をとった。例えば、1024×1024 の行列を 64×64 のブロック行列を用いて実行すると 16×16 = 256 要素を持つ行列共同体で実行することになる。但し、測定時間は演算のみの時間であり、プログラムの起動、 *OCore* のランタイムシステムの初期化及び行列共同体の生成時間 (およそ 20~40 sec) は含まれていない。

この図からわかることは、各計算ノード数でのブロック化の効果はあまり顕著ではなく、計算ノード数が 16 以上はほとんど効果が見られない。結果的に、行列の大きさ (1024×1024) でのリーズナブルな計算ノード数は 8、 16 であるといえる。

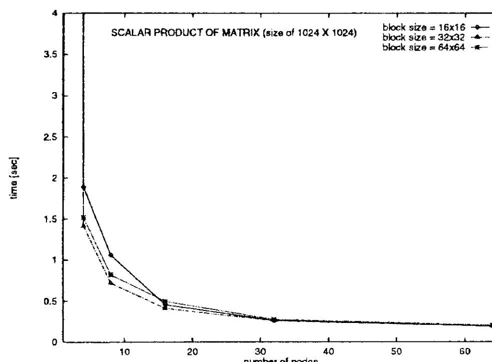

図 7: 行列共同体のスカラー積

表 1: 計算ノード数 8 の実行時間 (スカラー積)

ブロック行列の大きさ (行列共同体の大きさ)	実行時間 (sec)
16×16 (64×64 = 4096)	1.06
32×32 (32×32 = 1024)	0.72
64×64 (16×16 = 256)	0.82

6.2.2 データの移動をともなう評価結果

行列共同体間でデータの移動をともなう基本演算として、他の行列のコピー、他の行列とのスワップ、行列の加算、減算、乗算がある。本論文では行列の要素のコピー、乗算、加算が必要となる行列共同体どうしの乗算の場合の結果を図 8に示す。

本評価ではブロック行列の大きさ (32×32, 64×64, 128×128) を変えながら計算ノード数 4~64 の実行時間を各 3 回測定し、平均をとった。データの移動をともなわない評価の場合と同様に、測定時間は演算のみの時間である。

本評価の場合には、計算ノード数 4~16 ではブロック化の効果が現われているといえる。しかしながら、計算ノード数 32 あたりからブロック化の効果が薄れてきている。結果的に、行列の大きさ (1024×1024)

でのリーズナブルな計算ノード数は 32 であるといえる。

また、図 2の行列共同体を用いない行列の乗算プログラムを計算ノード 1 で実行した結果が約 3800 (sec) であることを考えると行列共同の本実装は概ね良好であるといえる。

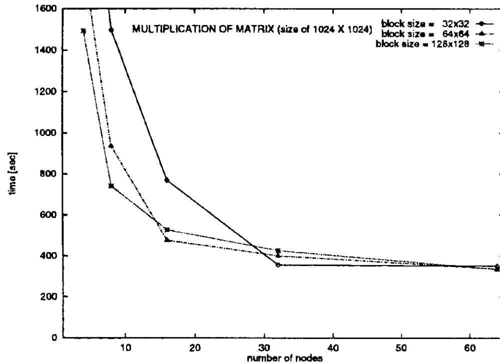

図 8: 行列共同体の乗算

表 2: 計算ノード数 8 の実行時間 (乗算)

ブロック行列の大きさ (行列共同体の大きさ)	実行時間 (sec)
32×32 (32×32 = 1024)	1498.99
64×64 (16×16 = 256)	935.67
128×128 (8×8 = 64)	740.96

7 おわりに

本論文では、並列オブジェクト指向言語 *OCore* における 4 つ組 +1 構造による行列共同体の実装方式及び行列共同のスカラー積、乗算による性能評価について記述した。本評価の結果から有効なブロック行列の大きさの範囲や計算ノード数との関係の傾向が明らかになり、4 つ組 +1 構造による行列共同の実装の妥当性を示すことができたと考える。

また、Paragon、CM-5 [7] 等の相次ぐ生産中止という時代の変化の中で今後注目される PVM [2]、ワークステーションクラスタ等を用いた分散環境での行列共同体の評価や実アプリケーションへ応用した場合の評価をふまえた行列共同体の改良を行なう予定である。

謝辞

本研究を行なうにあたり、日頃より指導、アドバイスを頂いている RWCP 筑波センターの石川 裕、小中 裕喜の両氏に感謝致します。

参考文献

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

[2] Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R. and Sunderam, V.: *PVM: Parallel Virtual Machine – A Users' Guide and Tutorial for Networked Parallel Computing*: The MIT Press, 1994.

[3] Intel Supercomputer Systems Division: *Paragon OSF/1 C System Calls Reference Manual*, 1993.

[4] M.Maeda,H.Konaka,Y.Ishikawa,T.Tomokiyo, A.Hori,andJ.Nolte:On-the-fly global garbage collection based on partly mark-sweep, In *Lecture Notes in Computer Science*, volume 986, pages 283-296, Sep., Springer-Verlag, 1995, (Proc. IWMM'95).

[5] Real World Computing Partnership: *OCore Operation Manual*, Feb., 1996.

[6] Real World Computing Partnership: *OCore Reference Manual*, Feb., 1996.

[7] Thinking Machines Corporation: *Connection Machine CM-5 Technical Summary*, 1993.

[8] 小国 力 (編): 行列計算ソフトウエア – WS、スーパーコン、並列計算機 –, 丸善, 1991.

[9] 小中 裕喜, 石川 裕, 前田 宗則, 友清 孝志, 堀 敦史: 超並列オブジェクトベース言語 *OCore* の商用並列計算機上での実装, 並列処理シンポジウム JSPP'94, pp. 113-120, 1994.

[10] 小中 裕喜, 友清 孝志, 前田 宗則, 石川 裕, 堀 敦史,Jörg Nolte: 並列オブジェクト指向言語 *OCore* における共同体の拡張, In SWoPP'95, 1995.