並列オブジェクト指向言語mosaicの塵集め方式

村山敏清*, 屋鋪正史*, 松山浩之*, 瀧和男**
*神戸大学大学院自然科学研究科
**神戸大学工学部情報知能工学科

本研究室で開発した並列オブジェクト指向言語 mosaic に適した，新しい塵集め機構を設計し，実装した．この塵集め機構は，重みなし参照カウント法と重みつき参照カウント法を併用して一つのカウンタで管理し，回収には free-list とインクリメンタルコピーイングを用いたリアルタイム GC である．さらに，参照カウント法では回収できない循環した塵が増えた時は，参照辿り法も用いてこれらの塵を回収する．本論文では，塵集めの新方式の提案と，その性能評価を行なう．

Garbage Collection Method for Concurrent Object-Oriented Language "mosaic"

Toshikiyo MURAYAMA* Masafumi YASHIKI*
Hiroyuki MATSUYAMA* Kazuo TAKI**
*Graduate School of Science and Technology,Kobe University
**Department of Computer and Systems Engineering
Faculty of Engineering,Kobe University

We designed and implemented a new Garbage Collection(GC) method for a concurrent object-oriented language named "mosaic" which we had been developing. Our method is a real time method that consists of reference counting and weighted reference counting in combination managed by only one counter in marking phase, free-list management and incremental copying in collecting phase. When circulation pointers increase, which reference counting can not distinguish, the method uses mark-and-sweep to collect them. We propose the new GC method and evaluate its ability in this paper.

1 はじめに

並列論理型言語KL1[1]の長所を採り入れた並列オブジェクト指向言語mosaic[2][3][4]を本研究室で開発し，今回新たに塵集め機構（Garbage Collection：以下GC）を実装した．

mosaicは，KL1の長所であるノード／優先度指定機構及び暗黙の通信機構を採り入れており，又メソッド内の記述はC言語を用いることが出来るため既存の手続き型言語からも移行しやすいようになっている．そしてmosaicは，疎結合並列マシン上で多数のオブジェクトを効率良く扱えるよう作られている．

以下ではmosaicの処理系に実装したGCの方式について新方式の提案を含め報告する．

2章ではmosaicの概要と今回設計したGCについて簡単に説明する．3章ではGC方式について詳述する．4章では実装に際しての工夫を，5章ではGCを実装したmosaicの性能測定とその結果を説明する．

2 mosaic向けGCの設計方針

2.1 並列オブジェクト指向言語mosaic

mosaicは，本研究室で開発した並列オブジェクト指向言語で，mosaic言語で記述されたソースを中間言語に翻訳する翻訳系と，中間言語コードから実行可能ファイルを生成する際にリンクするライブラリ群である実行系より，構成されている．

2.1.1 mosaicの実行モデル

mosaicでは，多数のオブジェクトを効率良く動かすため，オブジェクト内は逐次実行し，各オブジェクトの実行中は一切割り込みをしないモデルを採用している．GCも，このモデルに従って設計した．

2.1.2 ストリームオブジェクト

ストリームオブジェクトは，システムが用意しているオブジェクトで，今回設計したGCの中では，その機能を拡張してノード間参照管理を行なうオブジェクトとして使われている．

ストリームオブジェクトは，ユーザー定義オブジェクトを指す間接ポインタ（ユーザーからは意識されない）のようなもので，mosaicの未生成オブジェクト機能[4]を実現するために用いられている．

ストリームオブジェクトには，メッセージのバッファリング機能があり，そして連結と言う操作が定義されている[4]．連結操作と言うのは，論理変数のユニフィケーションのようなものである．

2.1.3 オブジェクトID

全てのオブジェクト間の参照関係は，オブジェクトID（以下，単にIDと書けばオブジェクトIDのことを指す）によってのみ作られる．ノード渡りの参照になるのもオブジェクトIDだけで，GCを実現するには，IDについての参照管理が必要である．

2.1.4 オブジェクト型変数

mosaicには，オブジェクトIDを表すための型としてオブジェクト型が用意してある．

mosaicのメソッド記述にはC言語が使われており，オブジェクト型は，メソッド中で通常のC言語の型と同様に使用できる．従って，実行系はオブジェクト型変数を検出するための処理が必要である．

2.2 従来の並列向けGC方式

参照辿り法[5]や参照カウント法[5](Reference Count, 以下RC)は，疎結合並列マシン上ではノード間通信／同期が増えるため効率が落ちる．

分散GC用のマーク付けアルゴリズムとしては，次のようなものが考えられる．

1. 重みつき参照カウント法[6]（Weighted RC: 以下WRC）
2. ノード間参照管理をWRCを使った輸出入表で行ない，ノード内のマーク付けには古典的手法（例えば参照辿り法）を使う方法[6]
3. 並列向けに改良を加えた参照辿り法[6]

マーク付けさえ終れば，回収には古典的アルゴリズムでも問題は少ない．

1の方法では即時GCが可能だが，ノード内でに関しては，参照管理の処理が比較的重い．また，数万～数百万のオブジェクトを扱う際は，重みの分割ができなくなった時の処理が問題となる．

3の方法は，全ての循環した塵を回収できると言う長所を持つものの，起動／終了方法，未到着メッ

セージなど，解決すべき問題も多い．更に，他の処理に比べ遅い処理であるため，他の方式の GC と組み合わせて使う必要がある．

それに対し 2 の方法は，ノード内の GC 方式によって，その性質が大きく変わるため評価しにくいが，ノード内の方式の設計次第では効率の良い GC になると考えられる．よって今回は 2 に分類される新しい方式を設計した．

2.3 mosaic のデータ構造と GC 方式の概要

GC の対象となるデータ構造と，各々の回収方法を説明する．

2.3.1 ユーザー定義オブジェクト本体

ユーザー定義オブジェクト本体は，ユーザー定義オブジェクトに共通した情報を保持するためのデータ構造である．これには，スロット変数へのポインタが含まれる．

これは固定長なので free-list で管理する．参照管理は RC と WRC を併用した方式（3.1で説明する）で行なう．

2.3.2 ストリームオブジェクト本体

ストリームオブジェクト本体はストリームオブジェクトに共通した情報を保持するためのデータ構造である．

固定長なので，free-list で管理し，参照管理は RC と WRC を併用する点は，ユーザー定義オブジェクト本体と同じである．なお，スロット変数は持たない．

2.3.3 スロット変数領域

スロット変数領域は，ユーザー定義オブジェクトの内部状態を保持するために使う変数用の領域である．そのサイズは可変で，ユーザーが定義したクラスによって定まる．

スロット変数はただ一つのユーザー定義オブジェクト本体からのみ参照され，他のスロット変数等からは参照されない．しかも，ID は，他のオブジェクト本体を指すものであり，スロット変数を別の領域にコピーしても影響がない．参照管理は，ユーザー定義オブジェクト本体のものを使う．

以上の性質を利用して，スロット変数はインクリメンタルコピーイングで回収する．リードバリアを用いてコピーイングをインクリメンタルに実行する方式は，既に提案されているが [7]，今回使うのは，スロット変数の性質を利用した，バリアを使わない方式である．

2.3.4 メッセージ領域

メッセージのための領域で，固定長のものを free-list で管理している．この領域は，実行系が不要となる時期を容易に検出できるので，即時回収する．

2.4 マーク&スイープ

2.3で説明した方式以外に，RC では回収できない循環した塵が増えた時，それらの塵を回収するために，参照辿りを用いたマーク&スイープも用いるようにする．この場合，スロット変数の回収には，インクリメンタルなコピーイング以外に，スライディングコンパクションも用いる．

3 mosaic の GC 方式

3.1 RC と WRC の併用

疎結合並列マシン上での参照管理を（重みなし）RC で行なうと，他ノード上の参照を増減させる度にノード間でメッセージ通信と同期を必要とする．WRC ではその必要がなく，RC に比べ効率が良いことが知られている [6]．

しかし，ノード内においては，各参照毎にカウンタが必要なためメモリ消費が増えること，各カウンタの値が大きくなりがちなこと，分割が出来なくなった時の対処などの問題点がある．更に，ノード内では通信／同期も容易に行なえるため，WRC のメリットがなくなる．

以上の理由から，ノード内では (重みなし)RC，ノード間では WRC を使い，それらを一つのカウンタで管理する方法を提案する．なお，このカウンタはストリームオブジェクト本体，ユーザー定義オブジェクト本体に有り，参照カウンタと呼ぶ．

ノード内の参照管理には RC を使う．具体的には次の例のように使う．

- メッセージを送信する時，宛先及び引数中の ID のオブジェクトの参照カウンタを 1 増やし，宛先オブジェクトに到着すると 1 減らす．
- スロット変数にオブジェクトの ID を入れる時，オブジェクトの参照カウンタを 1 増やす．
- スロット変数の ID を消す時，オブジェクトの参照カウンタを 1 減らす．

- メソッドが終了した時，スロット変数中に保持しているIDのオブジェクトのカウンタを1減らす．

ノード間の参照管理にはWRCを使う．具体的には次の例のように使う．この処理のほとんどに，輸入管理オブジェクトが関係している．輸入管理オブジェクトについては4.2で説明する．

ここで言う「輸出」とは，ノード内のオブジェクトIDをメッセージに載せてノード外へ送信することであり，「輸入」とはノード外のオブジェクトIDがメッセージと共にノード内へ送られることである．

- ノード内のオブジェクトIDを輸出する時，参照数を1ではなく，M以上の値にして輸出する．オブジェクトの参照カウンタもM増やす．
- IDを輸入した時，IDと重みを輸入管理オブジェクトが保持し，メッセージには輸入管理オブジェクトのIDと重みを載せる．
- 輸入管理オブジェクトのIDを輸出しようとした時，IDと重みをノード外オブジェクトのものに載せ換えてから輸出する．
- 輸入管理オブジェクトの参照数が0になると，保持しているIDと重みを被参照オブジェクトに返却する．

この方式ならば，ノード内での処理はRC同様に簡単であり，ノード渡りでの通信／同期もWRCと変わらない．また全てWRCで管理する時より，格段にカウンタは少なくできる上，重みの分割も容易である．

3.2 回収

3.2.1 free-list管理

ユーザー定義オブジェクト本体とストリームオブジェクト本体は，free-listで管理する．

ストリームオブジェクト本体は，カウンタが0になると同時に回収する．ユーザー定義オブジェクト本体は，カウンタが0になった後，インクリメンタルコピーイングで，スロット変数と同時に回収する．

3.2.2 インクリメンタルコピーイングの起動タイミング

スロット変数領域は，可変長なのでインクリメンタルコピーイングで回収する．

mosaicでは，メソッド内では割り込みがないモデルを採用している．また，メソッドの実行中ではなくスケジューラが処理している間にコピーイングを起動するならば，auto変数の管理が必要なくなり，参照管理が簡単になる．

以上の理由で，回収はメソッド実行中でなく，スケジューラが処理している時に起動することにした．

3.2.3 インクリメンタルコピーイング

疎結合並列マシンでは，逐次マシンよりもインクリメンタルに実行する必要性が高い．そのため，通常の一括して実行されるコピーイングではなく，mosaicではインクリメンタルなコピーイングを採用した．

インクリメンタルコピーイングの状態遷移図を図1に示す．

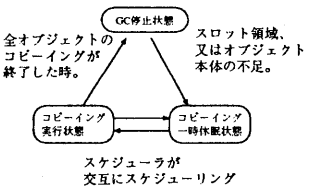

図 1: コピーイングの状態遷移図

インクリメンタルコピーイング中は，コピーイング実行状態とコピーイング一時休眠状態を交互に実行する．なお，全てのオブジェクトはGCの状態に関わらず，生成時にオブジェクト本体をalive-object-listと呼ばれる一つのリストに繋いでおく（図2）．

図1に示すように，mosaicのインクリメンタルコピーイングには三つの状態がある．次に，それらの状態について説明する．

- GC停止状態

  GCを全く起動していない状態．

  参照管理は行なうが，回収に関する処理は行なわない．新しくオブジェクトを作る時は，ユーザー定義オブジェクト本体はalive-object-listの最後に付け，スロット変数は旧領域から取る．

- コピーイング一時休眠状態

インクリメンタルコピーイングが起動されているが，スロットのコピーはせずオブジェクトのメソッドを実行している状態．

オブジェクトを生成する時は，alive-object-list の_Last_create_object （コピーイングの終った最後のオブジェクトを指しているポインタ，図2参照）の次に，オブジェクト本体を作りスロットは新領域から取る．(alive-object-list の順序とスロット変数のメモリ上での順序を一致させるため．スライディングコンパクションの時にこれが必要になる．)

それ以外のメソッドの実行は，通常と変わらない．

- コピーイング実行状態

  実際にスロットをコピーしている状態．(図2) 一回のコピーイングで，オブジェクトを n 個ずつ処理する．

```
/* コピーイング実行状態の疑似コード */

/* _Last_create_objectは，コピーイングの
   終った最後のオブジェクトを指す */
obj = _Last_create_object ;

/* n個のオブジェクトに対して */
for(  i = 0; i < n ; i++  ){
    obj = obj->next ;

    /* オブジェクトが残ってない時 */
    if( obj == NULL ){
        GC停止状態に移る ;
        return ;
    }

    if( 参照カウンタが0の時 ){
        スロット変数中の参照を解放する ;
        オブジェクト本体をfree-listにつなぐ ;
    }
    else{    /* カウンタ≠0の時 */
        スロット変数を旧領域から
            新領域へコピーする ;
        オブジェクト本体のスロット変数への
            ポインタを書き換える ;
        if( 新領域の残りが少ない時 ){
            マーク&スイープを起動;
            return;
        }
    }
}
```

この方法を使えば，リード／ライトバリアを使わずにインクリメンタルコピーイングが可能である．

3.3 マーク&スイープ

RC を用いた方法では循環した塵は回収できない．循環した塵が増えてきて回収の効率が落ちた時は，参照たどりによるマーク付けを行なう．この時は，ノード内の，GC 以外の全ての処理を中断してマーク付けを行なう．

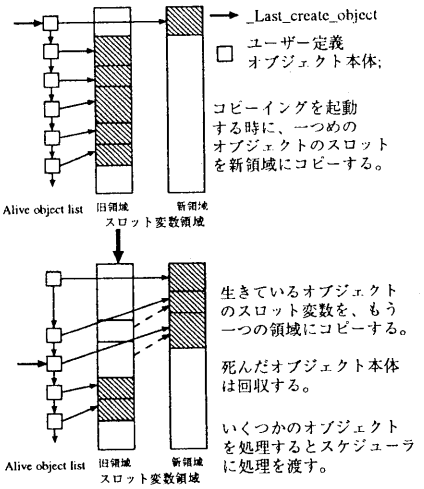

図 2: インクリメンタルコピーイング

3.3.1 参照辿りによるマーク付け

参照たどりによるマーク付けのマーキングルートは，外部から参照されているユーザー定義オブジェクト，プライオリティキューに継っているメッセージの宛先と引数中の ID，及び外部から参照されているストリームオブジェクトのフォワード先の ID とバッファリングしているメッセージ中の引数に使用されている ID である．

ノード外から参照されているオブジェクトを識別するために，ノード外に ID を輸出する時はノード内に比べ（ノード内では RC なので，1 に固定）大きな重みを追加して，ある一定値 M 以上の値にしてから輸出する．

カウンタが M 以上のオブジェクトを，外部から参照されているオブジェクトとみなして参照たどりによるマーク付けを行なえば，全ての参照されているオブジェクトはマーク付けされる．

3.3.2 回収

マーク付け後スロット変数の回収には，通常は，インクリメンタルなコピーイングを用いる．これは，3.2.3で説明したものと同じものである．ユーザー定義オブジェクト本体，ストリームオブジェクト本体は，固定長の領域なので，やはり free-list 管理する．

マーク&スイープは，インクリメンタルコピー

イングの途中からでも起動でき，その場合，マーク付け後に旧領域に残ったスロット変数の回収には，スライディングコンパクションを用いる（図3(a)）．スライディングコンパクションの後，新領域に残っているスロット変数を回収するために，新領域と旧領域を入れ換え，インクリメンタルなコピーイングを行なう（図3(b)）．

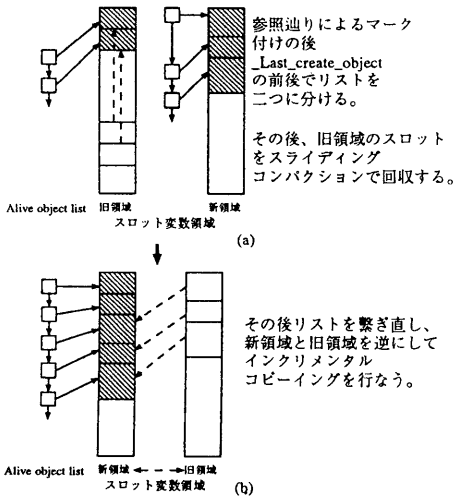

図 3: マーク&スイープ

4 GC の実装

4.1 翻訳系による情報抽出とコード挿入

mosaic の GC を実現するために，実行系には検出不可能な情報を，翻訳系を通じて得ている．

例えば，実行系はスロット変数に含まれているオブジェクト ID を検出する必要がある．しかし，スロット変数中のオブジェクト ID の位置と個数については，mosaic の実行系には分からない．そこで，mosaic の翻訳系はユーザー定義オブジェクトの，スロット変数中のオブジェクト ID の位置を検出し，翻訳時に実行系が分かるためのコードを入れている．

また，メソッド終了時には，メッセージ中に含まれている ID の参照を解放する必要があるが，これも本来は実行系には検出できないものであり，翻訳系がそのためのコードを前もって挿入する方式になっている．

4.2 輸入管理オブジェクト

4.2.1 輸入管理オブジェクトの機能

輸入管理オブジェクトはノード間参照管理を行なう，輸入表に代わるオブジェクトである．

その機能は，ノード外の参照及びそれに伴う重みの管理，輸入管理オブジェクト宛に送られたメッセージを，ノード外のオブジェクトへ再送信することである．

ノード外オブジェクトの ID 及び参照の重みは，必ず輸入管理オブジェクトが持ち，ユーザー定義オブジェクトが持つことはない．輸入管理オブジェクトの生成／消滅，動作は全てシステム側が管理し，ユーザーが意識することはない．

輸入管理オブジェクト宛に送られたメッセージは，輸入管理オブジェクトが各々に固有のフォワード先外部オブジェクトへ再送信する．その際，メッセージ中に含まれている輸入管理オブジェクトの ID とそれに伴う参照の重みを，そのフォワード先のものに載せ換える処置も行なう．その他の ID に対しては，RC から WRC に変更するため，参照の重みを増やす操作をする．(図 4)．

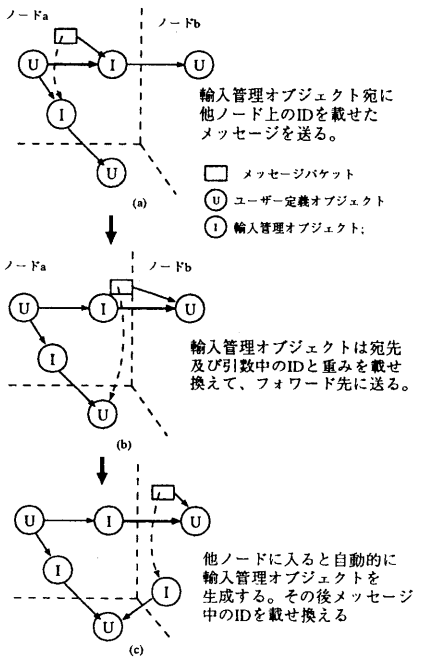

図 4: 輸入管理オブジェクト

4.2.2 輸入管理オブジェクトのストリームオブジェクトによる実現

輸入管理オブジェクトは，ストリームオブジェクトを用いて実現されている．

輸入管理オブジェクト本体とストリームオブジェクト本体は同じものを使っている．輸入管理オブジェクトの動作は，スケジューラが輸入管理オブジェクトのメソッドを呼び出すのではなく，直接システムレベルで管理している．メッセージ通信，及びストリームオブジェクトの連結／短縮アルゴリズム，メッセージ引数へ ID を載せるためのライブラリの変更で，ストリームオブジェクトを輸入管理オブジェクトとして機能させている．

また，生成された時点では，ストリームオブジェクトと輸入管理オブジェクトには，必ずしも区別はない．例えば，ストリームオブジェクトがノード外ユーザー定義オブジェクトと連結された場合は，そのストリームオブジェクトは輸入管理オブジェクトに変化する．

4.3 参照の重みが分割できない時

参照の重みが小さくなり過ぎて分割できない場合，ストリームオブジェクトの連結／短縮機構 [4] を用いて解決している．

1. ストリームオブジェクトを生成し，フォワード先 ID の代わりにストリームオブジェクトの ID をメッセージに載せて輸出する．それと同時に，ストリームオブジェクトと被参照オブジェクトを連結する (図 5(a))．

2. メッセージが宛先ノードに届いた後，そのノード上に輸入管理オブジェクトを生成し，輸入したストリームオブジェクトと連結する (図 5(b))．

3. 連結操作によって，宛先ノード上の輸入管理オブジェクトに，被参照オブジェクトの ID と参照の重みが伝達される．ストリームオブジェクトはバッファリングしていたメッセージをフォワードする (図 5(c))．

この方法により，メッセージのバッファリングを減らすと同時に，並列度が上がることが期待できる．

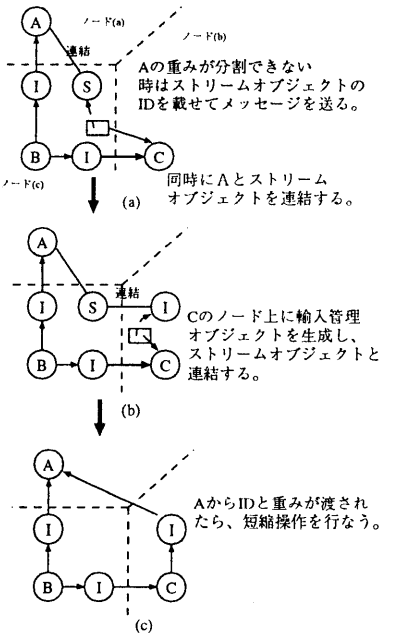

図 5: 重みが分割できない時の処理

5 性能評価

5.1 テスト条件

GC を実装した mosaic の性能を評価するため，Sparc station2 互換機 8 台を VMEbus で接続したマルチワークステーション [2] 上で，基本性能を測定した．

測定項目は，次の通りである．

- GC 停止状態でのノード内及びノード外オブジェクト宛メッセージの生成，スケジューリング，宛先オブジェクトの起動，にかかる時間

- コピーイング実行時に，ノード内メッセージ生成，スケジューリング，宛先オブジェクトの起動，にかかる平均時間．

  オブジェクトは 2 種類あり，スロットのサイズは 12 バイトと 4 バイトである．一回のコピーイングで 10 個のオブジェクトを処理した．参照されているオブジェクト数は各々2000 個ずつで，塵オブジェクトは 0～1000 個程度である．

表 1: 基本機能の測定結果

ノード内メッセージ通信	1[μsec]
ノード間メッセージ通信	8.5[μsec]
コピーイング	3.7[μsec]
ノード内オブジェクト生成	3[μsec]
ノード外オブジェクト生成	23[μsec]

- ノード内／外でのオブジェクト生成にかかる時間

5.2 結果と評価

基本性能の測定結果を表1に示す.

通常，あるオブジェクトが起動されてから，次のオブジェクトが起動されるまでの処理は，メソッドの実行，ノード内メッセージの生成／送信，スケジューリング，次のオブジェクトの起動，から成る．メソッド実行を除く時間が，非GC時では$1\mu sec$であるのに対し，コピーイング実行時では$3.7\mu sec$である．

メソッドの処理にかかる時間が，ある程度以上ならば，この差は十分小さいものと言える．例えば，メソッドが$10\mu sec$だとすると，1オブジェクト当たりの処理時間は，非GC時では$11\mu sec$，コピーイング中では$13.7\mu sec$になる．つまり処理全体にかかる時間の内，コピーイングの処理にかかる時間は約20%にしかならない．

しかしながら，コピーイングに関する数値は，コピーイングに関するパラメータの設定やプログラムの性質などによって，大きく変動することが予測される．例えば，一回のコピーイングで処理するオブジェクトの数を多くしたり，オブジェクトのスロット変数を大きくしたりすると，コピーイングの処理時間は長くなると考えられる．

6 終りに

今回は，mosaicに適したGCを設計，実装した．このGCはmosaicを前提として設計してあり，実装に無理がないため，大きく仕様を変更する必要もなく，効率良く実現できたと考えられる．

今回実装した方式では，RCとWRCを併用することにより，それぞれの欠点を補うことができた．そして，重みの分割ができなくなった時の処理を，ストリームオブジェクトの連結操作を用いて解決した．これにより，分割できない時の処理も統一的に扱える上，メッセージのバッファリングが必要なく，並列度も上がることが期待できる．また，輸出表／輸入表を使わず，オブジェクトに輸出／輸入管理をさせることにより，表自身のGCを統一的に処理できるようになった．

GCを実装したmosaicの，オブジェクトのメソッド以外の処理にかかる時間は，基本性能の測定結果より，非GC時では約$1\mu sec$，インクリメンタルなコピーイング実行時では約$3.7\mu sec$と見積もられる．

しかしながら，基本的な動作にかかる時間程度ならともかく，GCの処理にかかる時間は，各パラメータの設定だけでなく，対象となるプログラムの性質により大きく変化することが予想されるため，その評価が難しい．テストプログラムの選定を含む，GCの評価方法に関する考察が必要になると考えられる．

今後の予定としては，大規模並列計算機への移植を計画中である．

謝辞

本研究に関し，数々の助力を下さった本学の金田悠紀夫教授に対し，心より感謝いたします．

参考文献

[1] 瀧和男. 第五世代コンピュータの並列処理. bit 別冊. 共立出版, 1993.

[2] 瀧和男, 小倉毅, 小西健三. ワークステーション複合体による並列処理システム— 中・小粒度オブジェクト指向並列処理の実現 —. 情報処理学会 PRG13-7 研究報告, Vol. 93, No. 73, August 1993.

[3] 小倉毅, 瀧和男. 並列オブジェクト指向言語とマルチワークステーション上 の実装. JSPP'94 論文集, pp. 97–104, May 1994.

[4] 屋鋪正史, 石原義勝, 松川力, 小西健三, 瀧和男. 並列オブジェクト指向言語 mosaic のランタイム・システム. 情報処理学会 PRG18-7 研究報告, July 1994.

[5] 日比野靖. ごみ集めの基本アルゴリズム. 情報処理, Vol. 35, No. 11, pp. 992–999, Nov 1994.

[6] 市吉伸行. 分散ごみ集め. 情報処理, Vol. 35, No. 11, pp. 1027–1032, Nov 1994.

[7] Douglas M. Washabaugh Dennis Kafura, Manibrata Mukherji. Concurrent and distributed garbage collection of active objects. *IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS*, Vol. 6, No. 4, pp. 337–350, Apr 1995.