並列ガーベジコレクションの効率改善の手法

松井祥悟[†1] 田中良夫[†2]
前田敦司[†3] 中西正和[†4]

スナップショット型（snapshot-at-beginning）と分類されるタイプの並列GCは，無停止処理に向くが，ゴミの回収効率が悪い．筆者らは1982年以来このタイプの並列GCの効率の改善に取り組んできた．この成果が部分印づけ法と相補型GCという二つの方法である．本論文ではその研究の集大成として，スナップショット型の並列GCの問題点を解説し，この二つの方法について概説する．また，GC効率の測定法やCommon Lispを基にしたシステムへの応用例も報告する．

Improvement in Performance of Parallel Garbage Collector

SHOGO MATSUI,[†1] YOSHIO TANAKA,[†2] ATSUSI MAEDA[†3]
and MASAKAZU NAKANISHI[†4]

A class of parallel garbage collectors called *snapshot-at-beginning collectors* have preferable properties such as predictable pause time and ease of implementation. However, these collectors had been known to be inefficient in terms of collection performance, when compared to other types of collectors. We have been working on the improvement of performance of these collecters since 1982. In this paper we summarize the problems on snapshot collectors. Our major achievements, *Partial Marking GC* and *Complementary GC*, are described. Issues about measurement methodology of GC efficiency are discussed. We also report application of our algorithms on a Common Lisp based system.

1. はじめに

動的なデータ構造を取り扱うシステムでは，ヒープ領域に配置したメモリ資源をプログラムが必要に応じて領域確保し使用する，という形態のデータ処理を行うのが一般的である．このようなシステムでは，使用済みのメモリを自動的に回収するガーベジコレクション（以後GC）の機構が不可欠である．

通常，GCは，ヒープ領域のメモリが枯渇した時点でメモリ資源を消費するプログラム（mutator）を停止し，回収プログラム（collector）を起動し，使用済みメモリの回収後にプログラムを再開させるという停止-回収 (stop-and-collect) 方式で行う．

†1 神奈川大学理学部
Faculty of Science, Kanagawa University
†2 電子技術総合研究所
Electrotechnical Laboratory
†3 筑波大学情報処理センター
Science Information Processing Center, University of Tsukuba
†4 慶應義塾大学理工学部
Faculty of Science and Technology, Keio University

この停止-回収方式のアルゴリズムには，たとえばマーク法（mark-sweep GC）やコピー法 (copy GC) のように方法の異なったものがあるが，トレース処理（使用中のメモリを判別するためのメモリ空間中のリンク追跡処理）を行う点が共通している．この場合，停止時間（＝回収プログラムの実行時間）は，使用するヒープ空間の大きさやトレースすべきメモリの量に比例して長くなる．たとえば大きなセル空間を持つLisp処理系ではこの停止時間が無視できない長さになる．このため停止時間の短い世代管理型GCや並列GCが開発されている．

並列GCは，mutatorの動作中にもcollectorを並列に動作させる．完全な並列動作が実現できれば，mutatorの動作からcollectorによる中断を完全に取り除くことができる．無停止となることから実時間処理への応用を可能とし，全体の処理時間がGC時間分だけ短くなることから高速化にもつながる．

しかし，初期の並列GCシステムには解決すべき問題があった．アルゴリズム上の問題から完全な並列動作ができないためにmutatorの処理に中断が発生したり，並列化によるゴミの回収能力の低下からメモリ

枯渇による中断を招いた．

著者らは，1982 年に開発が開始されたマルチマイクロプロセッサ LISP マシンに，はじめて並列 GC (Synapse GC[1]) を実装して以来，汎用マルチプロセッサ計算機上の無停止処理に利用できる実用的な並列 GC アルゴリズムの確立を目的とし，これらの問題に取り組んできた．その成果が，世代管理型 GC の概念を取り入れた部分印づけ法と基本的アルゴリズムを相補的に組み合わせた相補型 GC である．本論文では，最初に，並列 GC の基本的な動作を解析し，問題点を明らかにする．つづいて，我々が提案した方法（基本的な並列 GC と二つの改善法）について述べる．並列 GC の効率の評価方法についても解説する．最後に，実際の Lisp システムへの応用例を紹介する．

2. 並列 GC のアルゴリズム

本節では，並列 GC の基礎事項と基本的なアルゴリズムについて述べ，それぞれの問題点を明らかにする．

2.1 並列 GC の動作

並列 GC (Parallel GC) は，マーク法やコピー法のような停止型の GC を並列化したものである．停止型 GC は，自由セルの枯渇のたびに起動され，その間リスト処理が完全に停止するのに対して，並列 GC はリスト処理プロセス (mutator) と GC プロセス (collector) が並列に動作する．並列 GC は，collector に独立した専用プロセッサを割り当てた完全な並列処理により GC を行う．1 台のプロセッサの疑似並列処理により GC を行う漸次 GC (Incremetal GC) も並列 GC の一種である．漸次的 GC では collector の動作を小さな部分に分け，mutator の一般の処理の中に埋め込む．たとえば，決められた個数のセルに対する印づけや回収作業を，cons 関数が呼ばれるたびに実行する．

マーク法を基にした並列 GC では，“ルート挿入 (root-insertion)”，“印づけ (marking)”，“回収 (sweep)” の三つのフェーズで一つの GC サイクルを形成する．ルート挿入フェーズでは，collector は mutator が保持する生きている（使用中の）セルを指すポインタをすべて収集する．これをルートポインタと呼ぶ．また，これを保持するものをルートと呼ぶ．実際には，mutator のレジスタやグローバル変数，スタック上の領域などがルートとなる．このようなルートの集まりをルートセットと呼ぶ．印づけフェーズでは，収集したルートポインタから到達可能なすべてのセルにマーク済みの印を付ける．回収フェーズでは，全セルを走査し，印のないセルを自由リストにつなげ，マーク済みの印を消す．

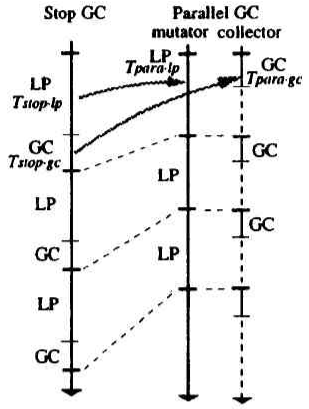

図 1 GC の並列化の原理
Fig. 1 Stop GC and Parallel GC

2.2 並列 GC の効率

並列 GC の概念図を図 1 に示す．時間は上から下に流れる．左は停止型 GC を表し，右の二つは並列 GC の mutator と collector を表す．LP はリスト処理，GC は GC 処理を示す．また，$T_{stop.lp}$ を停止型の LP 時間，$T_{stop.gc}$ を停止型の GC 時間，$T_{para.lp}$ を並列型の LP 時間，$T_{para.gc}$ を並列型の GC 時間とする．

図のように，停止型 GC を並列化すると全処理時間は GC 時間分だけ短縮される．GC が LP の時間内に収まる限り ($T_{para.lp} \geq T_{para.gc}$)，LP の中断は発生しない．

並列化のオーバヘッドにより GC 時間が r 倍になるとすると，$T_{para.gc} = r \cdot T_{stop.gc}$ とおける．この r は，実行時間の実測により推定可能である．$1/r$ を並列 GC の動作効率と決め，以下の議論で使用する．動作効率については第 4 節で詳細に検討する．

2.3 バ リ ア

停止型 GC を単純に並列化した GC を考えると，collector の印づけやコピー動作中に mutator がセルのポインタを書き換えた時，使用中のセルに印がつかず，誤って回収される場合がある．図 2 の例で考えると，collector がルート r1 につながる a～d のセルに印づけした時点で，*1 と *2 のポインタがこの順で破線で示すように書き換えられた場合，残りのルート (r2, r3) に対する印づけを行っても f, g, h のセルには印はつかない．これらのセルは書き換え後もルートから到達可能であるにもかかわらず回収されることになる．

並列 GC には，これを補償する処理が含まれている．

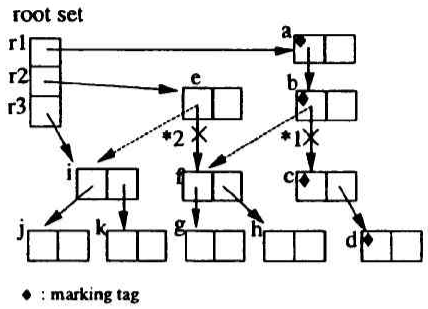

図 2 並列化の問題点
Fig. 2 Pointer rewriting problem

この原理を図 3 に示す．印づけ処理中には，mutator は，共有メモリ（shared memory）からの読み出し（read），書き込み（write）および新しいセルの生成（create）を行う．collector は，読み出し（read）を行う．補償処理のためには，mutator の動作にバリアを設け，各動作が発生した場合に，その情報が collector へ通知されるようにする．collector は，その情報に基づき，追加の印づけを行えばよい．

一般に，実行頻度は，read > create >> write であるから，読み出しバリアや生成バリアのオーバヘッドが大きいと，mutator の実行速度に大きく影響する．現在の並列 GC では，書き込みバリアだけで実現できるように，アルゴリズムを工夫している．

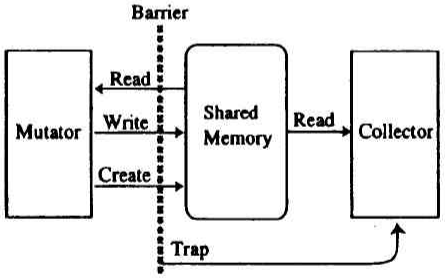

図 3 バリア
Fig. 3 Barrier

2.4 基本的アルゴリズム

一般的な並列 GC は，*incremental update* 型（IU 型）と *snapshot-at-beginning* 型（SB 型）の 2 種類に分類できる [2]．

IU 型アルゴリズム

IU 型アルゴリズムは，停止型 GC を単純に並列化したものである（図 4）．collector は，mutator のルートセットから到達できるセルに印づけを行い，印のないセルを回収し自由リストに戻す．停止型との大きな違いは，印づけフェーズ中にセルの書き換えを行った場合の補償処理と，印づけフェーズ終了時に必要なルートセット走査である．

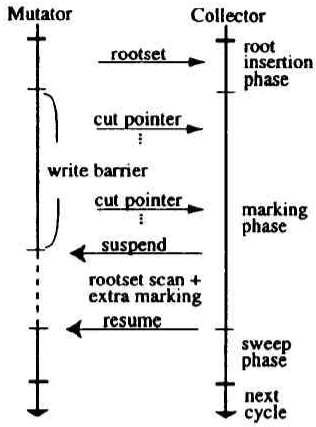

図 4 IU 型 GC アルゴリズム
Fig. 4 IU type of algorithem

補償処理には，書き込みバリアを設け，新たに書き込むポインタの値を collector に通知する．collector は通知されたポインタの指定先のセルに対して追加の印づけを行う．図 2 の例では，*1 の書き換え時にはセル f へのポインタを，*2 ではセル i へのポインタを collector へ通知する．問題となる f，g，h のセルは追加印づけされる．

生成バリアは不要である．印づけフェーズ中に生成されたセルは，もしゴミでないならば，そのセルへのポインタはどこかの生きたセルへ書き込まれるはずであり，書き込みバリアにより印づけされ，生き残る．ゴミならば書き込みは発生せず，印は付かない．

ルートセット走査は，ルートセットを走査し，印が付いていないセルへのポインタを保持していないかどうかをチェックする．もし，そのようなポインタが存在すれば，それらに対して追加印づけを行う．このルートセット走査と追加印づけは，mutator の動作を止めて実行しなければならない [2]．この点が無停止処理への応用の大きな障害となっている．（付録 A.1 参照）．

SB 型アルゴリズム

SB 型アルゴリズムは，GC サイクル開始時のセルの状態をスナップショット写真を撮るように記録しておき，その記録の上のゴミセルだけを回収しようという考え方である．スナップショット後に発生するゴミは回収できないが，予測不可能な mutator の停止がない．

基本的には，スナップショット時に，ルートセット

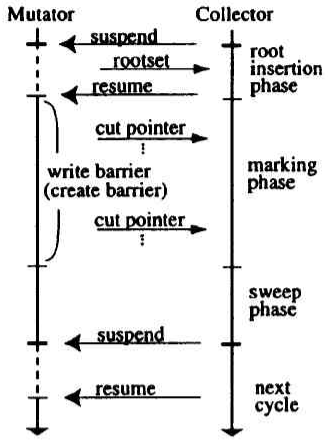

図 5 SB 型 GC アルゴリズム
Fig. 5 SB type of algorithem

を含むすべてのセルの状態を記録し，それに基づいて印づけを行えばよい．実際には，ルートセットだけを記録しておき，mutator がセルの書き換えを行う場合に，切断したポインタを collector に報告し，追加の印づけを行う．また，mutator が生成したセルは印づけされないので，回収されないように，mutator があらかじめ印をつけておく．

SB 型アルゴリズムを図 5 に示す．mutator を止めた状態でルートセットのコピーを行う．以後は，mutator を停止する必要はない．コピーしたルートに基づき印づけを行う．印づけフェーズ中の mutator のバリアは，1) 書き込みバリア: ポインタを書き換えた場合，上書きされ消去されるデータ（図 2 の例では，*1 の切断では c へのポインタ，*2 の切断では f へのポインタ）を cellector へ通知する；2) 生成バリア: 生成されたセルにあらかじめ印をつけておく☆; となる．すべてのルートから到達できるセルと mutator から通知されたセルに対して印づけが終わると，回収フェーズに移行する．ゴミセルを自由リストへ戻し，印を消す．

2.5 IU 型と SB 型並列 GC の比較

動作効率

SB 型では，回収フェーズで回収されるゴミは，ルート挿入時にゴミであったセルだけである．印づけ中にゴミとなったセルは同じ GC サイクルでは決して回収されず，次の GC サイクルで回収される．したがって，動作効率は小さい．このようなセルは，GC の 1 サイクルの処理時間が長いほど多く発生する．

☆ 自由リストのセルにあらかじめ特殊な印をつけておくことで，生成バリアは取り除くこともできる [1]．

世代管理型 GC の裏付けとなっているセルの生存時間に関する研究では，寿命が極端に短い大量のセルと非常に長い一部のセルに二極化することが指摘されている [3]．SB 型では，数多い短寿命のセルを 1 サイクル遅れで回収することになり，生存セルが多く印づけに時間がかかる場合に，特に動作効率が悪くなる．第 4 節で示すように，実測データにより，$1/r \approx 1/2$ が確認されている．

mutator の停止時間

IU 型と SB 型のアルゴリズムには，どちらも mutator の停止中に行う操作がある．SB 型の操作はルートセットのコピーであり，停止時間はルートの量（主としてスタックの長さ）に比例する．一方，IU 型の操作は，ルートセット走査と追加印づけの時間である．ルートセット走査の時間は SB 型と同様にルートの量に比例するが，追加印づけに関しては，予測不可能である．ほとんどの場合は，追加印づけが全く必要ないか，あるいは，あっても短時間ですむと考えられるが，セル空間のほとんどすべてのセルに印をつけるような場合も考えられる．

書き込みバリア

二つのアルゴリズムにはともに書き込みバリアが必要である．しかし，両者の操作は異なる．IU 型の場合は上書きする新しいポインタの情報，SB 型の場合は上書きされる古いポインタの情報が必要である．図 2 の例では，*1 の書き換えで，IU 型は f へのポインタを報告するが，SB 型では c へのポインタを報告する．*2 の書き換えでは，IU 型は i へのポインタを，SB 型では f へのポインタを報告する．

ある特定のセル上のポインタを何回も書き換えるような場合を考えると，書き換えのたびに毎回 collector へ報告する必要はない．IU 型では，最後に書き込まれたポインタ（最も上のデータ）だけが必要である．一方 SB 型は最初にあったポインタ（最も底のデータ）だけが必要である．

IU 型では，新しいポインタは書き換えられたセル上に残されているため，書き換えたセルの位置を記録しておくだけでよい．図 2 の例では，b や e の位置を記録する．collector は，記録されたセルを調べ，印のないセルへのポインタが書かれていれば，印づけ処理を行えばよい．SB 型は，必要なポインタ情報が上書きで消えてしまうので，このようなことはできない．

総合評価

IU 型と SB 型の GC の比較を表 1 にまとめる．IU 型は，動作効率が高く，書き込みバリアも簡単になることから，効率のよい並列 GC が実現できる．しか

表 1 IU 型と SB 型 GC の比較
Table 1 IU and SB

	IU 型	SB 型
動作効率 (1/r)	大 ≈ 1	小 ≈ 1/2
mutator の停止時間	予測不可能 追加印づけ	予測可能 root の量
書き込みバリア	小 ログ	大 データ保存

し，予測不可能な長さの mutator の停止が発生するため，無停止処理への応用は困難である．逆に SB 型は，mutator の停止時間が予測可能であるため，無停止処理への応用が可能である．しかし，動作効率が低いため，セルの枯渇による mutator の停止が発生しやすい．また，書き込みバリアは IU 型より複雑である．

3. 並列 GC の効率化

汎用マルチプロセッサ計算機をターゲットとして，無停止用途へも応用可能な並列 GC の実現を目的とすると，SB 型アルゴリズムにおいては，動作効率の改善，ルート挿入時の mutator の停止時間の短縮，書き込みバリアの改善が課題となる．また，IU 型アルゴリズムにおいては，ルートセット走査および追加印づけ時の mutator の停止の問題の解決が課題となる．

本節では，我々がこれまでに提案した 3 つの並列 GC について説明する．最初に，基本的な SB 型並列 GC である Synapse GC について説明し，つづいて，SB 型 GC の動作効率を改善する部分印づけ法と IU 型のルートセット走査および追加印づけの問題を改善した相補型 GC について説明する．

3.1 Synapse GC

Synapse GC[1] は，実際のマシンに実装され動作が確認できた初期の並列 GC のうちの一つである．図 5 に示した基本的な SB 型並列 GC であるが，SB 型の欠点である動作効率の悪さを補うため，複数台の collector が同時に動作できるように，アルゴリズムが工夫されている．

我々が開発をすすめたマルチマイクロプロセッサ Lisp マシンへの実装では，collector 用に複数の MPU を割り当て，実際に動作させた．また，印づけ用の特殊なメモリ装置やバリア処理の通信用のハードウェアスタックなど，ハードウエアからのサポートも行った．

Synapse GC の collector および mutator のアルゴリズムを，図 6，図 7 に示す．このアルゴリズムのデータ構造は付録 A.2 に示した．また，アルゴリズムで共通に使用する手続きである GP_mark と GP_append も同付録の図 21 に示した．

```
Procedure Synapse_GC_Cycle ;
  var i : t_pointer ;
  begin
    PHASE := rootins ;
    suspend_mutator ;
    for i := 1 to R do
      push( root[i] ) ;
    PHASE := marking ;
    resume_mutator ;
    for i := 1 to R do
      GP_mark(root[i]) ;
    while << the stack is not empty >> do
        begin
          i := pop ;
          GP_mark(i)
        end ;
    PHASE := sweep ;
    for i := 1 to M do
      if cell[i].color = white then
        GP_append(i)
      else if cell[i].car <> f then
             cell[i].color := white
  end
```

図 6 Synapse GC のアルゴリズム (collector)
Fig. 6 Synapse GC Algorithm (collector)

```
{replacing car pointer of cell[root[m]]
     to root[n] m, n : t_root }
procedure LPa  ;
  begin
    if PHASE = marking then
      push(cell[root[m]].car) ;
    cell[root[m]].car := root[n]
  end ;

{cons( root[m], root[n]) m, n : t_root}
procedure LPc  ;
  begin
    while FREE_LEFT = FREE_RIGHT do {waiting} ;
    root[R] := FREE_LEFT ;
    FREE_LEFT := cell[FREE_LEFT].cdr ;
    cell[root[R]].car := root[m] ;
    cell[root[R]].cdr := root[n] ;
    root[m] := root[R]
  end
```

図 7 Synapse GC のアルゴリズム (mutator)
Fig. 7 Synapse GC Algorithm (mutator)

印づけは，white, black, offwhite を用い，それぞれ，印づけなし，印づけあり，自由セルの状態を表す．この offwhite と自由セルの car 部分に埋め込まれる特殊なポインタ f によって，セルの生成の際のオーバヘッドが小さくなっている．アルゴリズム（図 7）でも，セルの書き換え（LPa）には書き込みバリアがあるが，セルの生成（LPc）では，バリアや特殊な印づけ操作などのオーバヘッドが一切ないことが確認できる☆．

☆ collector を停止させる場合や漸次型の GC として実装する場合には，動作効率をあげるために印除去の操作が必要になる．

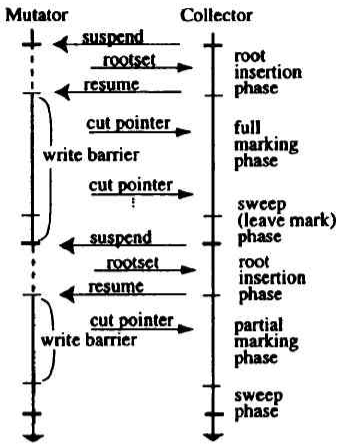

図 8 部分印づけ法
Fig. 8 Partial Marking GC

3.2 部分印づけ法

部分印づけ法 (Partial Marking GC)[4)~6)] は世代管理型 GC の考え方を取り入れた並列 GC である.

部分印づけ法は, 通常の GC サイクル (full marking サイクル) の直後に, 部分的なセルに対してだけ印づけを行う GC サイクル (partial marking サイクル) を一回以上挿入した並列 GC である. partial marking サイクルの挿入回数により, 世代管理型 GC における advancement threshold を決定することができる. 本節では説明を容易にするために, partial marking サイクルを一回だけ挿入したものについて説明する (図 8).

partial marking サイクルでは, 直前の full marking サイクルの間に生成されたセルに対してだけ印づけを行う (この印づけを partial marking という). したがって, 印づけが短時間で終了し, 一回のサイクルの所要時間 (サイクル時間) も短くなる. 結果的に, 直前の full marking サイクルの間に生じたゴミを短い時間で回収することが可能となる. また, サイクル時間が短いのでこの間に生じるゴミが少なくなり, 直後の GC サイクルでの動作効率を上昇させることになる.

部分印づけ法の collector, mutator のアルゴリズムを, 図 9, 図 10 に示す. PAIR_OF_GC_CYCLE が主手続きである. full および partial の GC サイクルを一度ずつ行う.

部分印づけ法は, "full marking サイクルの回収フェーズにおいて, 先の印づけフェーズでセルに付けられたマーク済みの印 (black) を消去しない" という単純な操作で実現できる. この残されたセル上の

```
Procedure PAIR_OF_GC_Cycle ;
  begin
    CYCLE := full ; GC_Cycle ;
    CYCLE := partial ; GC_Cycle
  end;
Procedure GC_Cycle ;
  var i : t_pointer ;
  begin
    PHASE := rootins ;
    suspend_mutator ;
    for i := 1 to R do
      push( root[i] ) ;
    PHASE := marking ;
    resume_mutator ;
    for i := 1 to R do
      GP_mark(root[i]) ;
    while << the stack is not empty >> do
        begin
          i := pop ;
          GP_mark(i)
        end ;
    PHASE := sweep ;
    if CYCLE = full then
      for i := 1 to M do
        if cell[i].color = white then
          GP_append(i)
        else if (cell[i].coler = offwhite) and
                (cell[i].car <> f) then
              cell[i].color := white ;
    else
      for i := 1 to M do
        if cell[i].color = white then
          GP_append(i)
        else if cell[i].car <> f then
              cell[i].color := white
  end
```

図 9 部分印づけ法のアルゴリズム (collector)
Fig. 9 Partial marking GC Algorithm (collector)

```
procedure LPa  ;
  begin
    if PHASE = marking then
      push(cell[root[m]].car) ;
    if CYCLE = full then
      if (PHASE = marking) and
         (cell[root[n]].color = offwhite) then
           push(root[n])
      else if (PHASE = sweep) and
         (cell[root[n]].color <> black) then
           push(root[n]) ;
    cell[root[m]].car := root[n]
  end ;

procedure LPc  ;
  begin
    while FREE_LEFT = FREE_RIGHT do {waiting} ;
    root[R] := FREE_LEFT ;
    FREE_LEFT := cell[FREE_LEFT].cdr ;
    cell[root[R]].car := root[m] ;
    cell[root[R]].cdr := root[n] ;
    root[m] := root[R]
  end
```

図 10 部分印づけ法のアルゴリズム (mutator)
Fig. 10 Partial marking GC Algorithm (mutator)

印により, その直後に行われる GC サイクルの印づけは, 他のサイクルと全く同じ通常の印づけの動作を行うにもかかわらず, 自動的に partial marking となる.

書き込みバリアは複雑になる. 基本的に SB 型であるので, mutator がポインタを書き換える場合には, 上書きされる古いポインタを collector に通知する. full

marking のフェーズ中は，それに加えて，書き換える新しいポインタも通知する必要がある☆．つまり，SB 型のバリアと IU 型のバリアが同時に必要になる．これは，印のついたセルから印のないセルへのポインタの発生を防ぐためである．

ポインタの書き換えによって印のついた（black）セルから印づけのされていないセル（white や offwhite）へのリンクが生成されると，印を残す回収フェーズ（full marking サイクルの回収フェーズ）を実行した後にも，印のついたセルから印のないセルへのリンクとして残される．このような状態が発生すると，次のサイクルの印づけフェーズでも，印の残された直前のセルのために印が伝播せず，そのセルは印のないまま誤って回収されてしまう．

Synapse GC のアルゴリズムと比較すると，追加された部分が，場合分けされた回収フェーズと LPa のバリアの部分だけであることがわかる．実際，常に CYCLE = partial として GC_CYCLE を実行すると Synapse GC と全く同じ動作になる．

3.3 相補型 GC

相補型 GC(Complementary GC)[7),8)] は IU 型アルゴリズムと SB 型アルゴリズムを相補的に組み合わせた方法である．

IU 型アルゴリズムの大きな問題点はルートセット走査と追加印づけを mutator を止めた状態で行わなければならない点である．(付録 A.1)．この追加印づけを mutator の動作中にも実行できれば SB 型と同様に mutator の停止時間の予測が可能になる．

相補型 GC は，IU 型アルゴリズムのルートセット走査と追加印づけを SB 型アルゴリズムで行う（図 11）．IU 型でルート挿入，印づけを行った後に，SB 型でルート挿入，印づけ，回収を行う．この五つのフェーズが一つの GC サイクルを構成する．IU 型の部分は通常の IU 型アルゴリズムから回収フェーズを取り除いたものである．この間に mutator がポインタの書き換えを行った場合は，IU 型のマナーで collector へ通知を行う（上書きする新しいポインタを通知する）．後半に行う SB 型の部分は，単独で動作する場合の SB 型アルゴリズムと同じである．この間のポインタの書き換えは，SB 型のマナーで collector へ通知を行う（上書きされる古いポインタを通知する）．また，生成処理では，セルにマーク済みの印をつけておく．後半に行う SB 型部分のルート挿入および印づけが前半の IU 型部分の “ルートセット走査と追加印づけ” として機能する．

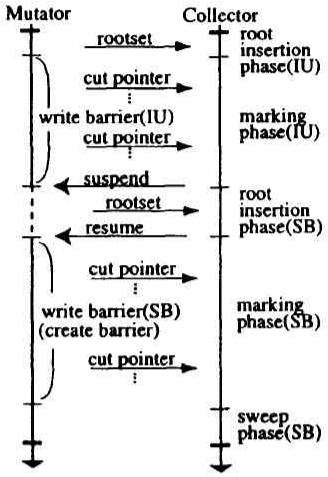

図 11 相補型 GC
Fig. 11 Complementary GC

前半の IU 型の部分の印づけフェーズ後にルートセットに残される可能性がある印づけされていないセルへのポインタは，直後に行う SB 型アルゴリズムのルート挿入で必ず検査され，印づけが行われる．したがって，ルート書き換え問題は補償される．また，SB 型の部分では基本的にルート書き換え問題は生じない．このため，後半の SB 型の動作は mutator の動作中に行っても問題は生じない．つまり，IU 型では不可能であった mutator 動作中の追加印づけができることになる．

相補型 GC の collector，mutator のアルゴリズムを図 12，図 13 に示す．Complementary_GC_CYCLE が主手続きである．

効率について考えると，それぞれのアルゴリズムを単独で動作させる場合と同様に，前半の IU 型の印づけフェーズの間にゴミになったセルは後の回収フェーズで回収される可能性があるが，後半の SB 型の印づけフェーズの間にゴミになったセルは直後の回収フェーズでは決して回収されない．したがって，後半の SB 型の印づけフェーズの時間が短いほど，回収フェーズで回収できるゴミの数が多くなる．総合的な動作効率は，それぞれの型の印づけフェーズの処理時間の比率によって決まることになる．最悪の場合は SB 型アルゴリズムの効率と等しくなり，最良の場合は IU 型アルゴリズムの効率と等しくなる．一方，後半の SB 型の動作で印づけされるセルは，前半の IU 型のルート書き換え問題で生じるセルだけである．これらは，前

☆ 実際のアルゴリズムでは，さらに細かい場合分けを行い，不要な通知を省いている．

```
Procedure Complementary_GC_Cycle ;
var i : t_pointer ;
begin
  {Root-insertion and Marking phase
              of incremental-update}
  REQUEST_PUSH := incremental ;
  for i := 1 to R do
    GP_mark(root[i]) ;
  while << the stack is not empty >> do
      begin
        i := pop ;
        GP_mark(i) ;
      end ;
  {Root-insertion phase
        of snapshot-at-beginning}
  suspend_mutator ;
  REQUEST_PUSH := snapshot ;
  CONS_COLOR := offwhite ;
  for i := 1 to R do
    push( root[i] ) ;
  resume_mutator ;
  {Marking phase of snapshot-at-beginning}
  while << the stack is not empty >> do
      begin
        i := pop ;
        GP_mark(i) ;
      end ;
  {Sweeping phase}
  REQUEST_PUSH := idle ;
  for i := 1 to M do
      if cell[i].color = white then
        GP_append(i) ;
      else if cell[i].car <> f then
        cell[i].color := white ;
  CONS_COLOR := white ;
end
```

図 12 相補型 GC のアルゴリズム (collector)
Fig. 12 Complementary GC Algorithm (collector)

```
procedure LPa  ;
  begin
    if REQUEST_PUSH = incremental then
      push(root[n])
    else if REQUEST_PUSH = snapshot then
      push(cell[root[m]].car) ;
    cell[root[m]].car := root[n]
  end ;

procedure LPc  ;
  begin
    while FREE_LEFT = FREE_RIGHT do {waiting} ;
    root[R] := FREE_LEFT ;
    FREE_LEFT := cell[FREE_LEFT].cdr ;
    cell[root[R]].car := root[m] ;
    cell[root[R]].cdr := root[n] ;
    if   CONS_COLOR <> offwhite then
      cell[root[R]].color := white ;
    root[m] := root[R]
  end
```

図 13 相補型 GC のアルゴリズム (mutator)
Fig. 13 Complementary GC Algorithm (mutator)

半の印づけフェーズの間に，ポインタを切断されたり，新たに生成されることにより印づけフェーズが終了した時点でルートセットだけから到達できるようになったセルである．このようなセルの数は実行するアプリケーションにより変化するが，一般に，生きているセル総数に比べると十分少ない．したがって，総合的な動作効率は単独の IU 型アルゴリズムの動作効率に近くなると推定できる．

LPa のバリアは，REQUEST_PUSH により切り替わるが，Synapse GC と同程度の処理である．しかし，LPc には，セルの印の解除の処理が追加されている．この操作は Synapse GC や部分印づけ法では不要であるため，オーバヘッドとなる．この印解除は，本来，IU 型アルゴリズムには必然の処理である．このために IU 型 GC は SB 型では回収できないセル（その GC サイクル中にゴミになるセル）も回収できる．

4. 並列 GC の評価法

本節では，我々が考案した並列 GC の効率の評価法について説明し，それによる部分印づけ法および相補型 GC の評価を行う．

4.1 並列 GC の効率

我々は，GC 率（GC ratio）と改善率（Improvement ratio）を使用した評価法を考案して用いている．

従来の停止型 GC を持つ lisp 処理系を stop-lisp，並列 GC を持つ lisp 処理系を para-lisp と表す．GC ratio G, Improvement ratio I を次のように定義する．

$$G = \frac{T_{stop.gc}}{T_{stop.total}},$$

$$I = \frac{T_{stop.total} - T_{para.total}}{T_{stop.total}},$$

ただし，

$T_{stop.gc}$	stop-lisp の GC 時間の合計，
$T_{stop.total}$	stop-lisp の全処理時間，
$T_{para.total}$	para-lisp の全処理時間．

mutator のセルの消費率は定常状態であると仮定する．すでに定めたように，$r = T_{para.gc}/T_{stop.gc}$ とおく．並列 GC の G と I は，$T_{para.total} = max(T_{para.lp}, T_{para.gc})$ より，次のようにまとめられる．

$$I = min(G - O, 1 - rG), \qquad (1)$$

ただし，para-lisp の mutator の処理のオーバヘッド時間を $T_{para.oh}$ とし，オーバヘッド率（overhead ratio）を $O = T_{para.oh}/T_{stop.total}$ とする．

同様に，漸次型 GC の場合は，$T_{para.total} = T_{para.lp} + T_{para.gc}$ より，

$$I = (1 - r)G - O \qquad (2)$$

となる．

動作効率

$1/r$ を GC の動作効率と定義する．並列 GC のゴミの回収能力が停止型 GC の $1/r$ 倍であることを意味する．O は mutator の処理のオーバヘッドを示す．理想的な状態，すなわち，並列 GC の回収能力が停止

型 GC の回収能力と全く同じであり mutator のオーバヘッドが全くない場合には，$r = 1, O = 0$ となる．

並列型 GC の動作効率は，動作させるアプリケーションに依存する．式 (1) および式 (2) を用いると，その G と I を実測することにより，それぞれのアプリケーションを実行した場合の GC 動作効率を個別に求めることができる．実際には，G と I をプロットすることで，グラフの移動量や傾きから O と r が求まる．

4.2 SB 型アルゴリズムの動作効率

マルチプロセッサワークステーション LUNA88k, mach OS 上の並列 GC Lisp 処理系で実験した スナップショット型 GC の G と I のグラフを図 14 に示す．並列型は ParaGC，漸次型は IncGC と記した．自由セルはそれぞれ 250,000 個ある．並列型は従来型の SB 型 GC (Synapse GC) である．mutator, collector にそれぞれ 1 台ずつプロセッサを割り当てた並列動作を行う．collector はセルの枯渇がないかぎり，常に並列に動作する．並列動作は C threads パッケージで実現した．漸次型は，湯浅の方法 [9] と同様である．全セルの 10%を切った時点で collector が GC 処理を開始し，1 セルを cons する間に，20 個のセルに対し印づけまたは回収を行う．並列 GC の理想曲線 ($r = 1, O = 0$) を破線で付記した．漸次型の理想曲線は $I = 0$ の直線となる．実験に使用した関数は，結果を残さない cons を一定回数繰り返す関数 eatcell☆ である．また，G を変えるために，数種の長さの固定リストを作り，同じ関数を実行した．

並列型のグラフにおいて，単調増加部分 (G が 0 から極大部分まで) は，Hickey らの論文 [10] の stable の状態 (セルの枯渇が発生しない状態)，単調減少部分のうち I が正の部分は alternating の状態 (2 回に 1 度の GC サイクルで，セルの枯渇が発生する状態)，単調減少部分のうちの負の部分は critical の状態 (毎回の GC サイクルでセルの枯渇が発生する状態) である．セルの枯渇が発生すると，mutator の処理は，collector が新たにセルを回収するまでの間，中断される．したがって，グラフの極大かつ最大を示す部分の G の値は，実時間処理の可能な G の上限値 (**実時間処理限界点**) である．

並列型の場合，式 (1) より，ゴミ回収能力の低下により r が大きくなると，グラフのピークは原点方向に移動し，実時間処理限界点も小さくなる．また，グラフの改善率が負の部分は，停止型に比べて処理時間が長くなっている部分である．単調減少部分で改善率が負になる G の最小値 (効率改善限界点) は，r が大きくなるほど小さな値となる．漸次型の場合，式 (2) より，ゴミの回収能力の低下により r が大きくなると，グラフの傾きが大きくなり，全般の改善率が下がる．

図 14 の場合の r は並列型で約 2，漸次型で約 3 である．すなわち，ゴミの回収能力はそれぞれ停止型の約 1/2 および 1/3 に低下していることを示す．並列型の場合，実時間処理限界点は約 0.35 であり，理想的な曲線の 0.5 と比べると，低くなっていることがわかる．また，効率改善限界点は 0.5 となり，停止型の場合より処理時間が長くなる部分が現れる．漸次型の場合，グラフの $G = 0.35$ 付近に不連続部分があり，G がこれより大きい部分では r はさらに大きい．実験でもこの部分ではセル枯渇が生じた．理論上，実時間処理限界点はグラフ上に現れないが，実際には，この不連続部分がこれを示すものと考えられる．

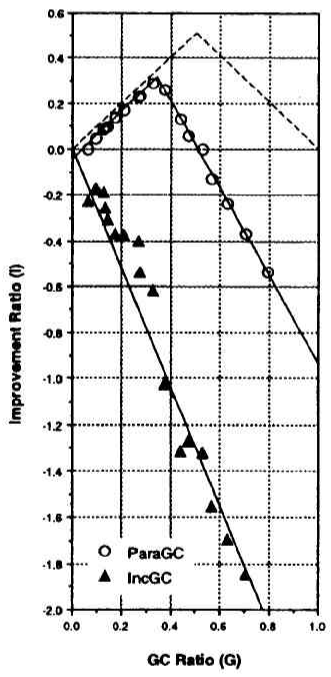

図 14 GC 率と改善率
Fig. 14 GC Ratio and Improvement Ratio

4.3 部分印づけ法と相補型 GC の動作効率

図 14 と同じ並列 GC Lisp 処理系に同じ条件で実装し実測した G と I のグラフを図 15 に示す．従来型 (SB 型) アルゴリズムの並列 GC を Basic ParaGC，漸次 GC を Basic IncGC と示した．同様に部分印

☆
```
(defun eatcell (n)
        (cond((zerop n) nil)
             (t (cons nil nil)(eatcell (1- n)))))
```

づけ法の並列 GC を Partial ParaGC，漸次 GC を Partial IncGC と示し，また，相補型 GC の並列 GC を Comple.ParaGC，漸次 GC を Comple.IncGC と示した．

並列型の場合，部分印づけ法と相補型 GC はほぼ同じ動作効率を示した．この時の r の値は約 1 となり，ゴミの回収能力は停止型と同程度であることがわかる．これは，従来の SB 型 GC の 2 倍の動作効率を持つことを示す．この結果，実時間処理限界点は 0.5 付近まで上昇し，実時間性が向上していることがわかる．これは，全処理時間の半分が GC に費やされるようなリスト処理でも実時間処理が可能であることを示す．また，効率改善限界点は 0.8 付近まで上昇している．理想の曲線（破線）に近づいていることがわかる．また，オーバヘッド率 O は，3 つの方法ともに約 0.1 であり，大きな差異は認められない．

また，漸次型でも，G が 0.4 を越える部分から大きく改善されている．また，グラフには現れないが，実際の計測では実時間処理限界点が並列型と同様，0.35 付近から 0.5 付近まで上昇したことが確認できた．

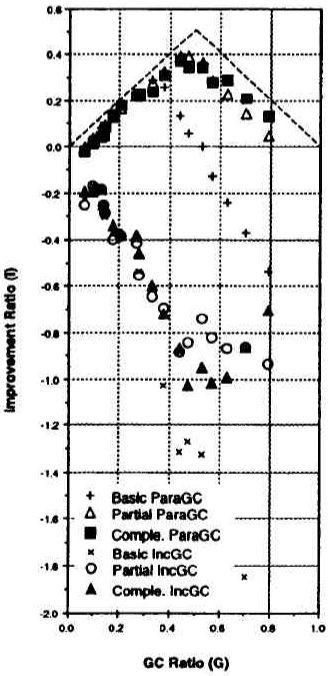

図 15 部分印づけ法と相補型 GC の改善率
Fig. 15 Improvement Ratio of Partial Marking GC and Complementary GC

4.4 部分印づけ法の効率の解析

部分印づけ法は，世代管理型 GC と同様に，短寿命のセルの生存の割合が多いことが前提となっている．SB 型 GC の回収フェーズにおいて，ゴミであるのに回収できないセルには，そのサイクルで生成された短寿命のセルと，それ以前から存在する長寿命のセルが含まれる．この回収できないセル全体に占める長寿命のセルの割合を b とする．b が小さい場合，部分印づけ法を用いると，partial サイクルで回収できるゴミが多くなり，動作効率も大きくなる．逆に b が大きい場合には動作効率は小さくなり，SB 型を下まわる場合も発生する．

これを確認するために，Hickey らの方法 [10] にならった解析を行った [6]．セル空間の大きさが $5M$，生きているセルの数が $500K$ 個であり，1 つのセルの印づけに 0.5μ 秒かかり，走査に 0.1μ 秒かかる場合で試算すると，1 つのサイクルで mutator が待たされることなく生成できるセルの最大数 G^{crit} は，SB 型の場合 $2.75M$ 個と計算できる．同じ条件で求めると，部分印づけ法の G^{crit} は b の関数となる（図 16）．図のように，$b \leq 0.45$ では，SB 型の値 $2.75M$ を上まわる．一般的なアプリケーションがとる b の値である 0.02 から 0.2 では，十分大きな値をとることが確認できる．

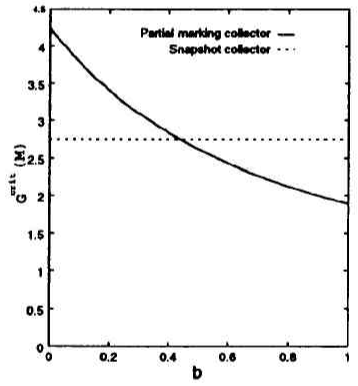

図 16 部分印づけ法の G^{crit}
Fig. 16 Maximum value of G^{crit} to preserve the stable state

5. 部分印づけ法の EusLisp への実装

EusLisp は，電子技術総合研究所で作られた *Common Lisp* を基にしたオブジェクト指向プログラミング言語である [11]．3D 幾何モデラの実装，およびそのハイレベルロボットプログラミングへの応用を目的としている．応用範囲を実時間処理分野へ広げるために，EusLisp は Solaris 2 OS のマルチスレッド機能を用いた並列プログラミングをサポートする．

初期バージョンの EusLisp は停止型マーク法の GC しか持たず，このために実時間処理への応用が限定された．通常，一回の GC 動作は数百ミリ秒かかり，その間は通常処理に中断が生じる．この中断は，動画処理やアクチュエータのサーボ制御のようなアプリケーションでは重大な問題となる．EusLisp の実時間性を改善するために，試作システムに我々の部分印づけ法を組み込んだ．本節は，その設計と実験結果について述べる．

5.1 部分印づけ法の EusLisp への実装方法

我々は，EusLisp の試作システムをマルチ mutator，マルチ collector Lisp 処理系として設計した [12]．このシステムでは，複数の mutator と collector が並列に動作する．さらに，システムは collector の実行を制御するだけでなく，mutator と collector の動作個数も制御する．つまり，自由セルが十分残っている場合には，collector を mutator として機能させ，セルの消費に collector の回収が追いつかない場合には mutator を collector として機能させる．これらの複数の mutator と collector を制御するために，スケジューリング用のスレッドを用いる．このスレッドは，自由セルの量を監視し，ある制限値以下に減少した場合に collector を起動する．また，自由セルの残量や消費速度などの要因により，mutator および collector の動作個数を動的に制御する．

部分印づけ法の EusLisp への実装は，我々の実験システムと同様に行った．mutator を停止して行うルート挿入は実時間性の大きな問題となることから，特に注意深く設計した．実際には次のような手順となる:

step 1 スケジューラは mutator にルート挿入の開始を通知する．全 collector は次の通知が来るまで何もしない．

step 2 全 mutator は通知によって，同期をとられる（バリア同期）．

step 3 全 mutator は個別のルートセットを印づけスタックへ積む．その後，処理再開の通知が来るまで待つ．

step 4 スケジューラは大域的なルートセットを印づけスタックへ積む．

step 5 スケジューラは，mutator と collector に対して，処理の再開を通知する．

上記のスケジューラの各通知処理はシグナルを用いた．上記 step 1, 2, および 5 の実行時間は mutator の台数に依存する．step 3 の実行時間は各 mutator のスタックの深さに依存する．step 4 の実行時間は大域的なルートの数に依存するが，通常は定数となる．

5.2 実験および結果

Sun Microsystem 社のマルチプロセッサシステム (SPARC Center 2000, 16cpu) において mutator の数を変えながらそのルート挿入の時間を測定した．図 17 は，mutator の数とルート挿入の時間，シグナルによる通知の時間，バリア同期の時間のグラフである．

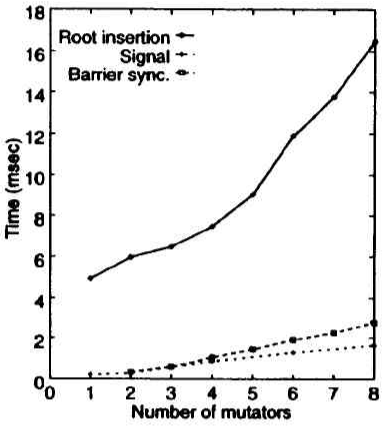

図 17 スレッドの数とルート挿入時間

Fig. 17 A relationship between the number of threads and root insertion time

mutator の個数が増えるにしたがって，シグナルの通知時間およびバリア同期の時間は増加する．これらは mutator の個数にほぼ比例するが，それに比べると，ルート挿入の時間の増加率は大きい．したがって，step 3 の処理時間が大きく関係していると考えられる．図 18 は，mutator が一台の場合の mutator のスタックの深さとルート挿入の時間の関係を示している．スタックの深さとルート挿入の時間はほぼ比例していることがわかる．また，数ミリ秒から 12 ミリ秒で終了することがわかる．前述したように停止型マーク法では数百ミリ秒かかるので，部分印づけ法を導入したことにより，mutator の停止時間は大幅に減少したといえる．

6. おわりに

本論文では，並列 GC の効率改善への取り組みについて述べた．並列 GC の基本的なアルゴリズムである IU 型および SB 型アルゴリズムの問題点を明らかにし，動作効率，書き込みバリア，mutator の停止時間の問題の存在を明らかにした．また，動作効率を改善する方法である部分印づけ法と相補型 GC の有効性を示した．

著者を含むグループは，残された課題である書き込みバリアの改善やルート挿入時間の短縮などの研究

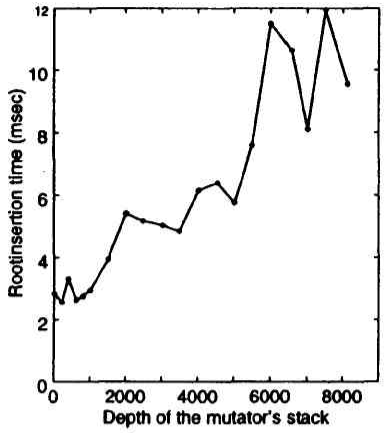

図 18 mutator のスタックの深さとルート挿入時間
Fig. 18 Depth of the mutator's stack v.s. root insertion time

テーマに取り組んでいる.

謝辞

本研究を行うにあたり，並列計算機を使わせていただいた電子総合研究所の松井俊浩様に感謝致します.

参 考 文 献

1) Matsui, S., Kato, Y., Teramura, S., Tanaka, T., Mohri, N., Maeda, A. and Nakanishi, M.: SYNAPSE — A Multi-Microprocessor Lisp Machine With Parallel Garbage Collector, *Lecture Notes in Computer Science*, Vol. 269, pp. 131–137 (1987).
2) Wilson, P. R.: Uniprocessor Garbage Collection Techniques, *Proceedings of International Workshop on Memory Management* (Bekkers, Y. and Cohen, J.(eds.)), Lecture Notes in Computer Science, Vol.637, Springer-Verlag (1992).
3) Lieberman, H. and Hewitt, C. E.: A Real-Time Garbage Collector Based on the Lifetimes of Objects, *Communications of the ACM*, Vol. 26(6), pp. 419–429 (1983).
4) Tanaka, Y., Matsui, S., Maeda, A. and Nakanishi, M.: Parallel Garbage Collection by Partial Marking and Conditionally Invoked GC, *Proceedings of the International Conference on Parallel Computing Technologies*, Obninsk, RUSSIA, pp. 397–408 (1993).
5) Tanaka, Y., Matsui, S., Maeda, A. and Nakanishi, M.: Partial Marking GC, *Proceedings of Third International Conference on Vector and Parallel Processing*, Lecture Notes in Computer Science, Vol.854, Linz, Austria, Springer-Verlag, pp. 337–348 (1994).
6) 田中良夫, 松井祥悟, 前田敦司, 中西正和: 部分印付けを併用した並列 GC の提案および効率の解析, 電子情報通信学会論文誌, Vol. J78-D-I, No. 12, pp. 926–935 (1995).
7) 松井祥悟, 田中良夫, 前田敦司, 中西正和: 相補型ガーベジコレクタ, 情報処理学会論文誌, Vol. 36, No. 8, pp. 1874–1884 (1995).
8) Matsui, S., Tanaka, Y., Maeda, A. and Nakanishi, M.: Complementary Garbage Collector, *Proceedings of International Workshop on Memory Management* (Baker, H.(ed.)), Lecture Notes in Computer Science, Vol. 986, Kinross, Scotland, Springer-Verlag (1995).
9) Yuasa, T.: Real-Time Garbage Collection on General-Purpose Machines, *Journal of Software and Systems*, Vol. 11, No. 3, pp. 181–198 (1990).
10) Hickey, T. and Cohen, J.: Performance Analysis of On-The-Fly Garbage Collection, *Communications of the ACM*, Vol. 27, No. 11, pp. 1143–1154 (1984).
11) Matsui, T. and Inaba, M.: Euslisp: An object-based implementation of lisp, *Journal of Information Processing*, Vol. 13, No. 3, pp. 327–338 (1990).
12) 高橋聡子, 岩井輝男, 前田敦司, 田中良夫, 中西正和: 並列 GC を備えた並列 Lisp システムの実装および評価, 電子情報通信学会論文誌, Vol. J80-D-I, No. 3, pp. 247–257 (1997).

付　　録

A.1 ルートセット走査

IU 型 GC では，ルートセットの書き換えによって印づけできないセルが生じる．図 19 に例を示す．a〜d のセルの印づけが終わった時点で，*1 のようにルート r1 を書き換え，つづいて*2 のようにセルを書き換えるとする．書き換えられたルート r1 には，印にないセルへのポインタが残り，残りのルート（r2, r3）に対する印づけを行っても f, g, h のセルには印はつかない．同様の問題は，ルート間のポインタのコピーや，セルの生成においても発生する．ルートセット走査は，このようなセルを発見するためのものである．

また，このようなセルへの追加印付けの際に mutator が動作していると，同じようなルート書き換えが発生する可能性があり，そのためのルートセット走査がさらに必要になる．すなわち，上のようなセルがルートセット上に存在するかぎり，フェーズが終了できないことになる．したがって，この操作は mutator を止めて行わなければならない．

A.2 並列 GC アルゴリズムのデータ構造

アルゴリズムは pascal 風の言語で示した．データ構造を図 20 に示す．共通の手続きを図 21 に示す．

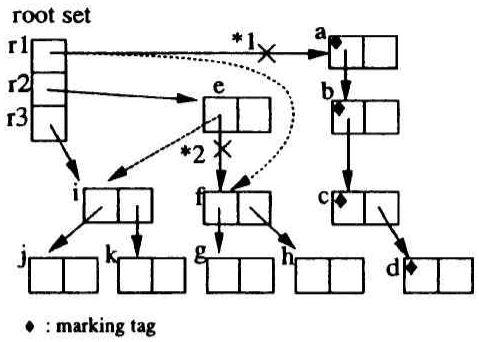

図 19 ルートの書き換え
Fig. 19 Rewriting of the root

並列 GC のアルゴリズムは，次のデータ構造を持つ．セルは 2 つのポインタフィールド (car, cdr) とカラーフィールド（color）を持つ．セルの総数は M である．ルートポインタは総数 R であり，配列 root に格納されているものとする．自由リストは 1 本のリストである．FREE_LEFT は自由リストの先頭のセルを，FREE_RIGHT は最後のセルを指す．また，自由リストのセルは，color は offwhite にセットされ，car には f という特殊なポインタが格納されている．フラグ PHASE は Synapse GC と部分印づけ法で用いる．フラグ CYCLE は部分印づけ法で用いる．フラグ REQUEST_PUSH と CONS_COLOR は相補型 GC で用いる．

mutator と collector の通信はスタックを用いて行う．このスタックの操作（push(), pop および stack empty のチェック）は不可分処理であるとする．また，suspend_mutator は mutator にリスト処理を中断させる手続きである．resume_mutator は mutator にリスト処理を再開させる手続きである．

LPa はポインタの書き換え処理（rplaca）を表す．rplacd も同様である．LPc はセルの生成（cons）を表す．それぞれ，root[m], root[n] のポインタを処理し，root[m] に値を返すものとする．

```
type t_pointer = 1..M ;
     t_root = 1..R ;
     t_color = ( black, white, offwhite ) ;
     t_phase = ( rootins, marking, sweep );
     t_cycle = ( full, partial );
     t_mode = ( incremental, snapshot, idle ) ;
     t_cell = record
              car, cdr : t_pointer ;
              color :    t_color
              end ;
var  cell : array[ t_pointer ] of t_cell ;
     root : array[ t_root ] of t_pointer ;
     FREE_LEFT : t_pointer ;
     FREE_RIGHT : t_pointer ;
     PHASE : t_phase ;
     CYCLE : t_cycle ;
     REQUEST_PUSH : t_mode ;
     CONS_COLOR : t_color ;
```

図 20 並列 GC アルゴリズムのデータ構造
Fig. 20 Parallel GC Algorithm (data structure)

```
procedure GP_mark( j : t_pointer ) ;
  begin
    while ( j <> NIL) and
          (cell[j].car <> f) and
          (cell[j].color <> black) do
      begin
        cell[j].color := black ;
        GP_mark(cell[j].car) ;
        j := cell[j].cdr
      end
  end ;
procedure GP_append( i : t_pointer ) ;
  begin
    cell[i].color := offwhite ;
    cell[i].car := f ;
    cell[FREE_RIGHT].cdr := i ;
    FREE_RIGHT := i
  end
```

図 21 共通の手続き (collector)
Fig. 21 Common procedure(collector)

松井　祥悟

昭和 34 年生．昭和 57 年慶應義塾大学工学部数理工学科卒業．平成 1 年同大学大学院博士課程単位取得退学．平成 1 年神奈川大学理学部助手，同専任講師を経て，平成 9 年助教授．工学博士．Lisp 処理系 (Lisp マシン，並列 GC) の研究に従事．情報処理学会，電子情報通信学会，ACM 各会員．

田中　良夫

昭和 40 年生．昭和 62 年慶應義塾大学理工学部数理科学科卒業．平成 7 年同大学大学院理工学研究科博士課程単位取得退学．平成 8 年技術研究組合新情報処理開発機構に勤務．平成 12 年通産省電子技術総合研究所入所．現在に至る．工学博士．Lisp 処理系 (主にガーベッジコレクション)，並列/分散/広域計算システム上での高性能計算に関する研究に従事．情報処理学会，ACM 各会員．

前田　敦司

1994 年慶應義塾大学大学院理工学研究科後期博士課程数理科学専攻単位取得退学．博士（工学）（慶應義塾大学 1997 年）．現在 筑波大学 電子・情報工学系 講師．並列/分散処理，コンピュータアーキテクチャ，プログラミング言語処理系，ガーベッジコレクションなどに興味を持つ．情報処理学会，日本ソフトウエア科学会，ACM 各会員

中西　正和

昭和 41 年慶應義塾大学工学部卒業．昭和 44 年慶應義塾大学工学部助手．平成 1 年慶應義塾大学理工学部教授．工学博士．昭和 42 年，日本初の実用 Lisp 処理系を作成．以後，記号処理言語，人工知能言語等の研究に従事．昭和 50 年プログラムの性質の自動証明系，昭和 57 年 Lisp マシン SYNAPSE の開発など．情報処理学会，電子情報通信学会各会員．前情報処理学会プログラミングシンポジウム委員会幹事長．

追　　記

本論文の著者の一人である中西正和慶應義塾大学教授は，2000 年 11 月 4 日逝去されました．本論文は 2000 年 6 月 30 日に作成しました．