記　　号　　処　　理　74－3
プログラミング－言語・基礎・実践－　17－3
（１９９４．　６．　３）

Partial Marking GC

田中良夫　　松井祥悟　　前田敦司　中西正和
慶應義塾大学　神奈川大学理学部　　慶應義塾大学

通常ガーベッジコレクション (GC) はリスト処理を中断して行なわれる．GC をリスト処理と並列に行なう (並列 GC) ことにより，GC による中断時間をなくし，リスト処理の実時間化が可能となる．並列 GC では GC の処理中にリスト処理によってデータが書き換えられるので，GC の正当性を保証するために特殊な処理が必要となる．そのため並列 GC は停止型 GC に比べてあまり効率が上がらず，実用化されているものもほとんどない．mark and sweep 方式の並列 GC においては，ゴミセルの回収効率が停止型 GC に比べて約 1/2 になってしまうことが知られている．これらの欠点の改善は，並列 GC の実用化へ向けての重要な研究テーマである．本論文では，mark and sweep 方式の並列 GC の欠点を改善した GC である，**Partial Marking GC(PMGC)** の提案，実装および評価に関する報告を行なう．PMGC は mark and sweep 型の並列 GC に世代別 GC の概念を導入した GC である．PMGC を実装し様々な実験を行なった結果，PMGC によってゴミセルの回収効率は従来の並列 GC に比べ最大で 2 倍に改善されることが確認された．PMGC は並列 GC の実用化に向けての有効な GC である．

Partial Marking GC

Yoshio Tanaka　　Shogo Matsui　　Atsushi Maeda　Masakazu Nakanishi
Keio University　Kanagawa University　　Keio University

A traditional parallel mark and sweep garbage collection (GC) algorithm has well known disadvantage: the collection efficiency is about 1/2 compared with sequential mark and sweep GC algorithm. In this paper, we propose a new parallel GC scheme called *Partial Marking GC (PMGC)* which is a valiant of generational GC. We also report an implementation and evaluation of PMGC. In the best case, PMGC is provided to be twice as efficient as the original parallel GC algorithm. It is easy to implement high performance parallel garbage collector on wide variety of multi processor machines using PMGC.

1 まえがき

Lispなどのリスト処理言語においては，**ガーベッジコレクション(GC)**によってリスト処理に中断時間が生じてしまう．この中断時間を短くするために世代別GC[5]などの高速なGCアルゴリズムが研究，開発されているが，これらのアルゴリズムを用いても中断時間を完全に取り除くことはできない．並列GCは，処理速度の向上とリスト処理の実時間処理(無停止処理)の実現が可能なGC方法である．汎用のマルチCPUマシンが次々と開発されている今日，並列GCの研究は重要な課題の1つであるが，現在はまだ効率の良い優れた並列GCが開発されておらず，実用化されているものもほとんどない．

GCの処理は，基本的には仕分けフェーズと回収フェーズの2つのフェーズに分けられる．仕分けフェーズではルートから到達可能なセル(生きているセル)とそれ以外のセル(死んでいるセル)を区別し，回収フェーズでは到達不可能なセルを回収し，自由セルとして再利用できるようにする．並列GCや実時間GCにおいては，GCの処理が行なわれている最中にリスト処理が行なわれる．リスト処理では新しいセルが生成されたり，リンクの書き換えやルートの書き換えが行なわれるので，GCの処理において「特別な処理」をしないと実際には生きているセルを「死んでいる」とみなして回収してしまう可能性が生じる．この「特別な処理」の方法に応じて並列GCのアルゴリズムはいくつかの種類に分類することができるが，現在並列GCで用いられているアルゴリズムのほとんどはSnapshot-at-Beginning型(SB型)[6]アルゴリズムである．たとえば，松井はSB型アルゴリズムを用いた並列GCであるSynapse GCを並列Lispマシン上で実装し，その評価を行なった[2]．湯浅のスナップショットGCは，SB型アルゴリズムを実時間GCとして実装したものである．Synapse GCとスナップショットGCはいずれもmark and sweep方式のGCをベースとしており，並列か実時間かの違いを除けば全く同じである．

SB型アルゴリズムでは，ルート挿入，印づけ，回収の3つのフェーズを1サイクルとし，GCの処理はこのサイクルを繰り返すことによって行なわれる．生きているセルを回収してしまうのを防ぐため，印づけや回収フェーズの間に生成されたセルは次回以降のGCサイクルで回収される．そのため，ゴミセルの回収効率はmark and sweep方式の停止型GCの約1/2になることが知られている[1]．SB型アルゴリズムを用いてリスト処理とGCを並列に行なった場合，回収効率の低下や並列処理によるオーバーヘッドなどの理由により，アプリケーションの実行にかかる時間や実時間処理の能力の点に関してあまり良い結果は得られず，停止型GCを行なう処理系よりも実行に必要な時間が長くなってしまう場合も確認されている[3]．mark and sweep方式やcopy方式など，各種のGCをリスト処理と並列に動作させる研究に関する報告はされているが，いずれにおいても並列処理を行なうことによるデメリットが大きく，アプリケーションを実行すると実行速度の面などにおいてあまり良い結果が得られていない．これらの並列GCに関する効率改善に関する研究はほとんどなく，リスト処理を並列に行なう処理系はいくつか存在するが，リスト処理とGCを並列に行なう処理系はまだ実用化されていない．

本研究の目的は並列GCの実用化であり，本論文ではSB型アルゴリズムの効率改善の方法として**Partial Marking GC(PMGC)**を提案し，その効果について報告する[4]．PMGCは，SB型アルゴリズムに世代別GCの考え方を導入したGCである．SB型アルゴリズムの通常のGCサイクルの間に，部分的なセルに対してだけ印づけを行う**partial marking**を含むGCサイクルを挿入する．このpartial markingでは直前のGCサイクルで生き残ったセルはその回のGCサイクルでも生きているとみなし，印づけを行わない．PMGCは，印づけにかかる時間を短縮することにより平均的なGC時間を短縮し，回収効率を上げる効果的な手法である．

2 Snapshot-at-Beginning型アルゴリズム

SB型アルゴリズムの流れ，特徴，欠点およびその評価に関して,Synapse GCの実装方法を例にして述べる．

SB型アルゴリズムの実装に際し，印づけ用のタグとしてblack，whiteおよびoff-whiteという3種類を使用する．off-whiteはフリーリストのセルに付加されるタグである．GCプロセス(GP)はルート挿入フェーズ，印づけフェーズおよび回収フェーズの3つのフェーズを繰り返す．印づけフェーズではルートからたどれるすべてのセルに対して印づけを行ない(タグをblackに変える)，回収フェーズではタグがwhiteであるセルをフリーリストへつなげ，タグがblackやoff-whiteのセルはフリーリストにつながっているセルを除き，タグをwhiteに変える．GPが印づけフェーズである時にLPがセルのポインタを切断した場合にはそのポインタをGPに連絡し，GPは切断されたセルに対しても印づけを行うようにする．また，consを行ってできたセルのタグはoff-whiteであるため，直後の回収フェーズでは回収されない．この仕組みにより，生きているセルが死んでいるとみなされて回収されることがなくなる．このように，SB型GCアルゴリズムは，簡潔であり，並列型および実時間型の実装も容易である．また，並列型の場合，LPにはルート挿入とポインタ切断時の処理が追加されるだけであり，LPのオーバヘッドが非常に小さい．

しかし，SB型アルゴリズムでは，ルート挿入の時点で生きていたセルと，印づけフェーズおよび回収フェーズの間にconsによって新たに生成されたセルは，直後の回収フェーズでは決して回収されない．直ちにゴミになった場合でも，その次の回収フェーズで回収される．しかし，印づけにかかる時間は停止型GCと全く同じであり，Hickey[1]が解析しているように，最悪の場合にはゴミセルの回収効率は停止型mark and sweep GCの約1/2となってしまう．これはほとんどのオブジェクトは生成されてから間もなく死んでしまうことを考えると[5]，大きな問題となる．印づけフェーズが終った時点では，印づけフェーズの間にconsによって生成されたoff-whiteのタグを持つセルが数多く存在する．それらのセルのほとんどは実際にはゴミセルであるが，それらのゴミセルは直後の回収フェーズでは回収されずにタグがwhiteに変えられるだけであり，実際にはその次の回収フェーズで回収される．我々が行なった数回の実験によればGC率(GC時間／全実行時間)の大きいアプリケーションの動作でゴミセルの回収が追いつかず，LPの動作が中断されることが頻繁に起こっている．

2.1 並列GCの評価

並列GCの効率を評価するために，GC率Gと改善率Iを次のように定義する．

$$G = \frac{T_{seq.gc}}{T_{seq.total}}, \tag{1}$$

$$I = \frac{T_{seq.total} - T_{para.total}}{T_{seq.total}} \tag{2}$$

但し，停止型GCを行なうLisp(seq-lisp)と並列GCを行なうLisp(para-lisp)においてアプリケーションを実行した場合の処理時間を次のように定める．

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

この改善率Iは，seq-lispの処理時間に対するpara-lispの処理時間の短縮の割合を表す．para-lispのLPのオーバヘッド時間を$T_{para.oh}$，para-lispとseq-lispのGC動作効率の比をnとし，オーバーヘッドの割合を$O = T_{para.oh}/T_{seq.total}$とすると，$I$は，次のようにまとめられる[3]．但し，動作させるLispプログラムは安定しており，定常的にゴミを出すと仮定する．

$$I = min(G - O, 1 - nG) \tag{3}$$

1LP対1GPおよび1LP対2GPで実行したSB型アルゴリズムの測定データを図1に示す．破線は1LP対1GPの理想的な並列GC($n = 1$, $O = 0$)の理論値である．結果を残さないconsを定常的に繰り返すeatcellをインタプリタを通して実行した．様々な長さの固定リストを置くことにより，GC率を変化させた．

グラフの単調増加部分(Gが0から極大部分まで)は，Hickeyらの論文のstableの状態(LPがwaitingを行わない状態)，単調減少部分のうちIが正の部分はalternatingの状態(2GCサイクルに1度LPがwaitingを起こす状態)，単調減少部分のうちの負の部分はcritical

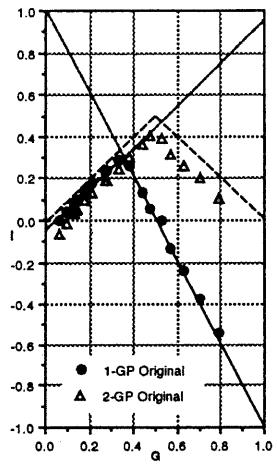

図 1: GC 率と改善率

の状態 (GC サイクル毎に LP が waiting を行う状態) である．このグラフのピークは実時間処理が可能な範囲を表す．1GP の場合，GC 率が 35%前後まで LP の動作は中断されず，2GP の場合は GC 率が 50%前後まで LP の動作は中断されない．

1GP のグラフより，n が約 2 であり，O は約 5%であることがわかる．2GP のグラフは，1LP 対 1GP の理想的な並列 GC とほぼ一致している．このことからも n が約 2 であることがわかる． また，この場合の O は約 10%であり，1GP の場合より大きくなっている．グラフの負の部分は停止型 mark and sweep GC の動作より処理速度が遅いことを示す．1GP のグラフでは，GC 率が大きい部分で負の部分が現れ，2GP のグラフでは，GC 率が小さい部分に負の部分が現れている．

以上より，SB 型アルゴリズムのゴミセル回収効率は，停止型 GC の約 1/2 であることがわかる．また，停止型 mark and sweep GC の動作より処理速度が遅くなる場合があることがわかる．

3 PMGC

PMGC は，SB 型アルゴリズムに世代別 GC の考え方を導入した GC である．印づけにかかる時間を短縮することによるゴミセル回収効率の上昇をはかった．

PMGC は，SB 型アルゴリズムに対し，GC サイクルの印づけの部分に変更を加える．数サイクルのうちの一回だけすべてのセルの印付けを行ない (full marking)，それ以外のサイクルでは一部のセルに対してだけ印付けを行なう (partial marking)．1 つの full marking を含む GC サイクルと，それに続く数回の partial marking を含む GC サイクルとで 1 つのシーケンスが構成され，PMGC はこのシーケンスを繰り返すことによって GC の処理を行なう．full marking は，通常の印づけと全く同じである．ルート挿入の時点で生きているすべてのセルをマークする．partial marking では，先の full marking でマークされたセルは現在も生きているとみなし，それらのセルに対する印づけは行わない．新たに印をつけるのは，前 GC サイクルで生成され，その回のルート挿入の時点で生き残ったセルである．full marking でマークされるセルは，generation scavenging GC[5] の長寿命セルとなる．partial marking を含む GC サイクルは，generation scavenging GC の生成領域に対する GC と同じである．PMGC は，数サイクルに 1 度，長寿命領域 GC を行う，周期の短い generation scavenging GC であると言える．partial marking では印をつけるセルの数が非常に少ないので，印づけにかかる時間も極めて短い．そのため，前回の印づけフェーズの間に生成され，印づけされる前に死んでしまったセル (生成後間もなく死んでしまうセル) をすぐに回収することができる．

PMGC の GP と LP のアルゴリズムを図 2と図 3に示す．

図に示したアルゴリズムでは，full markin を含む GC サイクル (full cycle) と partial marking を含む GC サイクル (partial cycle) を交互に行なうようになっている．セルは 2 つのポインタフィールド (`left`, `right`) と 1 つのタグフィールド (`color`) を持っている．セルの総数は `M` であり，フリーリストは `FREE` によって指されている．`FREE.left` はフリーリストの先頭を指し，`FREE.right` はフリーリストの末尾を指す．すべてのフリーセルの `left` フィールドには `f` という特別なポインタがしまわれている．もし GP が印づけフェーズの間に LP がポインタの書き換えを行なうと，切りとら

```
    procedure GP_root_insert;
      begin
        push all roots onto the stack
      end
    procedure GP_mark;
      begin
        while the stack is not empty do
          begin
            n := pop;
            while (n ≠ NIL) and
                  (n.left ≠ f) and
                  (n.color ≠ black) do
              begin
                n.color := black;
                push(n.right);
                n := n.left;
              end
          end
      end;
    procedure GP_collect_leave_mark;
      begin
        for i := 1 to M do
          if i.color = white then
            APPEND(i)
          else if (i.color = off-white) and
                  (i.left ≠ f) then
            i.color := white
      end;
procedure GP_collect_clear_mark;
  begin
    for i := 1 to M do
      if i.color = white then
        APPEND(i)
      else if i.left ≠ f then
        i.color := white
  end;
procedure GP_partial_marking_gc;
  begin
    while true do
      begin
        PARTIAL := true;
        GP_root_insert;
        GP_mark;
        GP_collect_leave_mark;
        GP_root_insert;
        GP_mark;
        PARTIAL := false;
        GP_collect_clear_mark;
      end
  end;
```

図 2: PMGC のアルゴリズム (GP)

```
procedure LP_rplaca(m, n);
  begin
    if PARTIAL then
      begin
        push(m.left);
        push(n);
        m.left := n
      end
  end;
procedure LP_rplacd(m, n);
  begin
    if PARTIAL then
      begin
        push(m.right);
        push(n);
        m.right := n
      end
  end;

procedure LP_cons(m, n);
  begin
    sleep while FREE.left = FREE.right
    NEW := FREE.left;
    FREE.left := FREE.left.right;
    NEW.left := m;
    NEW.right := n
  end;
```

図 3: PMGC のアルゴリズム (LP)

れたポインタだけではなく，新たに書き込まれたポインタも GP に渡され，そのポインタをルートとして印づけが行なわれる．これは generation scavenging GC の remembered set に対応する．

4 実装

PMGC は OMRON LUNA-88K ワークステーション上で実装された．LUNA-88K は 4 つの CPU を持ち，CMU で開発された MACH オペレーティングシステムを OS として用いている．MACH は**タスク**と**スレッド**を制御するための低レベルなプリミティブを提供している．1 つのタスク上で複数のスレッドを動かすことにより，共有メモリアーキテクチャ上での並列処理を容易に実現することができる．また，タスクとスレッドを制御するための高レベルなライブラリである Cthreads ライブラリも提供されており，今回の実装に際しては Cthreads ライブラリを使用している．

我々の実験では，1つのリスト処理のためのスレッド(LP スレッド) と 1 つの GC の処理のためのスレッド(GC スレッド) の 2 種類のスレッドを生成している．

我々は Lisp1.5 をベースとした Lisp を作成し，その Lisp 上で PMGC を実現した．GC スレッドが印づけフェーズである間にポインタの書き換えが行なわれた時に LP スレッドから GC スレッドに書き換えポインタを渡すためのスタックとして ps_stack を新たに追加した．また，GC スレッドがルート挿入フェーズに入ったことを LP スレッドに伝えるためのフラグとしてルート挿入フラグを追加した．

LP スレッドは GC の処理を行なわないという点を除き，通常の Lisp インタプリタと同様の処理を行なう．LP スレッドはルート挿入フラグがセットされているのを検出すると，すべてのルートポインタを root_stack と呼ばれるルート挿入用のスタックに積み込み GC スレッドに対してルートを積み終ったことを伝える．また，GC スレッドが印づけフェーズである間にポインタの書き換えが行なわれると，LP スレッドは切りとられたポインタと新たに上書きされたポインタの両方を ps_stack に積む．フリーリストが空である間は GC スレッドがセルをフリーリストにつなげるまで LP スレッドはブロックされる．

GC スレッドはルート挿入フェーズ，印づけフェーズ，回収フェーズの 3 つのフェーズを繰り返す．ルート挿入フェーズでは GC スレッドはルート挿入フラグをセットし，LP スレッドからルート挿入が完了したという通知を受けとるまで待つ．今回実現した PMGC は，図 2に示したアルゴリズムを用いている．つまり，full cycle と partial cycle を交互に行なっている．PMGC は直前のフルサイクルにおいて印づけフェーズで付けられた black のタグを回収フェーズで white に変えずに black のままにしておくことで実現することができる．オリジナルのアルゴリズムに対し，full cycle における回収フェーズの処理のみに変更を加えるだけで良い．full cycle の次の partial cycle では，生きているセルの中のほとんどのセルに black タグがついている (残されている) ために，ほんの少しのセルに印をつければ印づけフェーズが終了する．すなわち，印づけにかかる時間が極めて短くてすむ．その結果，以前の full marking フェーズの間に死んだセルを即座に回収することができる．印づけのアルゴリズムは full marking も partialmarking も全く同じで良い．

4.1 PMGC の評価

PMGC の効率を表すグラフを図 4に示す．アプリケーション等の動作条件は図 1と同じである．理想的な並列 GC ($n = 1, O = 0$) も付記した．

1GC プロセスの PMGC のグラフ (1-GP partial) は，従来の SB 型 GC(1-GP original) に比べ大きく改善されている．このグラフは理想的な曲線とほぼ一致し，2GC プロセスの従来の SB 型 GC(2-GP original) とほとんど同じである．1-GP original に見られるグラフの負の部分は解消され，停止型 mark and sweep GC の動作より処理速度が遅くなることはなくなった．このグラフのピークの位置は右に移動しており，G が約 0.5 の付近まで実時間処理が可能であることを示す．また，ピークの I の値も上昇し，約 0.4 になっている．これは処理速度に換算すると，並列 GC による Lisp で実行すると，停止型 GC を持つ Lisp で実行した時の約 60%の時間で実行が終了することを意味する．これは，1 つの LP スレッドと 1 つの GC スレッドの場合である．

2GC スレッドの PMGC では，さらに改善されている．G が約 0.6 を越えるまで実時間処理が可能となり，I の最大値は約 0.5 になっている．

G が 0.3 以下の部分では Iは，1-GP original，1-GP partial，2-GP original，2-GP partial の順に小さくなっている．これは，GC の動作効率が上昇するにしたがって単位時間あたりの GC の回数が増加し，セルへのアクセス競合が増えることによりリストプロセス側のオーバヘッドが大きくなったと考えられる．

5 GC 動作の抑制

図 4に示されているように，GC 率が小さい場合，GC の動作効率が上昇するに従いリストプロセス側のオーバヘッドが大きくなる．

GC 率の低いアプリケーションに対しても PMGC を

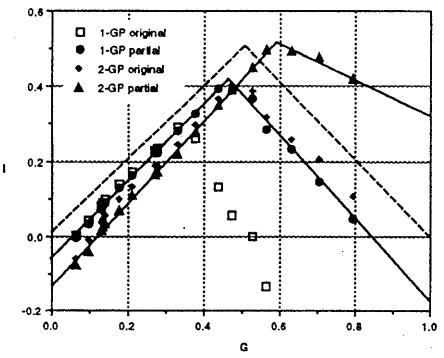

図 4: PMGC の改善率

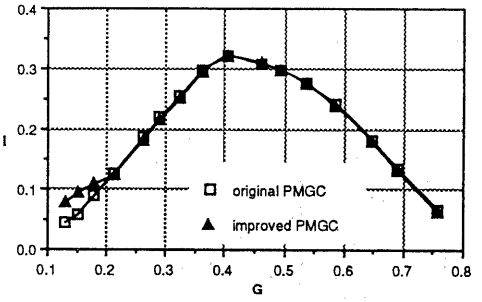

図 5: 改良版 PMGC の G と I

効率良く実行するために，フリーセルの残量が多い間は GC の処理を中断させておくことを考える．
そのためにフリーセルの残量を監視し，GC サイクルの起動を制御する．具体的には全セル数に対するフリーセルの個数の割合があるしきい値よりも低くなった時に GP を起動する．GC が終了したら再び GC が必要となるまで GP を停止させるようにする．
GP を起動する時のフリーセルの残量のセル総数に対する比率をしきい値とする．このしきい値を Invoking Threshold と呼ぶ．Invoking Threshold は，毎回の GC サイクルごとに集めたセルの数とそのサイクルの間に LP により消費されたセルの数により，次のように決定する．

$$invoking\_threshold = \frac{\text{消費されたセルの数}}{\text{回収したセルの数}} \quad (4)$$

(Invoking Threshold × セル総数) のセルを LP が消費する間に，GP はセル総数分のセルを回収できることになり，フリーセルがなくなって LP が待たされることはない．

GC 動作を抑制した PMGC の効率を表すグラフを図 5に示す．実験方法は図 1に示された実験と同じである．Invoking Threshold が小さく GC サイクルが停止するような場合には，full marking だけを行う．Invoking Threshold が大きく GC サイクルが停止しない状態の場合には partial markng GC を行う．

G が約 22%以下の場合は GC 動作を抑制した方が効率が良く，G が約 22%を越えたところでは GC 動作を抑制してもしなくてもほぼ同じ効率を示すことがわかる．
数種のアプリケーションを実行した際の実行時間を比較するグラフを図 6に示す．seq-lisp での実行時間を 1 とした時の相対時間を示している．実行したアプリケーションは，*Ackermann*(Ack) および *Boyer* である．Boyer に関してはインタプリタを通して実行したもの (Boyer) と，コンパイラによって実行したもの (C_Boyer) の 2 種類のデータを示した．左側の 3 つのグラフは GC 率が低い (約 10%) 場合のデータであり，右側の 3 つのグラフは GC 率が高い (約 40%) 場合のデータである．GC 率が高い場合には PMGC によって実行時間の点でも大きく改善されていることがわかる．GC 率が低い場合には，PMGC を用いた場合の実行時間はかえって長くかかってしまう場合もあるが，これは GC の効率が上がり，LP スレッドのオーバーヘッドが大きくなってしまったからである．しかし GC 動作を抑制することによって，GC 率が低い場合でも実行時間の点で改善されていることがわかる．

6 むすび

SB 型の並列 GC には，回収効率が停止型 mark and sweep GC に比べて約 1/2 であり，リスト処理プロセスに対してオーバーヘッドが生じるという欠点があった．我々はこれらを改善する方法として，**Partial Marking**

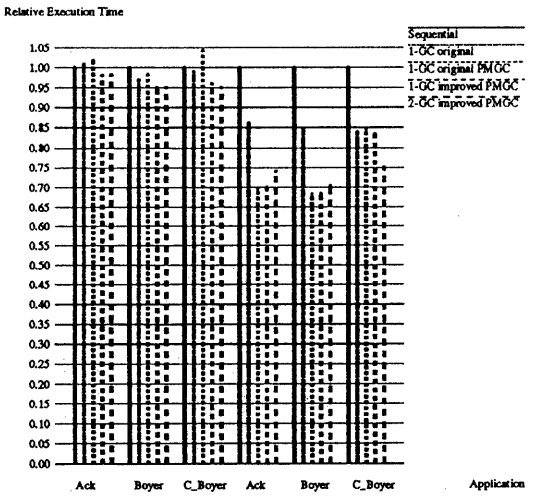

図 6: 実行時間

GC(PMGC) を提案した．PMGC は SB 型アルゴリズムに世代別 GC の考えを導入した GC である．通常の GC サイクルの間に，部分的なセルに対してだけ印づけを行なう **partial marking** を含む GC サイクルを挿入する．PMGC は印づけの対象を限定して印づけにかかる時間を短縮することにより，平均的な GC 時間を短縮し，回収効率を改善する．

GC 率が低い場合には処理速度の点においてかえって効率が低下してしまう場合があるので，GC 率が低いアプリケーションの実行の際には GC 動作を抑制することにより，GC 率が低い場合において効率が低下するのを防ぐことができた．PMGC に GC 動作抑制技法を加えることにより，今まで並列型 GC が停止型 GC に対して劣っていたゴミセルの回収効率を停止型と同等まで改善することができた．PMGC を用いれば，停止型 GC を持つ Lisp 処理系よりもアプリケーションの処理時間や実時間処理の性能の点において常に同等あるいはそれ以上の結果を得ることができる．PMGC は実現方法も容易であり，研究の目的であった「並列 GC の実用化」に対する有効な手段を得ることができた．

SB 型のアルゴリズムは複数の GC スレッドおよび複数の LP スレッドに対しても適用することができる．我々は複数のスレッドを使う並列 Lisp を実装することを考えている．まず始めはすべてのスレッドを LP スレッドとして動作させ，GC の必要が生じた時点でその LP スレッドのうちのいくつかに GC の処理をさせる．それらのスレッドは GC の処理が終ったら再び LP スレッドとしてリスト処理を行なう．すなわち，複数のスレッドに対して動的にリスト処理と GC の処理を割り当てることを考えている．

参考文献

[1] Hickey, T. and Cohen, J.: " Perfromance Analysis of On-the-Fly Garbage Collection ", Communications of the ACM, **27**, 11, pp.1143–1154 (Nov. 1984).

[2] Matsui, S., Teramura, S., Tanaka, T., Mohri, N., Maeda, A., Nakanishi, M.: " SYNAPSE: A Multi-micro-processor Lisp Machine with Parallel Garbage Collector ", Lecture Notes in Computer Science, 269, pp.131–137, Springer-Verlag (1987).

[3] 松井祥悟, 田中良夫, 前田敦司, 高橋尚子, 中西正和.: " 並列 GC Lisp の UNIX 上への実装と評価 ", 情報処理学会記号処理研究会報告, 67-5, pp.33–40 (Jan. 1993).

[4] Tanaka, Y., Matsui, S., Atsushi M., Naoko, T. and Masakazu, N.: " Parallel Garbage Collection by Partial Marking and Conditionally Invoked GC", Proceedings of the International Conference on Parallel Computing Technologies, **2**, (Obninsk, RUSSIA), pp.397–408 (Sep. 1993).

[5] Ungar, D.: " Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm ", ACM SIGPLAN Notices, **19**, 5, pp.157–167 (May 1984).

[6] Wilson, Paul R.: " Uniprocessor Garbage Collection Techniques", Lecture Notes in Computer Science, 637, pp.1–42, Springer-Verlag (1992).