計算機アーキテクチャ　83−35
（１９９０．７．１８）

共有メモリ型マルチプロセッサにおける並列ガーベージコレクション

渦原 茂
日本アイ・ビー・エム株式会社 東京基礎研究所

本稿では並列マークスイープ・ガーベージコレクションのアルゴリズムを提案し、そのアルゴリズムを共有メモリ型マルチプロセッサ上に実装した評価結果を報告する。本アルゴリズムの特徴は２つある。第１に、従来より提案された並列アルゴリズムでは予めガーベージコレクション用に割り当てられたプロセッサが常時処理を進めるのに対して、本方式ではガーベージコレクションを必要とする期間だけプロセッサを割り当てる。第２に、従来方式とは異なる３種類のマークを用いて使用中のセルをマークする。TOP-1マルチプロセッサワークステーションで測定したところ、ガーベージコレクションによるオーバーヘッドの94%を解消することができた。

A Parallel Garbage Collector on a Shared-Memory Multiprocessor

Shigeru UZUHARA

IBM Research, Tokyo Research Laboratory

5-19, Sanbancho, Chiyoda-ku, Tokyo 102, JAPAN

We propose a parallel mark-and-sweep garbage collection algorithm and report its performance evaluation on a shared-memory multiprocessor. The algorithm being proposed has two distinctive characteristics. First, whereas previous parallel algorithms assign certain processors to perform garbage collection constantly, our algorithm allocate processors only during the garbage collection phase. Second, the cells are marked with three colors of marks, different from the previous algorithms. The experiments conducted on TOP-1 multiprocessor workstation show that 94 % of garbage collection overhead can be eliminated using this algorithm.

1 はじめに

ガーベージコレクションの問題は、プログラムの実行中断とプログラムの実行時間全体の増加である。従来より、GC の処理をプログラムとは別のプロセッサに割り当てることによりこの問題を解決する、並列ガーベージコレクション (GC) のアルゴリズムが提案されてきた。本稿では、並列マークスイープアルゴリズムを説明し、それを共有メモリ型マルチプロセッサ上に実装した結果をもとに評価を行なう。本稿の構成は、2 章で GC の並列アルゴリズムを説明し、3 章でその動作特性を解析する。4 章でマルチプロセッサ上での実装と測定結果について考察する。最後に 5 章でその他の問題について議論する。

2 アルゴリズム

マークスィープ GC は、2 つのフェーズからなる。第 1 フェーズ (マークフェーズ) ではプログラムにとって必要なメモリセルにマークを付け、第 2 フェーズ (スイープフェーズ) ではメモリ内をスキャンしてマークの付いていないセル (ゴミ) を回収しフリーリストと呼ばれる場所に管理しておく。プログラムはフリーリストからセル (フリーセル) を取り出して使用する。

一括型 GC(Stop GC) では、フリーリストが空になるとプログラムの実行を中断して GC を実行し、GC が終了するとプログラムの実行を再開する。これに対して、並列 GC では、プロセッサは 2 つのグループ — プログラムを実行するグループと GC を実行するグループ (コレクタ) — に分かれており、2 つのグループは並行動作する。したがって、フリーセルを回収よりも速く消費しない (飢餓状態に陥らない) 限りプログラムの実行は中断されない。

本稿のアルゴリズムは、湯浅のインクリメンタル・マークスイープアルゴリズム [5] に基づいている。湯浅方式では 1 台のプロセッサがプログラムを実行しつつ少しずつゴミを回収する。本方式との基本的な違いは GC を別のプロセッサが行なう点である。また本方式は Kung & Song の並列アルゴリズム [7] に似ている。Kung & Song の方法との違いは、マークの種類と GC を開始するタイミングが異なることである。

2.1 GC の開始

従来より提案されてきた並列アルゴリズムでは、GC 専用のプロセッサが常時マークフェーズとスイープフェーズを繰り返す。本方式は、別の方法として、飢餓状態にならない範囲であれば、プロセッサを GC の期間だけ GC 用に割り当て、その他の期間はプログラムの実行に割り当てることを考える。本稿の方法では、フリーセル数がある閾値以下になったとき、最初にそれに気付いたプロセッサがコレクタを起動する。その閾値をどのくらいにすればよいかについては 3 章で解析する。

マークフェーズは、大域変数、スタックやレジスタが参照しているデータ (ルート) のマークから開始する。湯浅や Kung & Song の方法では、GC の開始時にルートを別のスタック (GC スタック) に退避しておき、マークフェーズではこの GC スタックに積まれたデータを GC スタックを使いながらマークしていく。そして、GC スタックが空になったときマークフェーズが終了する。

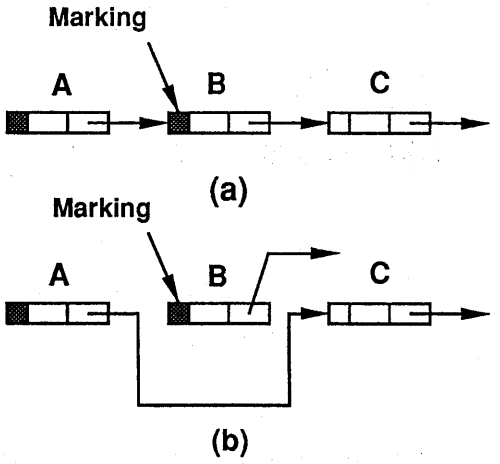

図 1: マーク洩れが生じる場合

2.2 プログラム側のオーバーヘッド

マークフェーズでは、プログラムがマークしているデータ構造を変更するために起こるマーク洩れが問題となる。例えば図 1(a) で、GC がセル A から B へとマークを付けているときにプログラムが図 1(b) のようにデータ構造を変更した場合、セル C は使用中にもかかわらずマークされない。

Dijkstra の方法 [4] では、セル C のように他から参照されるようになったセルをプログラムがコレクタ側に知らせてマーク洩れのないようにする。しかしながら、この方法では頻繁に起こるスタックやレジスタからの参照を含めたすべてのデータの参照が起こるたびにコレクタ側に知らせる必要がある。

これに対し 本稿でも採用した湯浅や Kung & Song の方法では、セルの内容が書き換える際に書き換える以前に参照していたセルの方をコレクタ側に知らせるようにする。図 1 では、セル B の内容を書き換えるときにセル B が指していたセル C をコレクタに知らせる。この方法では、セルの内容を書き換えるような操作に対してのみオーバーヘッドがかかる。セル C をコレクタ側に知らせるには、セル B を書き換える前にセル C を GC スタックに積んでおく。

2.3 マークの種類

本稿方式では、3 種類のマークを用いてセルをマークする。仮にそれらを GREEN, RED, BLUE としよう。GREEN の付いたセルはフリーセルを表す。RED と BLUE の意味は GC が開始されるごとに交換する。はじめは、すべてのセルはフリーセルで GREEN である。最初プログラムはフリーリストから取り出したフリーセルを RED にマークするが、実行が進んで GC が必要になったときから BLUE にマークする。コレクタも使用中のセルを BLUE でマークしていく。マークフェーズが終了した時点では、

BLUE は使用中のセルを、RED はゴミをそれぞれ表しているので、スイープフェーズでは RED のセルのみを GREEN にしてフリーリストに回収する。GC が終了したとき、メモリ中のセルは GREEN か BLUE のどちらかである。さらに実行が進んで次の GC の開始から、今度はプログラムは取り出したフリーセルを、コレクタは使用中のセルを RED にマークする。以下同様に RED と BLUE の意味を交換を繰り返す。

2.4 マルチコレクタ

複数のプロセッサが GC を行なう場合に注意すべきことは、GC のフェーズ切替えである。はじめにマークフェーズの終了は GC スタックが空になったときであることを述べたが、複数のコレクタが GC スタックをアクセスことによってまだマーク中のコレクタが存在するにもかかわらず、一時的に GC スタックが空になる状況が生じる。そこでマーク中のコレクタ数を表すカウンタを用意しておき、それが 0 かつ GC スタックが空の状態をマークフェーズの終了条件にする。ここで GC スタックからデータを取り出す操作とカウンタをアップする操作は非可分に行なうことに注意する。

スイープフェーズでは、メモリ領域を予め小さな領域(チャンク)に分けておき、複数のコレクタがチャンク単位にスキャンするメモリ領域を取り合いながらゴミを回収する。スイープフェーズはスキャンすべきチャンクがなくなれば終了してよい。ただし、もし次の GC を開始しなければならない時点でチャンクをスキャンしているコレクタが存在するならば、それらがスキャンし終るまで待って GC の開始を再検討しなければならない。

2.5 正当性

本方式における回収の特徴として、

1. GC 開始時に使用中のセルはすべてマークされる
2. GC 中にゴミになったセルはその GC 中には回収されず次の GC で回収される

の 2 つのことが言える。ルートとなるセルは GC 開始時にマークされる。それ以外のセルは他のセルから参照されるセルである。それらは、コレクタによってマークされるか、あるいは他のセルからの参照が破壊されるときに GC スタックに退避されてマークされる。参照の破壊を複数のプロセッサが同時に行なった場合でも、少なくとも 1 つのプロセッサが GC 開始時の参照を読み出して GC スタックに退避する。

3 解析

ここでは、プログラムがセルを消費する速度とコレクタがゴミを回収する速度との関係から、プログラムが飢餓状態に陥らない条件を求める。

セルの消費と回収の特性を表すために、6 つのパラメータ

N システム全体のセル数

L 使用中のセル数

M GC を開始するフリーセル数の閾値

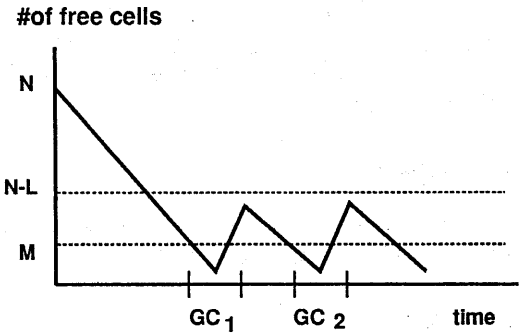

図 2: フリーセル数の変化

m コレクタが 1 つのセルをマークするのに要する時間

s コレクタが 1 つのセルをスキャンするのに要する時間

r プログラムがセルを消費する速度 (r 時間あたり 1 つのセルを消費する)

を用意する。これらを用いると、例えば GC の時間は、$mL+sN$ となる。

図 2 はプログラム使用中のセル数が一定 (L) とした場合の実行中のフリーセル数の変化を表したものである。フリーセル数が M になったときに GC を開始して (GC1)、終了すると次に M になるまで GC(GC2) を行なわない。

まず、GC 中に飢餓状態にならないための条件を求める。GC 開始時のフリーセル数が M であるから、M 個のセルをプログラムが消費する時間 rM がセルの回収され始めるまでの時間の最大値より大きい場合が求める条件である。

$$mL+s(L+M) \leq rM$$

$$M \geq \frac{m+s}{r-s}L$$

次に、GC 開始時のフリーセル数が M であることが前提であることから GC 終了時のフリーセル数は M 以上でなければならない。

$$N-L-\frac{mL+sN}{r} \geq M$$

$$N \geq \frac{rM+(r+m)L}{r-s}$$

L を最大値 (定数) と見なして、

$$M=\frac{m+s}{r-s}L$$

$$N=\frac{r(m+s)+(r+m)(r-s)}{(r-s)^2}L$$

これはシステムのメモリサイズの関係を示している。逐次 GC の生産性 (システムの実行時間のうち GC 以外の時間の占める割合) [9] は、

$$\frac{r(N-L)}{r(N-L)+mL+sN}$$

$$= \frac{r(1-\pi)}{(r+s)-(r-m)\pi}$$

である。ここで、$\pi = L/N$ (全メモリ量に対する使用できるメモリ量) である。並列 GC は飢餓状態に陥らなければ生産性は 1 である。

4 評価

ここでは、本稿のガーベージコレクタを実際に並列 Lisp 処理系 [8] に実装して測定した結果を示す。

4.1 実装

測定に使用したマシン TOP-1[14] は、インテル 80386 (動作クロック 14 MHz) 10 台をスヌープキャッシュ (128KB) によってバス結合した共有メモリ型マルチプロセッサである。

本方式の GC を実装した Lisp は、Kyoto Common Lisp [6] を基にマルチスレッドの機能を追加したもので、TOP-1 の複数のプロセッサを使用することができる。この処理系は TOP-1 OS[15] 上に作られているが、スレッド (Lisp のプロセス) の制御、同期や排他制御などは OS ではなくユーザレベルで実現している。[16] OS が提供するプロセスをプロセッサ台数分だけ用意しておき、それらが Lisp のプロセスコンテキスト (レジスタコンテキスト) を取り合いながら並列実行する。各プロセスは OS が提供する共有メモリの機能を用いてアドレス空間を共有し、そこに Lisp のデータなどを割り付ける (図 3) 。排他制御などは共有メモリに対する Test-and-Set 命令などを使って実現している。ここで複数のプロセスが同時にプロセッサに割り当てられることを想定して、排他制御の時間が短い部分ではビジーウエイトを行なっている。

メモリ管理は KCL の管理方法を流用している。メモリは一定の大きさ (2KB) のチャンクに分けられており、割り付けるセルのサイズがチャンクごとに決まっている。スイープフェーズでは、サイズの情報を基にチャンクごとにスキャンする。ただし、Common Lisp にはサイズの異なる可変長のデータタイプが存在するため、そういったデータ用に特別なチャンクを用意しておき、そこではデータごとにサイズを調べてスキャンする。

複数のコレクタが 1 つの GC スタックだけを用いてマークした場合、頻繁にアクセス競合が起こる。そこで GC スタックの構成は、1 つのグローバルなスタックと各プロセッサ (コレクタ) ごとにローカルな GC スタックを用意する (図 4) 。GC 開始時には、グローバルスタックにデータを積んでおく。コレクタはグローバルスタックからデータを取り出すと、しばらくはローカルスタックを使ってマークをし、ローカルスタックのデータ量が増えるとそれをグローバルスタックに戻す。

4.2 性能評価

3 章で述べたパラメータの値を、プログラム用に 1 台、GC 用に 1 台のプロセッサを割り当て、Boyer ベンチマーク [12] を使って調べると表 1 のような結果となった。特に r の値はセルの消費速度を表すものであるが、プログラム性質に依存する。1 つの目安として r の最小値を 1 つのフ

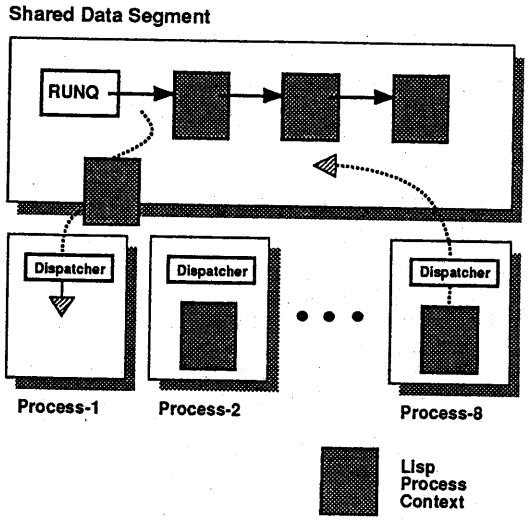

図 3: OS 上での実現方法

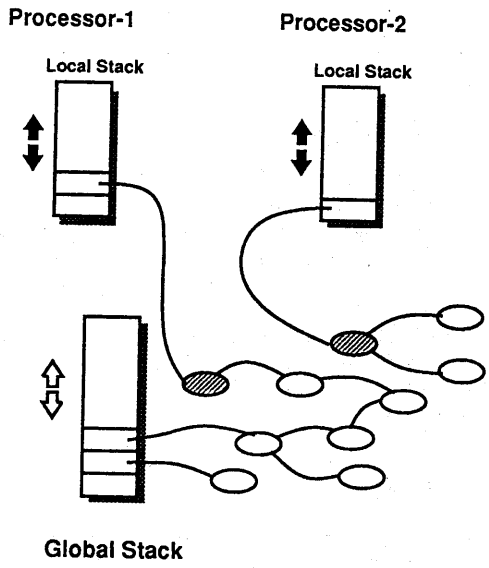

図 4: GC スタックの構成

1 つのセルをマークする時間 (m)	20 μsec
1 つのセルをスキャンする時間 (s)	5 μsec
1 つのフリーセルを消費する時間 (r)	100 μsec
メモリサイズ	1.4 M bytes
GC 開始時の平均スタックサイズ	280 words
マークしたセルの総数	50138
逐次 GC 時の実行時間 (GC 7 回)	67.4 sec (GC 14.1 sec)
並列 GC 時の実行時間	54.1 sec

表 1: 測定結果

リーセルを取り出す時間として測定すると、 20 μ sec であった。

これらの値をもとにシステムの他のパラメータを決めると、M = 0.26 L 、N = 1.54 L となり、GC を開始すべき閾値は全メモリ量の約 17%、プログラムが使用できるメモリ量 (π) は約 67%である。これらのパラメータから逐次 GC の生産性は、0.64 となるから 36%の GC オーバーヘッドの解消が並列 GC によって期待できる。

Boyer を測定した結果、逐次 GC でのプログラム実行時間は 67.4 sec で、そのうち GC が占める時間は 14.1 sec であった (プログラムが実動時間は 53.3 sec) 。一方、並列 GC での実行時間は 54.1 sec であった。逐次 GC の生産性は 0.79 でとなり推定とは異なっているが、プログラムの実行とコレクタの実行がオーバーラップして並行に動作していることがわかる。

図 5 のグラフは GC の処理を複数のプロセッサに行なわせた場合の台数効果を示している。この値は、プログラムがセルを消費する速度に応じてパラメータ m と s の値を小さくする場合に参考となる。スイープフェーズの台数効果に比べて、マークフェーズの台数効果が小さいことが分かる。

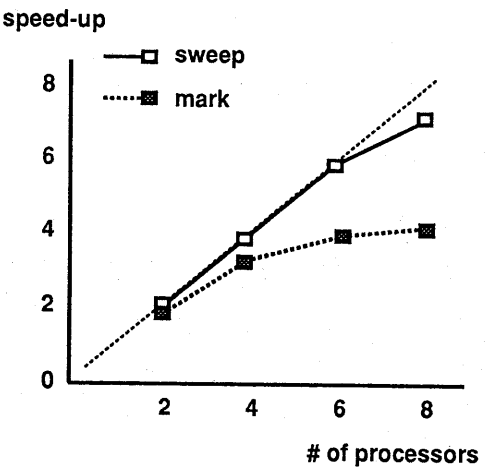

図 5: コレクタの台数効果

5 その他の問題

GC の方法としてセルを移動することによって領域を回収する方法 (Copy GC) がある。[1][2] この方法で並列 GC を行なう場合、プログラムからのアクセスとコレクタによる移動を排他的に行なわなければならない。専用のハードウエアなどを使わずにセルの移動を行なう優れた方法として Appel らの方法 [13] がある。彼らはページングのハードウエアを利用してページトラップを使ってデータのアクセスと移動の排他性を実現している。ただし OS はページトラップやページ管理情報をユーザレベルで高速に受け取る機構をサポートしなければならない。

マークスイープ GC では、全メモリ空間をスキャンするために、実メモリサイズを越えた仮想記憶において性能が低下する。これに対処する方法としてスキャンする領域を世代ごとに管理する Zorn の方法 [10] とマークするセルの領域を管理する小川の方法 [11] などが考えられる。ただし、前者はデータが世代を更新するときセルの移動を伴うため上で述べたのと同様の問題がある。後者の方法は特に仮想記憶を考慮して考えられたものではないが、図 6 のようにメモリ領域を世代に分けて管理する方法に相当する。

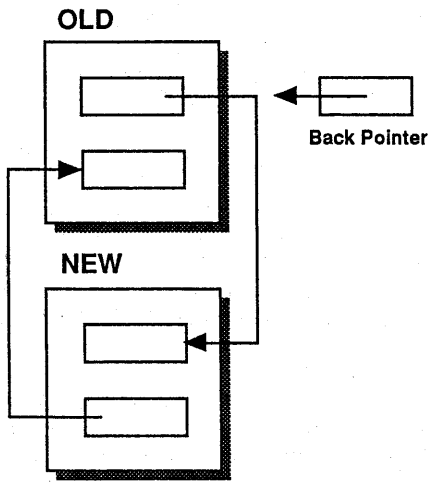

図 6: 世代別管理

[3] スイープフェーズでは新しい世代の領域のスキャンにとどめる。さらにマークフェーズではマークしていく過程で古い世代のセルに出会うと、そのセルが参照するセルのマークを打ち切る。ただし、プログラムは古い世代のセルが新しい世代のセルを参照するような古いセルへの変更が起こす場合に、そのことを記録しておく。小川の方法のように、世代の更新はセルを移動せずに領域単位で行なう。

6 おわりに

本稿では、並列マークスイープアルゴリズムの説明し、その実装した結果をもとに評価を行なった。マークスイープアルゴリズムはプログラム側のオーバーヘッドが少なく、汎用プロセッサ上で実現し易いアルゴリズムである。今回の実験ではGCによるオーバーヘッドを吸収することが確かめられた。今後の課題としては、従来方式のように常時GCにプロセッサを割り付けた場合の解析と本稿で説明した方法との比較を行なうこと、世代別管理がセルを移動せずに領域単位で行なった場合どの程度有効かを調査し、実装することなどが揚げられる。

7 謝辞

本研究の機会を与えて頂いたIBM東京基礎研究所の山内長承氏と鈴木則久所長に感謝いたします。

参考文献

[1] H.Baker: *List Processing in Real Time on a Serial Computer*, Communication of ACM, April, 1978

[2] R.Brook: *Trading Data Space for Reduced Time and Code Space in Real-Time Garbage Collection on Stock Hardware* ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, 1984

[3] H.Lieberman, C.Hewitt: *A Real-Time Garbage Collection Based on the Lifetimes of Objects* Communication of ACM, June, 1983

[4] E.W.Dijkstra, L.Lamport, A.J.Martin, C.S.Scholten, E.F.M.Steffens: *On-the-Fly Garbage Collection: An Exercise in Cooperration*, CACM, November, 1978

[5] T.Yuasa: *Realtime Garbage Collection on General-purpose Machines*, Technical Report 535, Research Institute for Mathematical Science, Kyoto University, 1986

[6] T.Yuasa, M.Hagiya: *The KCL Report*, Teikoku Insatsu Publishing, 1985

[7] H.T.Kung, S.W.Song: *An Efficient Parallel Garbage Collection System*, IEEE Symposium on Foundation of Computer Science, 1977

[8] M.Hozumi, T.Kurokawa, N.Suzuki, T.Tanaka, S.Uzuhara: *Multiprocessor Common Lisp on TOP-1*, US/Japan Workshop on Parallel Lisp, 1989

[9] T.Hickey, J.Cohen: *Performance Analysis of On-the-Fly Garbage Collection*, Communication of ACM, No.11, 1984

[10] B.Zorn: *Comparing Mark-and-sweep and Stop-and-copy Garbage Collection*, ACM Lisp and Functional Programming Conference, 1990

[11] 小川貴英: 印付けの対象領域を動的に管理するごみ集め, 情報処理学会記号処理研究会資料 45-4, 1988 年 3 月

[12] R.Gabriel: *Performance Evaluation of Lisp Systems*, MIT Press, 1985

[13] A.Appel, J.Ellis, K.Li: *Real-time Concurrent Collection on Stock Multiprocessors*, ACM SIGPLAN '88 Conference on Programming Language Design and Implementaton, 1988

[14] 清水, 他: 高性能マルチプロセッサワークステーション TOP-1, 並列処理シンポジウム JSPP'89, 1989 年

[15] 穂積, 他: TOP-1 オペレーティングシステム, 第 39 回情報処理学会全国大会, 1989 年

[16] 渦原茂, 森山孝男: マルチプロセッサシステムにおけるライトウエイトプロセス, 第 6 回ソフトウエア科学会全国大会, 1989 年