

オブジェクト指向 Lisp EusLisp への並列 GC の実装

田中良夫

tanaka@nak.math.keio.ac.jp

慶應義塾大学

松井俊浩

matsui@etl.go.jp

電子技術総合研究所

神奈川県横浜市港北区日吉 3-14-1 茨城県つくば市梅園 1-1-4

EusLisp は、幾何モデリング機能とマルチスレッドを用いた並列プログラミング機能を備えたオブジェクト指向プログラミング言語である。EusLisp の並列性能と実時間性を向上させ、ロボット制御等への適用範囲を拡大するために、マルチスレッドを用いた並列型のゴミ集めを導入した。このゴミ集めでは、ゴミ集め専用のスレッドと計算専用のスレッドを並列に走らせることにより、ゴミ集めによる中断時間を削減する。今回用いた Partial Marking GC に関しては、データ構造としてセルしか持たない単純な Lisp 処理系上での実装および実験によりその効果が報告されている。本研究においては並列ゴミ集めの実用化へ向けて、さまざまな機能を備えた EusLisp 上に並列ゴミ集めを導入する際の実装方法およびその結果に関して報告する。

Design and Implementation of Multi-thread Concurrent Garbage Collector in the Object-Oriented EusLisp

Yoshio Tanaka

tanaka@nak.math.keio.ac.jp

Keio University

Toshihiro Matsui

matsui@etl.go.jp

Electrotechnical Laboratory

3-14-1 Hiyoshi, Kouhoku, Yokohama 223, Japan 1-1-4 Umezono, Tsukuba, Ibaraki 305 Japan

EusLisp is a concurrent object-oriented programming language designed for robot programming. It has geometric modeling facilities and threads. We implement the parallel garbage collector on EusLisp to improve the ability of parallel and realtime processing. In our system threads for garbage collection and threads for list processing run concurrently, and reduce the disruption of list processing by garbage collection. Previous reports on the efficiency of parallel garbage collector considered the simple Lisp interpreter which have cell data type only. We report the implementation and the evaluation of parallel garbage collector in the practical Lisp system with rich set of data types.

1 はじめに

特にロボット研究への応用を目指して EusLisp と呼ぶオブジェクト指向型 Lisp を開発してきた [1, 2]. Common Lisp のほとんどの機能をオブジェクト指向の上に実現しており、さらにロボットがセンシングや行動を計画するために必要な 3 次元幾何モデルを定義・操作・表示する機能を備えている。また、マルチスレッドを用いた並列評価機構を実装しており、近年普及を見せしているマルチプロセッサ型のワークステーションにおいて、いくつかのベンチマークでは明らかな並列利得を得ることができた [3, 4]. しかし、メモリ管理がボトルネックとなって十分な台数効果比が得られない問題があることも明らかになっている。

EusLisp は、もともとは逐次型の言語として設計されており、メモリ管理によるオーバーヘッドが最小となるように、停止・走査型のゴミ集めを行っている。また、高いメモリ利用効率を目的として、フィボナッチバディメモリ管理と非複写型のゴミ集めを行っている。非複写型 GC ではポインタの付け替え作業およびポインタが変更されたことを他のスレッドに伝播させる必要がなく、マルチスレッドへの拡張には非常に有利に働く。メモリ効率が高く、複写を行わないので、ワーキングセットを比較的小さく保つことができ、数メガバイト程度のヒープ容量であれば、ゴミ集めは 0.3 秒 (Sparc-20) 程度で完了する。ワークステーションの前に座った人間を相手にした応用や、EusLisp の応用分野であるロボットの動作計画問題では、この程度の休止時間はほとんど気にならない。しかし、ロボットの制御にまで踏み込もうとすると、GC による休止時間は大きな障害となる。たとえば、フレームレートでの画像処理では 33msec 毎の応答が求められるが、0.3 秒の休止によって 10 枚の画像がとりこぼされ、対象の追跡が困難になる。アクチュエータのサーボを行おうとすれば、10msec 以下の応答が必須である。

もう一つの問題は、マルチスレッド並列を行う場合のメモリ管理負担の増大である。スレッドを増やすことで計算処理能力は数倍から数十倍まで拡大することができるが、それに伴って大量のメモリ要求が発生し、空きメモリを回収するために GC が頻繁に走ることになる。この GC がただ一つのスレッドによって処理されると、その間、他の計算スレッドは計算を進めることができない。また、ある逐次プログラムが最適に並

列化できたとしても、GC が逐次部分として処理されると、いくら並列度を上げて GC 時間以下に短縮することはできない。たとえば GC に全計算時間の 20 すると、5 倍以上の並列利得は得られないことになる。

そこでマルチスレッドとしての特質を生かして、計算と並行して別のスレッドがゴミ集めを行う並列ゴミ集めを導入する。並列ゴミ集めとしては、snapshot GC [7], Partial Marking GC [8, 9] や Complementary GC [10] のような方法が提案されており、中でも筆者らが先に発表した Partial Marking GC や Complementary GC は、高い効率が得られることが確認されている。今回、Partial Marking GC を EusLisp に適用し、比較的大きな応用を実行できる本格的な Lisp システムの上で本手法がどのような振る舞いを示すかを観測した。

2 EusLisp の概要

原始 EusLisp は、幾何モデラの効率的な実現を目的としていた。EusLisp のオブジェクト指向機能は、モデルの表現に適しており、Lisp のポインタ、リスト操作機能は面や稜線などの要素間のトポロジーの操作に都合がよい。

オブジェクト指向型 Lisp の効率的実行に重要なのは、メモリ管理と型検査である。EusLisp では、可変長のオブジェクトの割り付けと回収を効率よく実行するために、フィボナッチバディ法によるメモリ管理を行っている [1, 2, 4]. 動的に変化する型階層の中で、オブジェクトの属する型継承木を高速で判定する方法として、区分木の考え方にに基づき、2 回の整数の大小比較でクラスの包含関係が検査できる機構を実装している [1, 2].

EusLisp は、いくつかの点で Common Lisp と非互換である。EusLisp のオブジェクト指向は、単一継承であることとメソッドコンビネーションができない点で CLOS とは異なる。また、関数閉包 (closure) は、無限エクステンツを持ってない。さらに、有理数、複素数などのデータ型と多値が実装されていない。

一方、Common Lisp にはない機能として、幾何計算機能の組み込み、プロセス間通信機能、他言語プログラムインタフェース、Xwindow インタフェース、OpenGL/Mesa グラフィックスライブラリ、非同期入力などがある。すでに障害物回避軌道の生成、動作シ

ミュレーション、動作拘束の導出、把握動作計画などの研究において顕著な成果を上げて来ている [5].

EusLisp は, Sun/Sparc で開発され, SGI/mips, Windows, Linux/i486 などにも移植されているが, これから述べる並列ゴミ集めは, Solaris 2 オペレーティングシステムの上で実現している.

3 Partial Marking GC

並列ゴミ集めでは, リスト処理を行なうプロセス (ミュータータ) とゴミ集めを行なうプロセス (コレクタ) を並列に走らせる. Partial Marking GC は Snapshot 型アルゴリズムを改良したものであり, 停止・走査法に基づいている. 黒, 白および灰色の 3 種類の色によってセルの状態を表す. フリーセルの色は灰色である. また, cons などの関数によって生成されたセルの色は灰色のままにしておく.

コレクタはミュータータから印づけのもととなるルートを集めるルート挿入フェーズ, ルートから到達可能なセルに印をつける (セルの色を黒にする) 印づけフェーズ, セル空間全体を走査して印のついていないセル (白いセル) をフリーセルとして回収する回収フェーズからなる GC サイクルを繰り返す. 回収フェーズでは, フリーセル以外の灰色のセルは色を白に変える. また, 黒いセルを「白に変えてしまう」サイクルと「黒のままにしておく (印を残しておく)」サイクルを交互に行なう.

世代別ゴミ集めの考えに基づけば, 新たに生成された灰色のセルのほとんどはすぐにゴミセルになってしまうと考えられる. Partial Marking GC は, 黒いセルを黒のままにしておく (印を残しておく) ことにより, 引続き行なわれる印づけフェーズを短時間で終了させ, 「生成後すぐにゴミになったセル」を即座に回収することができる. Partial Marking GC を用いると並列ゴミ集めとしてはほぼ理想的な処理を行なうことができるが, 実験および簡単なモデルを用いた解析より明らかになっている.

4 EusLisp への並列ゴミ集めの実装

Partial Marking GC の実装に際し, 注意しなければならない点が 2 つある.

1. ルート挿入の際には, ミュータータのリスト処理は中断させる必要がある. ルート挿入時にヒープの snapshot をとる必要があるため, すべてのミュータータがルート挿入を終えるまでミュータータはリスト処理を中断させなくてはならない. 正確にはルートの書き換えやヒープ中のポインタ書き換えを行なうてはいけないということであるが, 実際にはスタックやジレスタなどはルートとして扱われるため, ルートの書き換えは頻繁に行なわれ, その際に毎回「ルート挿入中であるかどうか」のチェックを行なうことは大きなバリアになってしまうため, ルート挿入中はミュータータを止めてしまう.
2. ミュータータがセルに対してポインタの書き換えを行なった場合には, そのポインタをコレクタに通知する必要がある. rplaca, rplacd のように明示的にポインタ書き換えを行なう以外に, インタプリタが内部的にポインタの書き換えを行なった場合にも通知する必要がある. ただし, 通知する必要があるのはヒープ中のポインタの書き換えのみであり, ルートの書き換えの際には通知する必要はない.

具体的な実装方法は, 以下のようになる.

1. スケジュールスレッドの作成

ミュータータとコレクタの間でバリア同期をとったり, コレクタのフェーズ切り替えの管理を行なう必要がある. そのため, スケジュールスレッドを作成し, それらの管理はスケジュールスレッドに行なわせる. EusLisp が起動されると, まずスケジュールスレッドが起動される. 起動されたスケジュールスレッドはミュータータとコレクタを生成し, ミュータータやコレクタの間のバリア同期をとるために以下の動作を繰り返す.

- (a) ミュータータへのルート挿入要求発信
- (b) すべてのミュータータからのルート挿入完了の合図待ち
- (c) グローバルなルートの挿入
- (d) ミュータータとコレクタへのルート挿入完了の合図発信

(e) すべてのコレクタからの印づけ完了の合図待ち

(f) コレクタへの回収開始の合図発信

(g) すべてのコレクタからの回収完了の合図待ち

2. ミューテータの処理

基本的には、通常のリスト処理を行なっていれば良い。ただし、スケジューラからルート挿入要求が来たら、ルートポインタをゴミ集め用のスタックに積み、スケジューラからリスト処理再開の合図が来るまで待つ。各ミューテータがルート挿入の際に積むものは、各ミューテータがローカルに保持しているスレッドオブジェクトへのポインタ、ミューテータが持つスタックに積まれているもの、スペシャル変数、最後にミューテータがアロケートして得られたポインタのみである。

スナップショットをとるために、グローバルなポインタは全ミューテータのルート挿入が完了してからスケジューラが積むようにしている。システムが起動された状態では、トップレベルミューテータは48個のルートポインタを積み、30から40 μ sec程度の時間がかかる。また、新たにスレッドを作成すると、そのスレッドがアイドル状態にある場合には8個のルートポインタを積み、20 μ sec程度の時間がかかる。また、ポインタの書き換えを行なった場合には、捨てられたポインタと新たに書き込まれたポインタの両方をコレクタに通知する。

3. コレクタの処理

コレクタはスケジューラを通じて同期をとりながら、印づけと回収の処理を繰り返す。

スケジューラがミューテータにルート挿入の合図を送る方法としては、(1) スケジューラからミューテータにシグナルを使う方法と、(2) スケジューラがフラグを立てミューテータが時々フラグをチェックする方法の2通りが考えられるが、方法(1)と方法(2)を比較すると、方法(1)の方がオーバーヘッドが少なく、方法(2)の場合にはフラグをチェックする部分をミューテータが通らない限り、ルート挿入要求が検出できない。また、方法(1)の方が既存のシステムに手を加える部分が少なく済むと考えられるため、今回の実装においてはシグナルを用いてルート挿入を通知する。以上の方法に

基づいて EusLisp へ加えた修正点をまとめると以下のようになる。

1. データ構造の変更

黒、白、灰色の色を表すためのデータ領域を確保し、フリーセルの色を灰色にする。

2. ミューテータの変更

ルート挿入のシグナルハンドラの登録し、ポインタ書き換えの際にコレクタに通知する処理を追加する。

3. 新たなスレッド (スケジューラスレッドとコレクタ) の追加

4. コンパイラの変更

ポインタ書き換えが発生する場合にコレクタに通知するコードを生成するようにする。

複写法のゴミ集めを並列化する場合には、オブジェクトの位置が移動してしまうためにリードバリアが発生してしまい、修正点が増加するだけでなく、効率の改善も望めない。また、Partial Marking GC は「印を残す」という処理を必要とするため、複写法のゴミ集めを行なう処理系に対しては適用することができない。

しかし、EusLisp のように停止・走査型のゴミ集めを行なう処理系に対しては比較的容易に実装することができる。ミューテータに対してはルート挿入およびポインタ書き換え時の処理を追加するだけで良い。また、コレクタの処理は停止・走査型のゴミ集めとほぼ同じアルゴリズムで実装できる。前述のように EusLisp はバディ法によるヒープ管理を行なっているが、並列ゴミ集めの実装にあたっては回収処理の際にバディを管理するためのグローバルなデータベースに対して排他処理を必要とした他は、バディ法による付加的な処理は発生しなかった。

コンパイラに対しては、オブジェクトのスロットに対してポインタの代入を行なうコードを生成する部分に、代入のコードの前にポインタ通知を行なうコードを生成するように修正した。今回の実装では、Lisp で書かれた約 2400 行のコンパイラに対しては 8 箇所の修正を行なった。既存の処理系に並列ゴミ集めを実装するという点に関して述べると、すべてのソースコードを調べてポインタ書き換えを行なっている部分を検出するという作業が最も時間を要する。発生したバグ

の原因は、ほとんどがポインタの通知洩れであった。C 言語で書かれている EusLisp のカーネル部分は全部で約 20000 行あるが、ポインタの通知を必要としたのは全部で 139 箇所であった。

5 結果

5.1 中断時間

ゴミ集めを並列に行なう目的は、停止型ゴミ集めの場合に発生する、ゴミ集めによるリスト処理の中断時間の削減にある。本稿では、どの程度中断時間が削減されたかによって評価を行なう。前述のように、停止型のゴミ集めを行なう場合には最低でも約 300msec 中断時間が発生してしまう。Partial Marking GC では、ルート挿入の処理の際にスケジューラからのシグナルを受けとってから、全ミューテータがルート挿入を完了してスケジューラからリスト処理再開のシグナルを受けとるまでリスト処理が中断する。リスト処理の中断時間はミューテータの台数によって異なってくると考えられる。以下に、ミューテータの台数を変化させた場合のルート挿入による中断時間を表およびグラフで示す。実験は 4CPU 搭載の SS20 上で行なった。実行したアプリケーションは 8-queen である。横軸がミューテータの台数、縦軸が中断時間を表す。

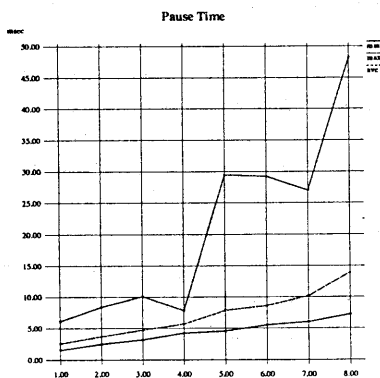


図 1: ルート挿入による中断時間
Fig.1: The Pause Time by Root Insertion

測定した中断時間は、スケジューラがミューテータ

にルート挿入要求のシグナルを送る直前から、全ミューテータからルート挿入を完了の合図を受けとってミューテータとコレクタに再開の通知を行なうまでの実時間である。結果のグラフより、最短中断時間と平均中断時間はミューテータの台数にほぼ比例していることが分かる。各ミューテータのルート挿入の手間はミューテータの台数に依存しないので、ミューテータの台数が多くなるにしたがってバリア同期にかかる時間が長くなるために、中断時間が長くなると考えられる。

実際にルート挿入の各処理ごとに時間を計測したが、ミューテータは 300 から 400 程度のルートポインタを積み、100 から 200 μ sec 程度の時間がかかる。このことより、全中断時間のほとんどはその前後のバリア同期のために発生していることが分かった。今回はバリア同期は mutex 変数と condition 変数を用いて実装しているが、ルート挿入による中断時間を短縮するためには、このバリアの機構をチューンアップする必要がある。しかし停止型ゴミ集めに比べると極めて中断時間が短縮されており、平均的にはミューテータ台数が 7 台まで、最悪でもミューテータ台数が 3 台までは、中断時間は約 10msec 以下でおさまることが分かる。

5.2 実行時間

ゴミ集めを並列に行なうことにより中断時間は削減されるが、ミューテータとコレクタとのアクセス競合や、ポインタ書き換えの際のポインタ通知などのミューテータにかかる付加処理のため、アプリケーションの実行時間は停止型ゴミ集めを行なう場合に比べて長くなってしまふことが多い。表 1 に停止型ゴミ集めを行なう EusLisp と並列ゴミ集めを行なう EusLisp で 11-queen および enya を実行した際の実行時間を示す。enya は CSG(Constructive Solid Geometry) の集合演算によって 500 程度の面からなるソリッドを Brep(Boundary Representation) として生成する、ソリッドモデラの基本的なプログラムである。11-queen はほとんどがコンスの消費だけのリスト処理を行なうものであり、enya はベクタなどの不定長の割り付けを行なうものである。

停止型 GC を行なう EusLisp で 11-queen および enya を実行し、それぞれが消費するメモリの量および GC にかかる時間を測定した。その結果、11-queen を 10 回実行した時のメモリの消費量は全部で約 240K バイトであり、GC の回数は 1 回、GC にかかる時間は 0.11

表 1: 11-queen と enya の実行時間

Table1: The Execution Time of 11-queen and enya

	11-queen	enya
停止型 (sec)	7.15	145
並列型 (sec)	14.57	150

秒であった。11-queen の 10 回の実行時間は合計で 71.5 秒であるので、1 秒あたり約 3Kbyte のメモリが要求されることになる。また、GC 率 (GC にかかった時間/リスト処理にかかった時間) は 0.15% とかなり低い。それに対し enya を実行した時のメモリの消費量は全部で 22M バイトであり、GC の回数は 92 回、GC にかかる時間は 25.06 秒であった。enya の実行時間は 145 秒であるので、1 秒あたり約 151K バイトのメモリが要求されることになる。また、GC 率は 17.3% とかなり高い。このことより、11-queen での実行時間の増加は、不要なゴミ集めの処理を行なうことにより発生するオーバーヘッドによると考えられる。enya の場合には、停止型ゴミ集めを行なうとゴミ集めに時間が全処理時間の 17% もかかってしまうが、ゴミ集めを並列に行なうことにより、この時間が短縮され、並列処理によるオーバーヘッドを帳消しにすることができる。

6 結論

本稿では、オブジェクト指向 Lisp EusLisp への並列 GC の導入に関してその方法および結果を示した。実験によると、並列 GC により発生する中断時間はミューテータの台数に依存するが、ミューテータの台数を制限すれば画像処理などの高い実時間性が要求されるアプリケーションにも Lisp が適用可能であることが示された。

今後、「回収処理の際にコレクタとミューテータの間で排他処理をしなくても済むように、buddy_base のエントリにフリーリスト末尾へのポインタを持たせる」「ゴミ集めの処理を必要としない時のコレクタの動作の抑制」などの改善を加える予定である。さらに、移動ロボットを動かしたり、グラフィックスを制御するなどの、より応用的なアプリケーションを実行し、実時間処理を必要とするアプリケーションに対する Lisp の可

能性を追求する予定である。

参考文献

- [1] 松井俊浩, 稲葉雅幸: EusLisp: オブジェクト指向に基づく Lisp の実現と幾何モデラへの応用, 情報処理学会記号処理研究会, SIGSYM, Vol. 89-SYM, No. 50-2, (1989).
- [2] Matsui, T. and Inaba, M: EusLisp: an Object-Based Implementation of Lisp, Journal of Information Processing, Vol. 13, No. 3, pp.327-338 (1990).
- [3] 松井俊浩, 関口智嗣: マルチスレッドを用いた並列 EusLisp の設計と実現, 情報処理学会論文誌, vol. 36, no. 8, (1995).
- [4] 松井俊浩, 関口智嗣: マルチスレッド EusLisp の分割型メモリ管理手法, 情報処理学会プログラミング研究会, SIGPRO, 3-3 pp.9-14 (1995).
- [5] 松井俊浩: オブジェクト指向型モデルに基づくロボットプログラミングシステムの研究, 電子技術総合研究所研究報告, 第 926 号, (1991).
- [6] Multithreading, SunOS 5.3 System Services, pp. 105-134, Sun Soft, (1994).
- [7] Yuasa, T.: Real-Time Garbage Collection on General-Purpose Machines, Journal of Systems and Software, Vol. 11, No. 3, pp. 181-198 (1990).
- [8] 田中良夫, 松井祥悟, 前田敦司, 中西正和: 部分印づけを併用した並列 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-1, No. 12, pp. 926-935 (Dec. 1995).
- [9] Yoshio Tanaka, Shogo Matsui, Atsushi Maeda and Masakazu Nakanishi: Partial Marking GC, Proceedings of International Conference on Third Joint International Conference on Vector and Parallel Processing (CONPAR 94 - VAPP VI), pp. 337-348 (Sep. 1994).
- [10] 松井祥悟, 田中良夫, 前田敦司, 中西正和: 相補型ガベージコレクタ, 情報処理学会論文誌, Vol. 36, No. 8, pp. 1874-1884 (Aug. 1995).