プログラミング 8－2
（1996. 8. 28）

Portableでrobustなglobal garbage collectorの構築について

遠藤 敏夫　田浦 健次朗　米澤 明憲
{endo,tau,yonezawa}@is.s.u-tokyo.ac.jp
〒113 東京都文京区本郷7-3-1
東京大学大学院理学系研究科

概要

並列計算機用のガーベージコレクション(global GC)を、並列言語で記述する方法を述べる。多くのglobal GCは逐次言語で記述されているため、計算機の通信方式などの差異の影響を受け移植性が低い、また分散メモリ計算機においては非同期メッセージのやりとりでアルゴリズムを記述しているため複雑になる、という問題を持つ。我々は並列オブジェクト指向言語Schematicのための GCをSchematic自身で記述する。計算機に依存しない並列言語によってGCを記述することにより移植性が得られ、remote procedure callなどの高レベルな通信機構を利用することにより簡潔にアルゴリズムを記述できることを示す。この方法により分散メモリ並列計算機AP1000にGCを実装し、性能評価を行なった。

Abstract

This paper illustrates a methodology for constructing a garbage collector on parallel computers. Many garbage collectors are written in sequential languages, therefore they are influenced by the difference between distributed memory and shared memory machine. We implemented a garbage collector for concurrent object-oriented language Schematic by using Schematic. We show that, by describing a garbage collector on the top of machine-independent language like Schematic, a garbage collector can be more portable and simple. We implemented a garbage collector on a parallel machine AP1000, and measured its performance.

1 はじめに

並列計算機用の汎用言語(手続き型、オブジェクト指向、関数型 など)を構築するにあたっての困難な部分の一つは、ガーベージコレクション(GC)の実装である。並列計算機における GC(global GC, 以下単にGCと呼ぶ)[5]を実装しようとする際の問題点として、以下のものが挙げられる。

開発効率の悪さ 特に分散メモリ計算機において、GCそのものの複雑さから来る開発の困難さが挙げられる。これまでに多くの方式が提案されているが、実装されているものは少ない。

分散環境のGCにおいては、オブジェクト探索の過程でリモートな参照を見つけた場合、参照先にメッセージを送って探索を依頼する必要がある。各プロセッサがメッセージのやりとりを行なうのだが、全ての探索が終るタイミングを知るのは容易ではなく、複雑な同期機構を導入しているなど、多くの場合実装が困難になっている。

移植性の悪さ これには以下の2種類考えられる。

計算機間の移植性 分散メモリ、共有メモリの差異など、通信方式が異なる計算機のために、それぞれ独立にアルゴリズムが提案されてきた。

言語の実装間の移植性 同一の計算機環境、言語のためのGCであっても、言語の実装の低レベルな層の変更に伴い、大幅に作りなおす必要がある場合が多い。

本来、並列オブジェクト指向言語などの汎用並列言語は、計算機間の差異を、言語が規定する共通な計算機モデルで吸収し、移植性の良いプログラム開発を可能にすることを大きな目的の一つにしている。この前提における自然な発想として、GCをそのような言語で記述するという提案ができる。これは、多くの言語システムで最も低レベルな部分と位置付けられているGCを、言語の他の機能の上に構築することを意味する(図1)。GCを計算機に依存しない高級言語で記述することによって、

- 単純で頑健なGCが分散メモリ計算機上に構築できる
- ソースコードの大部分を異なる計算機間で共有できる
- 単純なGCから出発して、性能改善のための変更を徐々に加えていくことが容易にできる

などの利点が生ずる。 最後の項目は特に分散メモリ計算機上のGCでは困難である.

本研究は、並列オブジェクト指向言語Schematic [6]のGCをSchematic自身で記述し実装することによって、GCの開発効率をあげることを目標とする。Schematicは計算機に依存しないプログラミングモデルを持つ、単純な並列オブジェクト指向言語であり、future 構文による同期/非同期 remote procedure call (RPC)や、その際の明示的なプロセッサ指定などを提供している。RPCの機構は、非同期メッセージの組合せでは書くのが複雑に

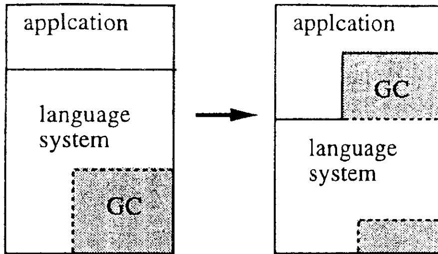

図 1: GC を言語の表層の上に実装

なる分散メモリ上での GC を簡潔に書くことを可能にしている。Schematic に限らず多くの並列オブジェクト指向言語はこのような機能をサポートしており、我々の以下の方式は上記の言語構文を備えた言語であれば、どのような言語でも当てはまるものである。

本論文の構成は以下の通りである．第2章で我々が GC の対象言語であると同時に、記述言語でもある Schematic について説明する。第3章で GC の移植性を確保するために、計算機の通信方式を捨象した抽象機械を考え、その上で我々の GC アルゴリズムについて概観する。第4章で我々の GC アルゴリズムの Schematic による記述を説明し、第5章で分散メモリ並列計算機 AP1000 上での実装の詳細を述べる。第6章で、その実験結果について報告し、第7章で議論を行なった後、第8章で関連研究、第9章で結論および今後の課題を述べる。

2 Schematic

Schematic は逐次言語 Scheme を拡張した並列オブジェクト指向言語である。本章では Schematic について、本論文の議論に関連のある部分を説明する。

チャネル チャネルは、プロセス間同期通信を行なうことができる通信媒体である。これは、後述の future 構文などで作成され、以下のように使用される。

`(touch c)` チャネル c から値を取り出す。c にまだ何も値が送られてなければ、値が来るまで停止する。

`(reply x c)` チャネル c に値 x を送る。

プロセスはチャネルに対し touch/reply を行なうことによって、同期通信を行なうことができる。チャネルは複数の通信を媒介するので、通信後であっても単純にはフリーできるとは限らない。一方、プロセッサ間で共有されることができるので、後に述べるように global GC の対象となる。

並列オブジェクト 並列オブジェクトは、プロセッサ間でデータを共有するための機構である。並列オブジェクトへのアクセスは、メソッド起動によって行なう。

GC を行なう観点から見ると、チャネルや並列オブジェクトは、複数のプロセッサに共有され、その後ゴミとなりうる。このようなゴミはプロセッサ内 GC のみでは回収することができず、global GC の対象となる。

関数 / メソッド呼び出し 関数やメソッド呼び出しは、future 構文によって非同期に実行できる。その際に暗黙のうちにチャネルが作られ、future の値はそのチャネルとなる。一方、呼び出された関数やメソッドは、自分の返り値を暗黙のうちにそのチャネルに送る。呼び出し側は future の後にチャネルを touch することによってその返り値を得る。`(touch (future (f x)))` を `(f x)` と略記することができ、Scheme の関数呼び出しと同様に記述できる。`(now (f x))` も `(touch (future (f x)))` の略記である。

また、`(future (f x))` や `(now (f x))` に `:on` オプションによってプロセッサを指定できる。`(now (f x) :on p)` は同期 remote procedure call に相当する。

Schematic は Scheme に対して単純な拡張を行なった言語であり、多くの言語を Schematic のように拡張することは可能である。

我々はこの言語を GC のターゲットとして、および GC の記述言語として用いる。以下で述べる GC 方式は記述言語が Schematic であることに強く依存しているわけではなく、GC の記述言語としては、同等な機能 (同期および非同期の RPC) を持つ言語であれば、以下の議論を適用することができる。

3 モデルとアルゴリズムの概観

本章では、計算機による通信方式の差を吸収する抽象機械を定義し、その上で我々の GC アルゴリズムを説明する。

3.1 抽象機械

n 台のプロセッサ上で、プロセスが動作し、同期 / 非同期の RPC が可能であるとする。ここでプロセスは、GC プログラムのプロセスを指す。

ユーザプログラムが直接使用しているオブジェクトをルートオブジェクトと呼ぶ。プロセッサはそれぞれルートオブジェクトとヒープを持ち、各ヒープは同じ大きさの半空間に 2 分されている。全てのオブジェクトはいずれかのプロセッサのヒープに属し、任意のプロセッサのオブジェクトへの参照を持つことができる。GC 開始時には全てのオブジェクトはどちらか片方の半空間に属しており、GC の期間中、そちらを各プロセッサの fromspace と呼び、もう一方を tospace と呼ぶ。ヒープの構造を、図 2 に示す。

また、GC プロセスはオブジェクトに対して、以下のような操作を行なうことができる。これらは抽象機械がプリミティブとして提供する。

- `(test-and-copy o)` fromspace 上のオブジェクト o が、forwarding pointer で上書きされていれば (初めて test-and-copy を適用されたならば)、すぐに真を返す。されていなければ、
  - o の複製を、同じプロセッサの tospace に作成する。
  - o を、新しいオブジェクトを指すポインタ (forwarding pointer と呼ばれる) で上書きする。

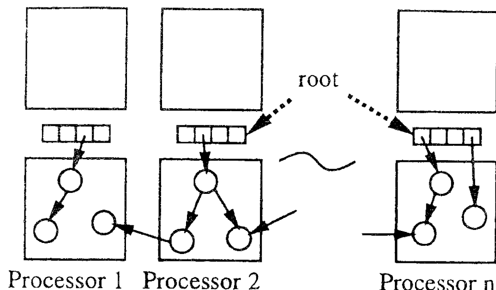

図 2: 抽象機械におけるヒープ構造 (GC 開始時)

- 返り値として偽を返す。初めに上書きされているかチェックしてから、コピー後に forwarding pointer を上書きするまでの動作を、他のプロセスに割り込まれずに行なう。

- (get-new-place o) fromspace 上で forwarding pointer で既に上書きされているオブジェクト o に対し、その forwarding pointer の指すオブジェクトを返す。
- (pe-of o) オブジェクト o が属するプロセッサを返す。

このうち (test-and-copy o) と (get-new-place o) に関しては、分散メモリの環境では o と同じプロセッサ上のプロセスのみが、一方共有メモリの環境では任意のプロセスが利用できるとする。ここで計算機の差異が現れてしまっているが、これに関しては 3.3節で述べる。

3.2 GC アルゴリズム

この抽象機械が提供するモデルの上で GC アルゴリズムの概略を述べる。GC の目的は、ルートオブジェクトから参照によって到達可能なオブジェクト (生きているオブジェクト) を全て tospace に移動させることである。なお、今回のアルゴリズムにおいては簡単のため、GC 中はユーザプログラムを停止させることにする。

1つリーダープロセスを設け、リーダープロセスは全てのプロセッサに対し探索プロセスを起動する。

各プロセッサ上で起動された探索プロセスは、それぞれのプロセッサのルートオブジェクトから再帰的に参照をたどり、探索過程で見つかったオブジェクト o に対して以下のような操作を行なう。

1. o に test-and-copy を適用し、o が初めて探索されたのなら以下の動作を o の複製に対して行なう。
2. o(の複製) が参照する全てのオブジェクト (子オブジェクト) について再帰的に探索を行なう。
3. その終了を待ち、o(の複製) に含まれる参照を、新しい子オブジェクトを指すように書き換える。

2. でリモートな参照を見つけた場合には、参照先のオブジェクトが属するプロセッサ上で探索を続けることにする。この場合でも呼び出し側はその終了を待ってから処理を続行する。ここにおいて、このリモートオブジェクトから到達できる全てのオブジェクトの探索が終了したことが保証されている点が重要である。この部分は同期 RPC を用いて実現されることを想定している。

リーダープロセスは全ての探索プロセスがオブジェクトの探索を終えるのを待つ。その後、fromspace と tospace の名前を交換し、GC は終了する。ユーザプログラムは、新しい fromspace(これまでの tospace) を対象に計算を再開する。

3.3 まとめ

本章で定義した抽象機械のモデルは、計算機による通信方式の差異などを覆い隠すものであり (7.1節で議論する)、このモデル上で GC アルゴリズムの議論を行なうことによってこのような差異に影響されずに議論することが可能である。また、この抽象機械はヒープの構造などを明示的に示しており、この上で GC アルゴリズムの議論を行なうことが可能である。3.2節で述べた探索アルゴリズムは抽象機械の機構のみを用いて記述することができる。

この抽象機械は、シングルプロセッサにおけるコピー方式の GC のモデルを拡張したものである。実際、ヒープの構造やプリミティブの意味などは、コピー方式を基としていることに依存している。しかし、マーク & スイープなどの他の方式であっても、本研究のように計算機に依存しない層で議論するという方法論や、test-and-copy のような競合状態をさけるプリミティブは、有効であると考えられる。

ここで、抽象機械の定義中に述べた、プリミティブ test-and-copy, get-new-place を実行できるプロセスの制限について補足する。分散メモリ計算機において、他プロセッサ上のオブジェクトに対しコピーなどの直接的な操作をするためには、なんらかの通信が必要である。プリミティブの内部での通信を避けるために、オブジェクトと同じプロセッサ上のプロセスのみがこれらのプリミティブを実行できる、という制限を設けた。一方、共有メモリ計算機において十分な実行性能を得るためには、この制限は厳し過ぎると考えられるため、この制限を外した。ただし、3.2節で述べた今回のアルゴリズムでは、必ず同じプロセッサ上でこれらのプリミティブを実行しているため、この議論とは関係ない。

これまで述べてきた事項とは別に、分散メモリの場合には飛行中メッセージの問題がある。これは、送信側プロセッサから送られ、受信側プロセッサにまだ届いていないメッセージは、生きているにも関わらずどちらのルートオブジェクトからも参照されない可能性があるという問題である。このような飛行中メッセージが探索されることを保証する必要があるが、抽象機械においてどう扱うか検討中の段階である。現在の実装では 5.2節のように対処している。

4 アルゴリズムの記述

3.1節で説明した抽象機械は、Schematic を含む多くの並列言語が想定する計算モデルと非常に似たモデルを提供している。本章では、3.2節においてこの抽象機械上で提案された GC アルゴリズムを、Schematic を用いて記述する。

From/tospace の存在や、オブジェクトの属するプロセッサは

Schematic の層からは見えないが、Schematic に 3.1節で定義したいくつかのプリミティブを追加すれば、抽象機械と等価なモデルのもとで GC プログラムを記述することができる。

3.2節のアルゴリズムの Schematic を用いた記述は、図 3のようになる (中心部分のみ示した)。これについて概略を述べると以下のようになる。

(traverse-object o)　o は任意のプロセッサ上のオブジェクトである。同期 RPC に相当する (now :on) を用い、o が属するプロセッサで traverse-object-local を呼び出す。

(traverse-object-local o)　o に test-and-copy を適用し、o が初めてコピーされたならば、

- プリミティブ get-new-place により o の複製を得る。
- o の複製 new-o が含む全てのポインタについて traverse-object-iter を呼び出す。

コピーされているか否かに関わらず、o の複製を返り値として返す。

(traverse-object-iter new-o idx num)　ここでは new-o が含む idx 番目のポインタについて操作を行なう。このポインタが指すオブジェクト child はまだ fromspace 上にあり、この child について、traverse-object を呼び出す。これによって再帰的に探索がなされ、返り値として tospace 上の child の複製が得られる。new-o のポインタを、この複製を指すように書き換える。

この中で、get-child, correct-child!, get-number-of-children は、オブジェクトに含まれるポインタの読み / 書き / 総数の取得を (vector-ref, vector-set!, vector-length のように) 行なう。

なおリーダープロセスは、future で全プロセッサに対し探索プロセスを呼びだし、touch でその終了を待つ。

このように Schematic を用いて抽象機械上で説明した GC アルゴリズムを記述した。

5　実装

これまで議論してきた GC を、分散メモリ計算機 AP1000 上に実装する。探索アルゴリズムについては第4章の通りに Schematic で記述する。本章では、Schematic の GC プログラムが利用する抽象機械の層の実現と、その他の事項について述べる。

5.1　抽象機械の実装

3.1節で説明した抽象機械の AP1000 上での実装について述べる。そこで定義したいくつかのプリミティブは Schematic の層では実現できない。これらについては、C 言語の関数群として作成する。

これらの詳細を説明するために、AP1000 などの分散メモリの環境において、Schematic のリモートオブジェクトがどのように実現されているかを述べる。

```
(define (traverse-object o)
  (now (traverse-object-local o) :on (pe-of o)))

(define (traverse-object-local o)
  (if (eq? (test-and-copy) #f)
      (let* ((new-o (get-new-place o))
             (num (get-number-of-children new-o)))
        (traverse-object-iter new-o 0 num)
        new-o)
      (get-new-place o)))

(define (traverse-object-iter new-o idx num)
  (if (= idx num)
      #t
      (let* ((child (get-child new-o idx))
             (new-child (traverse-object child)))
        (correct-child! new-o idx new-child)
        (traverse-object-iter new-o
                              (+ idx 1) num))))
```

図 3: Schematic で記述したオブジェクト探索ルーチン

プロセッサ 1 上で作られたオブジェクトに対しプロセッサ 2 から参照ができる際には、そのオブジェクトはプロセッサ 1 の輸出表と呼ばれるテーブルに登録される。輸出表もヒープのように 2 つ用意されている。プロセッサ 2 によって保持される遠隔ポインタは、本体の所有者であるプロセッサ番号 (この場合プロセッサ 1) と 2 つの輸出表の区別、輸出表におけるエントリから成る。

この環境において、プリミティブの動作を説明する。test-and-copy の動作は定義通りに記述できる。つまり、オブジェクトがまだ上書きされていなかったら、それを tospace に複製し、forwarding pointer を上書きする。実験に使用した Schematic コンパイラにおいては、C レベルで記述した関数は必ず不可分に実行されるので、単純に C 関数で記述すれば良い。get-new-place も、定義通りに forwarding pointer を得る C 関数を記述する。pe-of は、遠隔ポインタの場合はその中からプロセッサ番号を得てそれを返す。それ以外のオブジェクトの場合は pe-of が呼ばれたプロセッサ番号を返す。

ここで、第4章の記述について、遠隔ポインタの観点から論ずる。traverse-object-local の返り値として get-new-place の返り値が起動したプロセッサに渡される。その際、起動したプロセッサが遠隔プロセッサであった場合、輸出表への登録が新たに行なわれ、起動したプロセッサにはここで説明した (新たな輸出表エントリを含む) 遠隔ポインタが返される。

遠隔ポインタの表現に関しては、ここで述べられている方法の他に、共有メモリ計算機上で典型的なように、全てのヒープを直接アドレスで参照する場合もある。この場合先ほどの議論において traverse-object-local の返り値、すなわち get-new-place の返り値がそのまま起動プロセッサに渡される。このように我々の記述方法では遠隔ポインタの表現の違いが get-new-place のサポートにより吸収されている。

5.2 現在の実装の制限

現在のアルゴリズムではGCの期間中ユーザプログラムを停止させている。このとき、ユーザのプロセスとGCのプロセスを区別する必要があるのだが、現在は以下のような、言語システムに依存した対策をとっている。

- ユーザプロセスとGCプロセスを、区別してスケジューリングを行なっている。
- ユーザプログラムが使用し、GCの対象となるヒープ(3.1節で説明)と、GCプロセス自身が動作するためのヒープを、別にしている。

後者に伴い、GCプロセス自身が確保するオブジェクトはGCの対象とならないので、明示的にフリーされることが必要である。また、現在のSchematicの実装では、通信の際のチャネルなど、オブジェクトが自動的に確保されてしまう場合があり、これをフリーするコードを埋め込んだ。

3.3節に述べた飛行中メッセージの問題については研究中であるが、現在は以下のように対処している。低レベルな通信ライブラリの層において、各メッセージに対しACKメッセージを返すようにした。各プロセッサは、他プロセッサに送ったメッセージを全てテーブルに覚えておき、それに対応するACKメッセージが返ってきた時点でテーブルから削除する。GCが起こった時点でテーブルにあるメッセージは、ルートオブジェクトと一緒に探索の対象とされる。飛行中メッセージに対するこの対策はまだ洗練されておらず、効率的で柔軟なものにすべきである。

6 性能測定

本章では、第4,5章でAP1000上に実装したglobal GCの性能測定を行なった。プロセッサ数は1台または16台とし、ヒープの大きさは1つのsemispaceがそれぞれ1MBずつとした。プロセッサは25MHzのSparcプロセッサである。また開発期間の都合上、今回の実験で利用したSchematicコンパイラは、最適化などを行なっていない未熟なバージョンである。

実験にはフィボナッチ数を求めるプログラムなどを用い、引数の大きさなどを変えて複数回実行した。このプログラムは、再帰呼び出しの際に他のランダムなプロセッサに対しRPCを行なう。このプログラムは明示的にオブジェクトを確保するものではないが、再帰呼び出しの際に作られるチャネルやクロージャ(局所変数を保存するテーブル)などがゴミになる。

まず、今回の実装特有の事柄であるが、GC自身のメモリ使用量を調べ、GC開始時と終了時で変化していないことを確認した。これは、GCプログラムが確保するメモリは全て解放されていることを示している。

次に、一度のGCで生き残った(探索された)メモリ量と、GCにかかった時間の関係を調べた。その結果を図4に示す。プロセッサが1台の場合はメモリの量と時間はほぼ比例しているといえる。16台の場合の横軸は、生き残ったメモリが最も多かったプロセッサ1台のメモリ量である。この値を選んだのは、現在のアルゴリズムでは全プロセッサの探索が終るまで待つため、最も多くメモリを探索するプロセッサに実行時間が依存すると考えられるためである。この結果について、7.2節で論ずる。

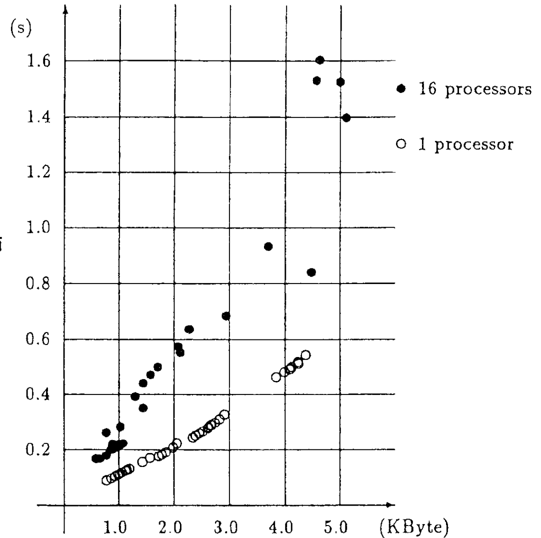

図 4: GCが探索したメモリ量と経過時間

7 議論

7.1 計算機間の移植性について

GCのアルゴリズムは、少なくとも概念的には非常に単純で、本質的には「オブジェクトをノード、オブジェクト間の参照を辺とする、グラフの探索」である。我々のアルゴリズムは、オブジェクトのマーキングを、オブジェクトに対するRPCとして表現することで、計算機の主記憶アクセス方式や言語処理系の通信実装の方式などに依存しないGCアルゴリズムの記述を得ている。

分散メモリ計算機においては、`traverse-object-local`はオブジェクトが割り当てられているヒープを持つプロセッサ上で実行されなくてはならないため，それを`:on`指定を用いて表現している。これは結局のところ、関数呼び出しがマーキングメッセージの送信および終了確認メッセージとの同期を、簡潔に表現したことになっている。一方、共有メモリ計算機上の実装においては、今回の実装のように分散メモリ計算機と同様に、`:on`指定を用いて他のプロセッサにマーキングを要求しても良いし、`traverse-object-local`を現在のプロセッサが直接実行しても良い。後者の場合、複数のプロセッサのGCが同じオブジェクトをほぼ同時期に発見した時に生ずる競合状態を、容易に実装できるプリミティブ(`test-and-copy`)を不可分とすることで調停している。

7.2 性能について

GCをSchematicで記述した場合、性能はCで書いた場合よりも悪くなることは十分考えられる。我々のとった方式がシステム全体の性能に致命的な影響を与えるかどうかは、現段階では不明である。今回の実験結果によれば、同じ量の生きているデータに対してGCに要する時間は、1プロセッサの場合と16プロセッ

サの場合とで結果は2倍程度しか変わらない。これは、少なくともこの実験においては、マーキングの遅延時間は, n プロセッサが並行動作することで隠蔽されており、それ以上の並列度抽出はあまり効果がないことを意味している。それよりも現時点では、ローカルな実行のオーバーヘッドに問題点があるといえる。今回の実験では、生きているデータが約4KB の時に、GC を終了するのに 25MHz の Sparc 上で約 500ms かかっており、これは実用的な速度ではない。しかし評価時に用いた Schematic のコンパイラは未熟なもので、現在のコンパイラ [8] はそれと比較して、10倍以上高速化されている。したがってこれらの数値は今後改善されていくものと思われる.

7.3 普遍性について

現在の我々の抽象機械のモデルや GC アルゴリズムは非常に素朴なもので、並列、分散環境において GC を本質的に複雑にするいくつかの問題を避けている。例えば、GC とユーザプログラム (mutator) の並行実行や、完了していない通信 (分散環境における飛行中メッセージや、共有メモリ計算機における store buffer 中の store 命令) の中に含まれる参照をどうマークするかなどの問題がある。これらの洗練されたアルゴリズムをも記述可能な、より普遍的な抽象機械を構築可能かはまだ明らかではない。例えば GC とユーザプログラムが並行実行できる方式 ([2, 4] など) を表現するためには、リードバリアやライトバリアなどのインタフェースを定義する必要がある。現時点でいえることは、それらのアルゴリズムを実装する場合でも、常に計算機間に渡ってできるだけ実装を共有できるように構築すべきであり、それは我々が今回述べた方法で可能になるであろう、ということである。つまり、機械依存部分を覆い隠すインタフェースを定義し、その上にマシン非依存な高級言語で GC を記述することで、異なる計算機間でコードを共有できまた、特に分散メモリ計算機上で、単純に実装することができるということである。

8 関連研究

分散環境における global GC としては、[1, 3] など多数提案されている。実装されたものとしては [7] などがある。[7] の方式はデータ駆動並列計算機上に実装され、GC とユーザプログラムが並行して処理を行なうことができる、またバリアなどのユーザプログラムに課するオーバーヘッドが小さい、という特徴を持つ。

9 おわりに

本研究では、移植性が高く、簡潔な global GC の構築法を提案し、実際に並列言語 Schematic のための global GC を分散メモリ並列計算機 AP1000 上に構築した。

計算機による記憶方式、通信方式の差異を吸収した抽象機械を定義し、その上で GC アルゴリズムを考えることにより、計算機に依存せずに GC アルゴリズムについて議論することができた。この抽象機械は、Schematic などの多くの並列言語と同様なモデルを持ち、ここで議論されたアルゴリズムはそのような並列言語で自然に記述することができる。また、RPC などの高レベルな通信機構を利用することによって、簡潔に GC アルゴリズムを記述することができた。

今後の課題としては、実際に他のマシンへの移植を行ない、本研究の構築法の移植性を実証したい。さらに、より性能の高い GC を実装し、性能評価を行ないたい。具体的には、GC のプロセスとユーザプロセスを並行に動作させる、通信量を削減をするなどの改善が考えられる。また、飛行中メッセージなどの問題に対する検討をすすめたい。今回は単純な深さ優先探索アルゴリズムを実装したが、global GC の研究がすすむにつれて新しいアルゴリズムがこれからも提案されることが予想される。本研究の方針に基づき、それらのアルゴリズムの効率的な実装、研究が可能になることが期待される。

参考文献

[1] Lex Augusteijn. Garbage collection in a distributed environment. *LNCS 259*, pages 75–93, 1987.

[2] Henry G. Baker, Jr. List processing in real time on a serial computer. *Communications of the ACM,21(4)*, pages 280–294, 1978.

[3] John Hughes. A distributed garbage collection algorithm. *LNCS 201*, pages 256–272, 1985.

[4] James O'Toole and Scott Nettles. Concurrent replicating garbage collection. *1994 ACM CONFERENCE ON LISP AND FUNCTIONAL PROGRAMMING*, pages 34–42, 1994.

[5] David Plainfossé and Marc Shapiro. A survey of distributed garbage collection techniques. *LNCS 986*, pages 211–249, 1995.

[6] Kenjiro Taura and Akinori Yonezawa. Schematic: A concurrent object-oriented extension to scheme. Technical report, Department of Information Science, Faculty of Science, University of Tokyo, dec 1995.

[7] 八杉 昌宏. データ駆動並列計算機上の並列オブジェクト指向ガーベジコレクション. オブジェクト指向計算ワークショップ WOOC96, March 1996.

[8] 大山 恵弘, 田浦 健次朗, 米澤 明憲. 動的なスレッド生成をサポートする言語のコンパイル技法. SWoPP'96, August 1996.