データ駆動並列計算機上の分散並行ガーベジコレクションの評価*

八杉 昌宏
神戸大学工学部情報知能工学科†

概要
データ駆動並列計算機上の並列オブジェクト指向言語処理系実装において開発した、並列オブジェクトのガーベジコレクション (GC) 方式とその評価について述べる。実装・評価した GC 方式は、(1) マーク&スイープに基づき循環した参照を持つゴミも回収する、(2) 通常のプログラムの実行を長時間ブロックせず、並行して処理を行う、(3) 通常のプログラム実行に課するオーバーヘッドが小さい、という特徴を持つ。これらの特徴を満たすため、本 GC 方式では、並列オブジェクトの内部に対してオーバーヘッドの大きいリードバリア・ライトバリアを用いずに、個々の並列オブジェクトを単位として一括処理を行う。実装と評価は、電総研で開発された非共有メモリ型のデータ駆動並列計算機 EM-4 上で行った。

Abstract
A garbage collection (GC) scheme of concurrent objects which is developed for a concurrent object-oriented language system on a data-driven parallel computer and its evaluation is presented. The GC scheme features that (1) it can collect garbages which have cyclic references, (2) it runs concurrently and does not block the normal computation for a long time, and (3) it does not impose large overhead upon the normal computation. These are realized by using atomic processing for each object rather than using read-barrier/write-barrier inside the object. The implementation and evaluation were performed on the ETL's distributed-memory data-driven parallel computer EM-4.

1 はじめに

本論文では、並列オブジェクト指向言語処理系 ABCL/EM-4[7] における並列オブジェクトのゴミ集め (GC) 方式 [8] の評価を行う。ABCL/EM-4[7] は、データ駆動並列計算機 EM-4[3] を対象マシンとし、並列オブジェクト指向言語 ABCL/ST[4] をソース言語とする処理系である。我々は ABCL/ST コンパイラ等の実装 [4] を行ってきた。

[8] で提案した GC 方式は、循環した参照を持つゴミも回収するため、リファレンスカウント方式とはせず、マーク&スイープに基づくものとした。GC 方式の設計においては次の点を考慮する必要がある。

- 並列オブジェクト指向言語の実行中には多数のアクティビティが存在し、これは実際に並列計算機により並列実行される。また、共有メモリ型ではないので分散 GC が必要となる。
- 対象とした並列計算機はデータ駆動計算機であるため、通常計算の処理とゴミ集めのための処理はハードウェアにより自然と並行実行される。このため、通常のプログラムの実行と並行してゴミ集め処理を行う並行 GC が必要となる。つまり、通常計算を止めて行う GC 方式は適用できない。(逆に、データ駆動計算機で実装可能な方式は、性能の問題を除けば通常の並列計算機で利用できる)。

また、並列オブジェクト指向言語では並列オブジェクトやメッセージを用いたモデルについてガーベジコレクションを考えねばならず、その設計を困難にしている。本論文では、並列オブジェクト指向言語モデルを mutator とセルからなる単純で、本質的なモデルの視点から捉えることで、単純な基本アルゴリズムの並行化を応用して分散並行ガーベジコレクションの設計を行う。

以下、2 節では、並列オブジェクト指向言語処理系 ABCL/EM-4 について述べる。3 節では、4 節で並列オブジェクト指向言語用の GC アルゴリズムについて述べるための準備として、単純で、本質的なモデルにおけるゴミの定義と並行 GC についてまとめる。4 節では並列オブジェクト指向言語用の GC アルゴリズムについて提案し、5 節ではその実装について述べる。6 節では、実際に実装を行い並列計算機 EM-4 で実験した評価結果について述べる。

2 並列オブジェクト指向言語処理系

この節では言語の計算モデルと、EM-4 のアーキテクチャについて述べる。

*Evaluation of Distributed Concurrent Garbage Collection on a Data-driven Parallel Computer

†Masahiro YASUGI, Department of Computer and Systems Engineering, Kobe University

2.1 並列オブジェクト指向言語 ABCL/ST

ABCL/ST[4] は、ABCL/1[5] に部分型を含む静的な型付けなどを導入した言語である。

ABCL/ST の計算/プログラミングモデルでは、並列の単位となるのは、独自のメモリと計算能力を持つ多数の並列オブジェクト (以下しばしばオブジェクトと略す) であり、これらはメッセージパッシングのみを通して互いに協調しながら並列に計算を行う。また、オブジェクトは動的に生成することができる。

ABCL では、オブジェクトは状態変数とメソッドを持つ。オブジェクトは休眠、待機、活性のいずれかのモードにあり、休眠状態でメソッド実行可能な要求メッセージを受け取ると、活性モードとなってメソッドを実行し、メソッド実行の完了とともに休眠モードに戻る。また、メソッド実行中に別のメッセージが必要な場合は、そのメッセージを受け取って実行を再開するまでの間、待機モードになる。

メッセージ送信には、past 型 (非同期メッセージ送信) と now 型 (非同期メッセージ送信+返答メッセージ待ち) がある。非同期に送られた要求メッセージは、受け手オブジェクトで、メッセージキューに入れられ、順番に処理される。また返答メッセージは、返答メッセージ待ちオブジェクトの実行を再開させる。

2.2 EM-4 のアーキテクチャ

コンパイラは、電総研で開発され、稼働中の細粒度ハイブリッド並列アーキテクチャEM-4[3] のアセンブリコードを生成する。EM-4(プロトタイプ) は 80 個の要素プロセッサ (PE) からなり 12.5 MHz のクロック速度で動作し、細粒度の通信メカニズムを提供する。例えば、レジスタ上から FIFO 性を持つ高速なオメガ網にデータを直接送出する 2 ワードパケット出力命令などがある。EM-4 のハードウェアはまたそのデータ駆動 (パケット駆動) の特徴と組み合わせることにより、複数個のスレッドの基本的スケジューリングメカニズムも提供する。ハイブリッドというのは、制御フローアーキテクチャとデータ駆動アーキテクチャの融合を意味する。

3 ゴミ集めの基本アルゴリズム

ここでは、4 節で並列オブジェクトの GC について説明する準備として、より基本的なモデルを用いて用語を整理する。

3.1 ゴミの定義

以下のような単純なモデルを考える。計算 (通常のプログラム実行) は **mutator** が、セルの集合 G を用いて進める。各セルは 0 個以上のスロットから成る。スロットには、G に属するどれか一つのセルへの参照 (ポインタ) が保持される。そのセルを、そのスロットから参照されるセルと呼ぶこととする。mutator に属するスロット (レジスタ等に対応) をルートと呼ぶ。mutator は、ルートから参照されているセルのスロットの内容 (参照) をルートにコピー (リード) したり、逆にルートの内容をルートから参照されているセルのスロットにコピーしたり (ライト) することができる。

このとき、ある瞬間に生きているセルはルートから参照を辿ることで到達可能なセルで、次のように再帰的に定義できる: (1) ルートから参照されているセルは生きている。(2) 生きているセル (のスロットのどれか) から参照されているセルは生きている。生きているセルでないなら、そのセルはゴミである。

3.2 ゴミの検出

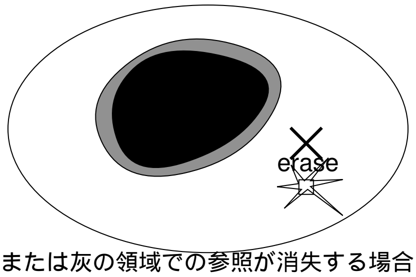
スロットに白、灰、黒の色のどれかを付けるものとして、ゴミの検出アルゴリズムを記述可能であるが、紙面の都合より割愛する。スロット毎にではなく、セル毎に白、灰、黒の色のどれかを付けるものとすると次のようになる。ただし、検出アルゴリズムにおいては、必要に応じてルートもスロット数が 1 のセルと考えることとし、通常のセルと同様に色を付ける。

この 3.2 節では mutator を停止させて行う GC について考える。次の手順で **collector** は、セルをゴミとゴミでないものに分けることができる。
[ゴミ検出の基本アルゴリズム]

1. ゴミ検出の開始時点においては、すべてのルートは灰、その他すべてのセルは白とする。
2. 灰色のセルがなくなるまで、灰色のセルを一つ選び o とし、次の処理を繰り返す:
   (1) o のすべてのスロットにつき、そのスロットから参照されているセルをマークする。ここでセルをマークするとは、セルが白だったら灰にすることをいう。白でなければそのままにする。
   (2) o を黒にする。
3. ゴミは白として残り、ゴミでないセルは黒となる。

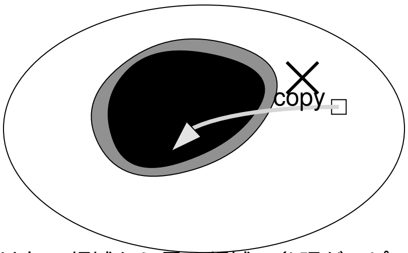
ゴミを検出中、灰または黒のセルは生きており、白のセルは生きているかどうかまだわからない。上の手順では、黒のセルが直接、白のセルを参照することはない。つまり、白のセルは灰のセルから参照されることがあっても黒のセルから参照されることはない。このため、灰色のセルがなくなった時点で白のセルは生きているセルから参照されていない、つまりゴミであることがわかる。

上の手順では色は濃くなる方向に変化する。マーク&スイープの言葉を借りて、白だったなら灰へ変



白または灰の領域での参照が消失する場合、
消失しようとする参照の参照先をマークする

図 1: 消失バリア



白または灰の領域から黒の領域へ参照がコピーされる
場合、コピーされる参照の参照先をマークする

図 2: コピーバリア

化させることをマークすると呼ぶ[1]。一方、灰から黒への変化 (上の手順における o に対する (1), (2) の処理) については、一般的な用語がないようなので、ここでは浄化すると呼ぶことにする。セルの灰から黒への変化において、そのセルから参照されるセルはマークされる。つまり、セルを浄化することで、そのセルの内部にはマークされていないセルへの参照が存在しないことが保証される。

ここで述べた検出アルゴリズムにおいてはルートにも色を付けたが、ルートには色を付けずに、1. と 2. の間でルートから参照されるセルを灰にする処理を行うほうが、どちらかというと一般的である。しかし次節以降で述べる並行 GC に、ルートから参照されるセルを灰にする処理も並行に行う場合も含めるにはルートにも色を付ける必要がある。

3.3 並行ゴミ検出

ゴミ検出が終了するまで mutator を停止させたままにしないで、ゴミ検出処理と並行動作させることができる。

並行動作させる場合、mutator の一部の操作に関して collector の処理への協力を得るためのバリアが必要になる。バリアには良く知られているように [9, 10]、(a) スナップショットを保存するものと、(b) 黒のセルが直接、白のセルを参照しないようにするものがある。

(a) は、次のような見方ができる。mutator が動作することで、参照のコピー操作やオリジナルの参照 (=ゴミ検出開始時に保持されていた参照) の消失が起こるが、このうち参照のコピー操作は自由とするが、まだその先のマークが行われていないオリジナルの参照が消失するのを防ごうというものである。このため、白または灰の領域でのオリジナルの参照の消失においては、その消失しようとする参照の参照先をマークする。(図 1のようなバリアを張る)。また黒の領域でのオリジナルの参照の消失はそれを辿るマークがすでに済んでいるので問題ない。このバリアにより、オリジナルの参照からなる参照関係のスナップショットに対して GC を行うことになる。

一方 (b) は、次のような見方ができる。参照のコピー操作やオリジナルの参照の消失のうち、参照の消失は自由とするが、まだその先のマークが行われていない参照が既に浄化したはずの黒の領域にコピーされるのを防ごうというものである。このため、白または灰の領域から黒の領域への参照のコピーにおいては、そのコピーされた参照の参照先をマークする。(図 2のようなバリアを張る)。また黒の領域からの参照のコピーはそれを辿るマークがすでに済んでいるので問題ない。このバリアにより、変化していく参照関係に対して GC を行うことになる。

(a) のバリアを消失バリア、(b) のバリアをコピーバリアと呼ぶこととする。並列ゴミ検出を実現するにはどちらか片方のみを用いればよい。([9] のように、両者を異なるフェーズで組み合わせて用いる方式の提案もある)。また、GC 開始時に mutator の処理や collector の他の処理に優先してまずルートをすべて黒にすることで、(a) ではルートでの参照の消失に関するリードバリアをなくして通常のセルへのライトバリアのみとすることができ、一方 (b) ではルートへのリードバリアのみとすることができる。

新しいセルの生成では、初期値をルートからコピーすることで初期化するとともに、新しいセルへの参照がルートにコピーされるものとする。ゴミ検出中に生成された新しいセルの色は次のようにする。(a) の消失バリアを用いる方式では、新しいセルは検出の対象外ですべて生きているとする。新しいセルはマーク処理や浄化処理をすることなく黒とすればよい。これは、新しいセルの生成に使われた参照のオリジナルは必ずマークに使われるからである。一方 (b) では、新しいセルも検出の対象し、その色は、新しいセルへの参照を保持することになるルートが必ず (あるいは既に) 黒であるかどうかに応じて二通り考えられる。保持ルートが黒でなければ、検出完了までにゴミになるかも知れないので新しいセルは白としておくのがよい。しかし保持ルートが黒であれば新しいセルへの参照の保持の際にコ

[1] この論文で用いる「マークする」という用語は必ずしもマークビットを立てるという意味ではなく、collector が処理の対象として認識するという意味である。コピー GC では、コピーすることで「マーク」を行う。

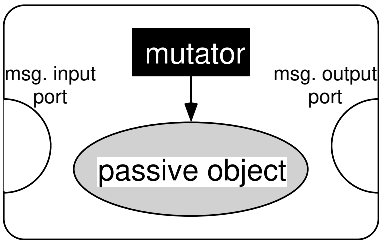


図 3: 並列オブジェクトのモデル

ピーバリアにより新しいセルはマークされ、灰になる。この場合、できれば新しいセルをあらかじめ黒にしておいたほうがよい。初期化に使われる参照をルートからコピーするときに、もし初期化に使われる参照も黒なら、初期化のコピーに対しコピーバリアはマーク処理をしなくてよいからである。

4 並列オブジェクトのゴミ集め

ここでは、3 節で述べた方式を用いて、並列オブジェクト指向計算におけるガーベジコレクションについて述べる。

4.1 ゴミの定義

並列オブジェクト指向計算においては、これまで見てきた単一の mutator とセル集合だけを扱う場合より複雑なモデルが必要になるが、以下では、並列オブジェクト指向計算モデルを解析して、mutator とセルからなるモデルの視点から捉えることで、3 節の考え方が適用できるようにする。その結果、ルートもしくは生きているセルから参照されるセルが生きていて、それ以外はゴミであるという 3 節での定義をそのまま利用できる。

モデルを並列オブジェクト空間とメッセージ空間 (ネットワーク) に分けて考える。

並列オブジェクト空間には並列オブジェクトが存在する。各並列オブジェクトは受動的オブジェクト (セル) と (仮想)mutator の組であると考える。(そのほかのメッセージキューなどは受動的オブジェクトの一部とみなして以後無視する)。ここで、オブジェクトの数だけの mutator を考えていることに注意してほしい。この mutator は、組となっているセル以外の他のオブジェクトのセルに直接アクセスすることはない (カプセル化)。また、この mutator はメッセージを作ってネットワークに送り出したり、ネットワークからメッセージを取り込んだりできる (図 3参照)。

さらに、並列オブジェクトには mutator に属するスロット (ルート) が存在し、アクティブな並列オブジェクトでは、受動的オブジェクトは mutator(のルート) から参照されていると考える (図 3参照)。つまり、並列オブジェクトがメッセージを受けるなどしてアクティブになっている間は、mutator からその受動的オブジェクトへの参照が存在し、逆にメッセージ待ちなどの非アクティブな状態になっている間は、mutator からその受動的オブジェクトへの参照は存在しないものとする。もし mutator からその受動的オブジェクトへの参照が常に存在しているものとしてしまうと、そのままでは、全ての受動的オブジェクトが生きていることになってしまうため、ここでは以上のように捉えることとした。また、mutator(のルート) と受動的オブジェクトは独立に色を持つものとする。

メッセージ空間 (ネットワーク) には、飛行中メッセージが存在する。各飛行中メッセージは、受動的メッセージ (セル) とそれを運ぶためのそれぞれの mutator の組になっていると考える。そして、並列オブジェクトの場合と同様に、それぞれのメッセージでは、mutator (のルート) から受動的メッセージへの参照があるものと考える。また飛行中メッセージはそのメッセージがオブジェクトに取り込まれた時点でメッセージ空間からは消滅するものとする。

ここではたくさんの mutator が存在しているわけだが、ゴミの定義は、3.1 節の場合と同じで、ゴミとは (いずれかの mutator に属する) ルートから参照を辿ることで到達可能な受動的メッセージ・受動的オブジェクト以外である。すべての飛行中のメッセージ、およびアクティブなオブジェクトは生きており準ルートと考えることもできる。

4.2 分散並行ゴミ検出

ゴミ検出も基本的には 3.2、3.3 節と同様である。つまり、4.1 節で述べたようにオブジェクトまたはメッセージの mutator に属するルートを灰、その他の受動的オブジェクトや受動的メッセージを白のセルとして [ゴミ検出の基本アルゴリズム] を用いる。

ただし、オブジェクト空間やメッセージ空間は、collector にとっても分散しているので、collector も複数個で分散して処理を行うものとする (それぞれが灰色のセルを選んで浄化していく)。また、オブジェクト空間は分散しているので、受動的オブジェクトの現在の色というものを系全体で同時に観測することはできない。提案方式では、受動的オブジェクトの色は次のようにして決める。まず、受動的オブジェクトと同じ場所に受動的オブジェクトが黒であることを示す情報は存在することとする。また、collector が系のどこかで、オブジェクトをマークしようとしたときには、オブジェクトが存在する場所へとマークメッセージを送出するものとする。提案方式ではマークメッセージ送出時点においてマークが行われた (つまり受動的オブジェクトが白だったら灰になった) と考える。つまり、灰になった時点ではオブジェクトが存在する場所では灰になったことを示す情報はなくてもかまわないものとする。

一方メッセージについていえば、collector はネッ

トワーク内部をアクセスできないと仮定しなくてはならないので工夫が必要となる。さらにいえば、受動的メッセージを運んでいる mutator を停止させるのも難しい。そこでここでは、全 mutator を停止させるタイプの GC については考えずに、並行ゴミ検出について考える。このとき 3.3 節で場合分けしたように、(a) 消失バリアベースか、(b) コピーバリアベースかの二つが考えられる。(a) をベースにするとスナップショットに対して GC が行われることになる。[6] はこのタイプの GC の終了判定の効率化を図ったものである。一方、以下では提案する方式を (b) をベースにして説明する。

(飛行中) メッセージは生きているので、ゴミ検出を停止するには、ネットワーク中のメッセージすべてが黒になる (浄化されている) 必要がある。そのためには、すでに飛行中のメッセージについては到着するのを待つこととするが、新しいメッセージについては最終的に黒として送出されるようにしていく必要がある。これを実現するには (b1) メッセージを送出するところで黒くする方法と、(b2) オブジェクトの mutator(のルート) を黒くしていく方法がある。(b1) はメッセージ送信時に新しいメッセージを黒として生成し、その初期化にコピーバリアを用いていると考えることができ、並列オブジェクト指向言語 POOL-T での GC[1] で用いられている。一方、(b2) は mutator が黒であるオブジェクトから送信されたメッセージはそのまま黒にできることを利用して、メッセージの初期化にコピーバリアを用いずに間接的にメッセージを黒にする。(b2) では黒にされた mutator が浄化されている状態を保つ必要があるが、これにはすでに説明したコピーバリアを用いればよい。(新たに受けとるメッセージが黒でなければメッセージから参照されるオブジェクトをマークして受けとる)。提案する方式はこの (b2) の方式を用いる。

mutator と、受動的オブジェクトの関係を含めて整理すると、必要なコピーバリアは、(B1) 黒の mutator がメッセージを取り込む際にメッセージが黒でなければメッセージから参照される先をマークする「メッセージ取り込みバリア」、(B2) 黒の mutator が組になっている受動的オブジェクトへの参照を回復するとき (その並行オブジェクトがアクティブになったことを意味する) に受動的オブジェクトが黒以外なら黒にする「アクティブバリア」、となる。(さらに新しいオブジェクト生成に関してその初期化のところにもバリアが必要である。新しいオブジェクトの mutator は黒にしておく)。

本来のコピーバリアの考えだけからいうと、(B2) は、(B2') 並行オブジェクトがアクティブになったときに受動的オブジェクトが白なら灰にする「コピーバリア」、を用いれば十分なのであるが、その後の受動的オブジェクトから mutator のルートへの参照の読み込みに対するコピーバリア (リードバリア) が必要となり、リードバリアのオーバーヘッドの大きいため (B2) のようにした。(B2) により、個々のオブジェクト内部の GC については、並行 GC ではなく一括 GC を行っていることになる。

リードバリアを使わずに一括処理をすることにより、提案方式は、(a) の消失バリアベースにおいてライトバリアを使わずに一括処理をしていると考えることもできる (メッセージが並列オブジェクトに取り込まれる際にはコピーと消失がセットで行われるから、メッセージ取り込みバリアに関しても (a) とも (b) とも考えられる)。この場合、各並列オブジェクト毎に mutator が黒くされた時点をローカルな GC 開始時刻とし、ローカルな GC 開始時刻の集合をグローバルな GC 開始時点とみなせばよい。ただし、提案方式は、実際の (物理的な)GC 開始時点にすべてのオブジェクトの mutator を黒に塗ることなく、通常計算と並行して黒に塗っていく方式であるため、(b) のコピーバリアの考えに基づいているといえる。

5 実装

この節では、実際に EM-4 に実装した、より具体的なアルゴリズムを示す。

マスター collector と、各要素プロセッサ (PE) に配置した collector が、通常計算と並行して行う GC アルゴリズムは次のようになる。マスター collector は PE の一つに配置した。各 PE に配置した collector は、同じ PE 内のオブジェクトやメッセージは自分でマーク・浄化するが、異なる PE 上のオブジェクトやメッセージはその PE の collector にマークメッセージを出してマーク・浄化を依頼する。この時、ACK の返答先も送る (実は、マーク先と ACK の返答先は同時に EM-4 の一つのパケットに収まらないのでそれぞれ別のパケットで送っている。この際、デッドロックを防ぐにためには ACK の返答先を先に送る)。また、各 collector は、ACK の返答先を一つだけ保存し、PE 内に灰色のオブジェクトがなくなり、他の collector からの ACK もすべて戻ってきていれば、その一つだけ保存していた返答先に ACK を返す。既に ACK の返答先を保存しているときに ACK の返答先を受けとった場合はそのまま ACK を返す。マスター collector が、各 collector に最初に ACK 返答先を保存させておくことで、マスター collector にすべての ACK が帰ってきた時点で、系全体に灰色のオブジェクトが存在しないことが検出できる。

GC 処理の大まかな流れは以下のようになる。

1. mutator はオブジェクトの生成ができない場合、

すでに GC が開始されていなければ、マスター collector へ GC 開始の依頼。

2. マスター collector から各 PE の collector へ GC 開始の依頼 (6. 用の ACK 返答先付きで)。
3. 各 PE の collector は、mutator と並行動作しながら、その PE の中の並列オブジェクトを一つ一つアクティブかチェックし、(1) アクティブであればその並列オブジェクトを浄化する。(2) アクティブでなければ、そのオブジェクトのハンドラを置き換えることでアクティブバリアを有効にする。各 PE の collector は、すべてのチェックが終わればマスター collector へ通知。
4. マスター collector はすべての collector からのアクティブチェック終了の通知を受けとったら、ブルドージングパケット [6] をオメガ網に沿って流す。アクティブチェック終了以後のメッセージは黒として送信され、また、ネットワークは FIFO 性を保証しているため、ブルドージング終了後は黒のメッセージしかないことが保証される。
5. マスター collector から各 PE の collector へ黒のメッセージしかないことを通知。この通知を受けとるまでは、各 PE の collector は、最後の ACK を返せないものとする。
6. 各 collector で、マーク・浄化処理が続けられるが、マスター collector がすべての collector から ACK を受けとると終了。
7. マスター collector から、各 collector へスイープの依頼。
8. 各 collector は、白いオブジェクトを回収する。回収後、マスター collector へ通知。
9. マスター collector はすべての collector からのスイープ終了の通知を受けとったら、各 collector へ GC 終了の通知。

アクティブバリアはメッセージハンドラ (へのポインタ) を置き換えるだけで着脱可能であり、ノード collector は、そのノードでサスペンドしている mutator に、(他の mutator と並行動作しながら) アクティブバリアを順に付けていくことができる。

また、mutator と collector は同じ PE 内で並行に動作するが、排他的にスケジュールされる。mutator は、自分がスケジュールされていない時には、collector がスケジュールされた場合に備えて、オブジェクト内の浄化すべき変数の地図を保存しておく。

mutator が collector に協力するためのバリアは、アクティブバリアだけでなく、黒のオブジェクトが黒でないメッセージを受けとった場合にメッセージに含まれる参照の先をマークするメッセージ取り込みバリアもある。このため、メッセージ送信時に送信元の mutator は、メッセージ中の参照の位置がわかるようにしておくとともに、メッセージに mutator の色を引き継がせる。送信元の mutator の色を引き継がないで、ブルドージング完了以前に届いたすべてのメッセージに関して、メッセージから参照される先をマークする方法も考えられる。この方法は、色の情報が必要ない分、通常計算にかかるオーバーヘッドは小さくなるが、GC 時のマークの数は増える。

	relative overhead			
	create		**cycle**	
N	**o1**	**o2**	**o1**	**o2**
500	1.16	1.20	1.06	1.08
1000	1.18	1.21	1.09	1.13
1500	1.18	1.22	1.12	1.16
2000	1.18	1.22	1.14	1.17
2500	1.18	1.22	1.12	1.16
3000	1.17	1.22	1.13	1.17

(N: 質点数)

表 1: GC 組み込みの相対オーバーヘッド

実際には、さらに、パイプライン送信 [4] へ対応のために mutator に、0 フィルさせたりする処理や、GC 中のオブジェクト生成に対応するための処理がある。後者で注意しなくてはならないことの一つには、GC 中に生成されたオブジェクトは、メッセージ取り込みバリアが有効になる前にもメッセージをキューに貯めていることがあり、これらのメッセージも浄化しなくてはならないということがある。

6 実験と評価

前節で提案した GC 方式を、並列オブジェクト指向言語処理系 ABCL/EM-4 に組み込む形で実装し、実際に GC 処理を含むプログラム実行を行い、動作を確認した。

EM-4 の 80 個の要素プロセッサは、それぞれ 0.5MB のヒープを持つ。その約 2/3 を 0.5KB 単位のフリーリストとして並列オブジェクトに用いる。

アプリケーションプログラムには N体問題の木構造を用いたアルゴリズム [2, 4] を用いた。[4] では、並列オブジェクトをノードとする木構造を生成するのに、木構造の足し合わせという手法を用いており、この際、足し合わせの中間結果にあたる並列オブジェクトがゴミとなる。また、木構造の親と子のノードは最終的に双方向の (循環した) 参照を持つ。木構造の生成後は力を計算するフェーズに移る。木構造生成フェーズと力の計算フェーズを合わせて一サイクルとして一タイムステップ先の N個の質点の位置と速度を計算する。各サイクルの木構造生成フェーズでは前回生成した木構造に沿って足し合わせを行う。このため、実際のプログラムでは、さらに初期木構造を生成するフェーズがあり、その後各サイクルを実行する。

今回はこの初期木構造生成フェーズ (**create**) と、通常のサイクル一回分 (**cycle**) を評価の対象とし、

create	N	N_{gc}	T_{off}	T_{on}	T'_{gc} (T'_{gc}/N_{gc})	T_{gc} (T_{gc}/N_{gc})	T'_{gc}/T_{gc}	T_m	T_m/T_{gc}
o1	500	0	73.52	74.51	0.99 (–)	0.00 (–)	–	0.00	–
	1000	0	202.18	201.14	-1.05 (–)	0.00 (–)	–	0.00	–
	1500	1	304.54	356.27	51.73 (51.73)	84.72 (84.72)	0.61	83.55	0.986
	2000	2	403.99	538.84	134.85 (67.43)	237.51 (118.75)	0.57	234.98	0.989
	2500	4	516.72	794.51	277.79 (69.45)	538.65 (134.66)	0.52	533.30	0.990
	3000	7	624.13	1051.68	427.54 (61.08)	787.97 (112.57)	0.54	778.48	0.988
o2	500	0	76.12	76.82	0.70 (–)	0.00 (–)	–	0.00	–
	1000	0	207.40	211.74	4.35 (–)	0.00 (–)	–	0.00	–
	1500	1	316.33	348.43	32.10 (32.10)	68.13 (68.13)	0.47	66.97	0.983
	2000	2	419.58	506.28	86.70 (43.35)	190.89 (95.44)	0.45	188.32	0.987
	2500	4	533.60	704.99	171.38 (42.85)	436.40 (109.10)	0.39	431.04	0.988
	3000	7	650.87	921.28	270.41 (38.63)	673.63 (96.23)	0.40	664.20	0.986
cycle	N	N_{gc}	T_{off}	T_{on}	T'_{gc} (T'_{gc}/N_{gc})	T_{gc} (T_{gc}/N_{gc})	T'_{gc}/T_{gc}	T_m	T_m/T_{gc}
o1	500	0	1230.89	1232.09	1.19 (–)	0.00 (–)	–	0.00	–
	1000	2	3458.93	3505.37	46.45 (23.22)	63.17 (31.59)	0.74	60.96	0.965
	1500	2	5075.89	5147.66	71.77 (35.88)	110.56 (55.28)	0.65	108.29	0.979
	2000	2	7063.27	7145.65	82.38 (41.19)	102.42 (51.21)	0.80	100.18	0.978
	2500	2	9761.81	9847.15	85.34 (42.67)	118.22 (59.11)	0.72	116.02	0.981
	3000	2	12766.83	12874.28	107.45 (53.72)	116.75 (58.38)	0.92	114.38	0.980
o2	500	0	1252.93	1252.45	-0.48 (–)	0.00 (–)	–	0.00	–
	1000	2	3579.20	3613.98	34.78 (17.39)	54.40 (27.20)	0.64	52.22	0.960
	1500	2	5255.01	5301.57	46.55 (23.28)	85.30 (42.65)	0.55	83.02	0.973
	2000	2	7251.72	7300.76	49.05 (24.52)	87.18 (43.59)	0.56	84.95	0.974
	2500	2	10095.87	10157.25	61.37 (30.69)	102.34 (51.17)	0.60	100.16	0.979
	3000	2	13195.85	13111.97	-83.87 (-41.94)	98.06 (49.03)	-0.86	95.69	0.976

(時間の単位はミリ秒; N: 質点数, N_{gc}: GC 起動回数, T_{off}: GC なしの実行時間, T_{on}: GC ありの実行時間, T'_{gc}: $T_{on} - T_{off}$, T_{gc}: GC 時間, T_m: マーク時間)

表 2: GC 実行時間の比較

		N_{obj}	$N_{live}(N_{live}/N_{obj})$	$N_{new}(N_{new}/N_{live})$	$N_{blk}(N_{blk}/N_{live})$	$N_m(N_m/N_{live})$
create	**o1**	48461.6	30615.1 (0.63)	3713.8 (0.12)	1008.9 (0.033)	63749.8 (2.08)
	o2	48543.4	30442.8 (0.63)	3885.9 (0.13)	1029.2 (0.034)	42898.9 (1.41)
cycle	**o1**	47068.9	13568.3 (0.29)	2883.4 (0.21)	368.7 (0.027)	48901.2 (3.60)
	o2	46807.7	13662.7 (0.29)	3110.8 (0.23)	368.9 (0.027)	39359.0 (2.88)

(N_{obj}: GC の対象とされたオブジェクトの数, N_{live}: GC を生き残ったオブジェクトの数, N_{new}: GC 中に生成されたオブジェクトの数, N_{blk}: GC 中に実行をブロックされたオブジェクトの数, N_m: GC 中にオブジェクトに対して行なわれたマーク数)

表 3: GC のプロファイル

質点数 (N) は、500, 1000, 1500, 2000, 2500, 3000 と変化させた。**create** は、**cycle** より頻繁にオブジェクトを生成し、両フェーズの性質は全く異なるため、二つのアプリケーションで評価していると考えて良い。また、メッセージに色を付けずにブルドージング完了前のメッセージに関して常にマークを行う方式 (**o1**) と、メッセージに色を付けて必要な時だけマークを行う方式 (**o2**) について比較した。

まず実装した GC 方式が GC を起動していない通常計算時にどれほどのオーバーヘッドを課すのか調べるために、(1)GC のためのバリアや色付けためのコードを mutator のコードに埋め込んだプログラムと、(2)GC のためのコードを含まないプログラムの、実行時間の比を測定し表 1に示した。(1)、(2) とも free を陽に埋め込んで GC が起動したりメモリが枯渇したりしないようにしてある。

表 1より、メッセージに色を付ない方式 (**o1**) は初期木構造生成フェーズ (**create**) で 16〜18%程度のオーバーヘッド、通常のサイクル (**cycle**) で 6〜14%程度のオーバーヘッドと、新しいオブジェクトの生成が多い実行部分の方がオーバーヘッドが大きいことがわかる。またメッセージに色を付ける方式 (**o2**) では、**create** で 20〜22%程度のオーバーヘッド、**cycle** で 8〜17%程度のオーバーヘッドとなっており、メッセージに色を付ける分、オーバーヘッドが大きい。

次に、提案した GC 方式が、GC を起動したときに必要な処理の量を調べるために、先の (1) のコードから free を除き普通に GC が起こるようにしたプログラム、と (1) の GC なしのプログラムの実行時間などを比較する。表 2には、普通に GC が起こるようにした場合の GC 起動回数 (N_{gc}) と GC なしの実行時間 (T_{off})、GC ありの実行時間 (T_{on}) とその差 (T'_{gc}) を示した。T'_{gc}は、GC が起動されるようにした時の (表の括弧内は GC 一回あたりの) 総実行時間の増加で、GC の処理に費やされているものと考えられる。一回の GC あたりの実行時間の増加は (非決定性の影響の大きい N=3000 のケースを除くと) **o1** で 23〜69ms 程度、**o2** で 17〜43ms 程度

	提案方式の選択	他の選択候補
基本アルゴリズム	検出・回収方式	リファレンスカウント方式
ルートの処理	並行	一括
バリア方式	コピーバリア	消失バリア
バリア対象の単位	メッセージと並列オブジェクト	ノード, またはスロット
メッセージの処理	取り込み時	送信時

表 4: 提案する GC 方式の特徴

であることがわかる。**o1** のほうがより多くの処理を必要とする。また、T_{gc}は実際に GC が起動していた時間を表す。この時間は T'_{gc}より長いが、これは GC が通常の計算と並行して実行されているためで、T'_{gc}/T_{gc}に示したようにこの間 39~80%程度の処理が GC に費やされていると考えられる。また、T_mは T_{gc}のうち、マークフェーズが実行されている時間を表しており、T_m/T_{gc}に示すようにほとんどの時間はマークに費やされていることがわかる。

メッセージに色を付ない方式 (**o1**) とメッセージに色を付ける方式 (**o2**) の実行時間を比較すると、**create** については GC が起動される $N \geq 1500$ で **o2** のほうが総実行時間 (T_{on}) が短い。一方、**cycle** についてはもともとのオーバーヘッドの小さい **o1** のほうが総実行時間が短い。**o1** の方が GC の処理が増える理由は、メッセージに色を付けずおいて、メッセージを受け取ったオブジェクトが必要以上にマークするためと考えられる。そこでマークの数などのプロファイルをとった結果を表 3にまとめた。表 3は、表 2であげた **create** と **cycle** のそれぞれ 14 回、10 回の GC でのオブジェクトの数やマーク数などの合計から一回の GC 当たりの平均値を求めたものである。これによると **o1** のほうがマーク数に関してだけ多いことがわかる。また、表から今回評価に用いた N体問題のプログラムでは、GC を生き残るオブジェクトは **create** で平均 63%程度、**cycle** で平均 29%程度であること、GC 中にも GC を生き残ったオブジェクトの 12~23%程度の新たなオブジェクトの生成が行われていること、GC 完了までオブジェクトを生成できずに実行をブロックされたオブジェクトは GC を生き残ったオブジェクトの 3%程度と少ないこと、などがわかる。なお、空きメモリ量のばらつきについては、25% 以上の空きがあるケースは 1%未満であり、約半数のケースで空きは 12%以下であった。

7 おわりに

本論文では、データ駆動並列計算機上の並列オブジェクト指向プログラミング言語処理系実装において開発した、並列オブジェクトのガーベジコレクション (GC) 方式について述べた。提案した GC はマーク&スイープに基づき循環したゴミも取り、通常のプログラムの実行を長時間ブロックせず、また、通常のプログラム実行に大きなオーバーヘッドを課さない、という特徴を持つ。

これらの特徴を満たすため、提案した GC 方式では、並列オブジェクトの内部に対してオーバーヘッドの大きいリードバリア・ライトバリアを用いずに、個々の並列オブジェクトを単位として一括処理を行う。また、通常計算時に完全に取り外せるアクティブバリアを提案し、これとメッセージ取り込みバリアを組み合わせて用いるものとした。その他、提案方式の特徴を表 4 にまとめる。

また、電総研で開発された非共有メモリ型のデータ駆動並列計算機 EM-4 上での実装を行い、GC の動作を確認した。メッセージに色を付ける場合とメッセージに色を付けない場合とを比較すると、メッセージに色を付ける場合は、GC そのものの処理量は小さくなるが、通常計算におけるオーバーヘッドがあり、比較的オブジェクトの生成が少ない (GC が起こりにくい) フェーズが長く続く場合は、メッセージに色を付けないほうがプログラム全体の実行時間は短縮されることが確認された。

今後は、今回評価しなかったメッセージ送信時のバリアを用いた方式との性能比較を行っていきたい。

参考文献

[1] L. Augusteijn. Garbage collection in a distributed environment. In *Proc. of PARLE*, Vol. 259, pp. 75–93. Springer-Verlag, June 1987.

[2] J. Barnes and P. Hut. A hierarchical $O(N \log N)$ force-calculation algorithm. *Nature*, 324:446–449, 1986.

[3] S. Sakai, Y. Yamaguchi, K. Hiraki, Y. Kodama, and T. Yuba. An architecture of a dataflow single chip processor. In *Proc. of the 16th Annual International Symposium on Computer Architecture*, pp. 46–53, June 1989.

[4] M. Yasugi. A concurrent object-oriented programming language system for highly parallel data-driven computers and its applications. Technical Report 94-7e, Department of Information Science, Faculty of Science, University of Tokyo, Apr. 1994. (Doctoral Thesis, Mar. 1994).

[5] A. Yonezawa ed. *ABCL: An Object-Oriented Concurrent System.* The MIT Press, 1990.

[6] 鎌田, 松岡, 米澤. 超並列計算機上の高効率な大域的ガーベジコレクション. 並列処理シンポジウム JSPP'94, pp. 33–40, May 1994.

[7] 八杉, 松岡, 米澤. ABCL/onEM-4: データ駆動計算機上の並列オブジェクト指向計算システムの高性能実装. 並列処理シンポジウム (JSPP'92) 論文集, pp. 171–178, June 1992.

[8] 八杉. 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. 第 12 回オブジェクト指向計算ワークショップ (WOOC'96), Mar. 1996.

[9] 松井, 田中, 前田, 中西. 相補型ガーベジコレクタ. 情報処理学会論文誌, 36(8):1874–1884, Aug. 1995.

[10] 湯浅. 実時間ごみ集め. 情報処理, 35(11):1006–1013, Nov. 1994.