TM-1240

分散GC

市吉　伸行（MRI）

December, 1992

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03)3456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

分散ＧＣ

市吉 伸行 *

1 はじめに

分散ＧＣという言葉は聞き慣れないが、ここでは、分散記号処理系におけるガーベジ・コレクション（ＧＣ）のを指すものとする。

ここ数年、大規模並列マシンが次々と世に出てきている。そのようなマシンは、プロセッサ数が増えてもメモリアクセスがボトルネックにならないようにメモリへのバンド幅がスケールするような分散メモリ型アーキテクチャを持っている。そのようなマシン上に記号処理システムを実装する試みも今後増えて行くであろう。分散メモリ並列マシン上の処理系（分散処理系と呼ぼう）では、処理系の並列性とともに分散処理であること故の問題が色々と生じる。ガーベジコレクションもその一つである。

第五世代プロジェクトでは、中期（1985 ～ 1988）に分散メモリ並列マシンであるマルチPSIを開発し、その上に並列論理型言語KL1処理系を設計・実装した。この処理系では、プロセッサ間即時ＧＣが実装され、また（実装はされなかったが）大域的な一括ＧＣ方式も検討された。筆者は処理系設計・開発に携わった者の一人として、ＧＣの検討内容を思い出しつつ、分散ＧＣの方法について述べよう。

2 分散メモリ型並列マシン

分散メモリ型並列マシンのアーキテクチャには、

- プロセッサ群とメモリ群がネットワークで結合されたもの。
- 局所メモリを持つプロセッサがネットワークで結合されたもの。

の２種類があり得るが、筆者の知る限り、全ての分散メモリ型並列マシンは後者の構成をしている。

これは、アクセスの局所性を生かして、頻繁にアクセスするデータを局所メモリに置くことによって全てのメモリアクセスをネットワーク経由にするよりも性能を高められるからである。[1]

初期には BBN Butterfly マシンのように、局所メモリでないメモリは全て一様に遠いというアーキテクチャがあったが、マシンが大規模化するにつれて、局所データと遠いデータという２種類だけでなく、局所データ、近いデータ、より遠いデータというような配置の工夫ができるようなネットワークで結合するアーキテクチャが主流になってきている。

多くの分散メモリ並列マシンでは、非局所データへのアクセスは明示的なメッセージ送受信によって行なう。このようなマシンのうち、MIMD 型のものは、プロセッサと局所メモリからなる各ノードが独立したコンピュータのようなものと考えるので、**マルチコンピュータ** [1] と呼ぶ。nCUBE 社 nCUBE シリーズ、Intel 社 Paragon、（MIMD マシンとしての）Thinking Machines 社 CM-5、富士通 AP-1000、Parsytec 社 GC などがその例である。

一方、メモリコントローラが非局所アドレスを検出して、ハードウェアで自動的またはＯＳカーネルへの割出しによって、ユーザプログラムに透過的にネットワークへ読出し／書込みパケットを送り出す仮想共有メモリ方式もある。BBN 社 Butterfly、スタンフォード大学の DASH、MIT の Alewife、Kendall Square Research 社 KSR-1 などがその例である。[2]

仮想共有メモリ方式分散メモリ並列マシンは、まだ例が少なく有効性が確立していないので、ここでは、マルチコンピュータをターゲットマシンとして仮定することにする。[3]

3 一括ＧＣ

逐次処理系におけるＧＣと同様、分散処理系のＧＣにも**一括型**と**即時型**があり得る。

一括型では、割付け可能なメモリ領域が無くなった時点で通常処理を中断し、全ての生きているデータを同定し、残った領域を回収してデータ割付け可能な自由領域とする。即時型では、通常処理の中でデータがゴミになった時にその場で回収する。

また、一種類のＧＣ機構を用意せずに、プロセッサ内に閉じた局所的なＧＣとプロセッサ間のＧＣとを組

* （株）三菱総合研究所 応用技術部

[1] 並列ランダムアクセス機械 PRAM は前者のアーキテクチャの分散メモリマシンと見なすこともできる。ただし、どのプロセッサからどのメモリセルへのアクセスも１ステップと仮定しているため、メモリアクセスの局所性はアルゴリズムの性能に影響しない。

[2] なお、Butterfly では readonly でない非局所データは原則としてキャッシュできなかったが、最近の仮想共有メモリ並列マシンでは、書き込み可能なデータもキャッシュできるような一貫性キャッシュ方式を実現しており、アクセスパターンに応じて動的に局所性が高まるようになっている。

[3] マルチ PSI/PIM がマルチコンピュータであったため、仮想共有メモリ方式の分散ＧＣを筆者が考えたことがない、というのが、本当の理由。

み合わせるという方式もあり得る。

本節で一括ＧＣの方法について述べ、4節で即時ＧＣ、5節で局所ＧＣについて説明しよう。

3.1 一括ＧＣの流れ

一括ＧＣの全体の流れは以下のようになろう。まず、あるタイミングで一括ＧＣを起動する。各プロセッサにおいて、ルートから始めて参照ポインタをたどって生きているデータの全てをマークする。各プロセッサはマーキング中に外部参照にぶつかったら、参照先プロセッサにマークメッセージを送信する。マークメッセージを受信したプロセッサは、その参照先からマーキングを続ける。マーキングが終了したら、プロセッサ毎にヒープ領域やフリーリストの再構成を行ない、それが終了した時点で、通常処理を再開する。

以下、順にそれぞれのステップの注意点などを述べる。

3.2 起動

一括ＧＣを始めるタイミングは、処理系によって都合の良いように決めれば良い。一つのプロセッサでデータ割付けができなくなった時点で一括ＧＣを起動することが考えられるが、一括分散ＧＣは比較的長時間かかると考えられるので、まず、１つのプロセッサ内で局所的にＧＣを行ない、それでも十分なメモリ量が残らなかった時にはじめて全体のＧＣを起動する方がいいかも知れない（5節参照）。

一括ＧＣ起動の要求は複数のプロセッサから同時に出るかも知れないので、正しく同期を取る必要がある。一括ＧＣ全体を指揮する**マスタプロセッサ**を一つ固定しておいて、そこで処理を逐次化するのが簡明でいいであろう。その場合、一括ＧＣの起動を要求するプロセッサは**一括ＧＣ起動要求メッセージ**によってその旨をマスタプロセッサに伝える。マスタプロセッサは、起動要求メッセージを受信すると直ちに**一括ＧＣ起動メッセージ**の指示を全プロセッサにブロードキャストする。複数の一括ＧＣ起動要求メッセージを前後して受信した場合は、２番目以降のメッセージを単に無視する。

一括ＧＣ起動メッセージを受信したプロセッサは、マスタプロセッサにACKメッセージを返すとともに、通常処理を中断して待ち状態に入る。待ち状態の間に到着する**通常処理メッセージ**（プロセッサ間データ読出し／書込み、プロセス移動などのためのメッセージ）は、処理せずにバッファ領域にコピーしておく。

マスタプロセッサは、全プロセッサからブロードキャストメッセージへのＡＣＫメッセージを受信した時点で、通常処理が中断したことが分かるので、**マーキングフェーズ開始メッセージ**をブロードキャストする。

ここで問題となるのは、ネットワークに残っているかも知れない通常処理メッセージである。ＧＣ起動前に送信されたが、まだ宛先プロセッサに到着していないメッセージがあるかも知れないのである。ネットワーク内にメッセージが存在しないことをハードウェアの機能として検出できればそれでよいが、そうでない場合は、ソフト的に確認する必要がある。メッセージのネットワーク内滞在時間に予め知られた上限があればその時間だけ待てばよいが、ネットワークは輻輳する可能性があり、一定時間内のメッセージ到着は一般に保証されない。

メッセージ消滅をソフト的に保証する一つのやり方は、何らかの分散終了検出アルゴリズムによって通常処理中断メッセージ到着以前の通常処理が完全に中断したことを検出する方法である[4]。ただし、分散終了検出アルゴリズムは、メッセージカウントの管理など何らかのオーバヘッドを通常処理に負わせることになる。

ネットワークの特性によっては、通常処理に変更を加えずに、以下のようにメッセージ消滅が保証できることがある。ネットワークが有限である限り、メッセージが通過するチャネルの数にはメッセージに依らない上限 d（ネットワークの直径）がある。もし、ネットワークのチャネル上でメッセージの追い越しがなく、しかも、全てのチャネルに一斉に掃き出し用の特別なメッセージを流すことができれば、それを d 回繰り返した後には、掃き出しを開始した時点でネットワークに残っていた全てのメッセージが宛先に到着していることが保証される。多くのネットワークは上記の性質を持っており、この**メッセージ掃き出し法**が有効であろう。

3.3 マーキング

プロセッサ内マーキングは、基本的に逐次ＧＣのマーキングの方法を用いれば良い。マーキングルートは、スケジューリング・キューにつながれたプロセス（あるいは関数呼び出しフレーム）および未処理メッセージ中のポインタである。また、他のプロセッサから送られてくるマークメッセージの参照先もプロセッサ内マーキングの出発点となる。各プロセッサはマーキング中に外部参照にぶつかったら、参照先プロセッサに**マークメッセージ**を送信する（図1。メッセージは %メッセージ名の形式で示す）。

マーキング方式として、コピーイングＧＣを用いた場合、マーキング中に生きているデータが新空間に移動するので、移動後のアドレスを参照元に正しく反映させる必要がある。そのためには、マークメッセージに参照元のアドレスを載せるようにし（図2(a),(b)）、マーク先が移動した際に、逆向きの**アドレス更新メッセージ**によって新しいアドレスを参照元に報告するよ

[4]終了検出（termination detection）というより、休止検出（quiescence detection）というべきか。

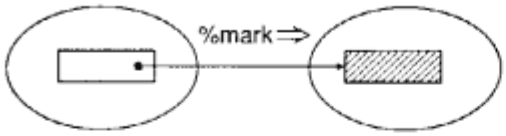

図 1: マーキングメッセージ

うにすればよい（図 2(b)）。参照元プロセッサでは新しいアドレスを待って中断する必要はなく、非同期に到着するアドレス更新メッセージに従ってアドレスを更新する（図 2(c)）。[5]

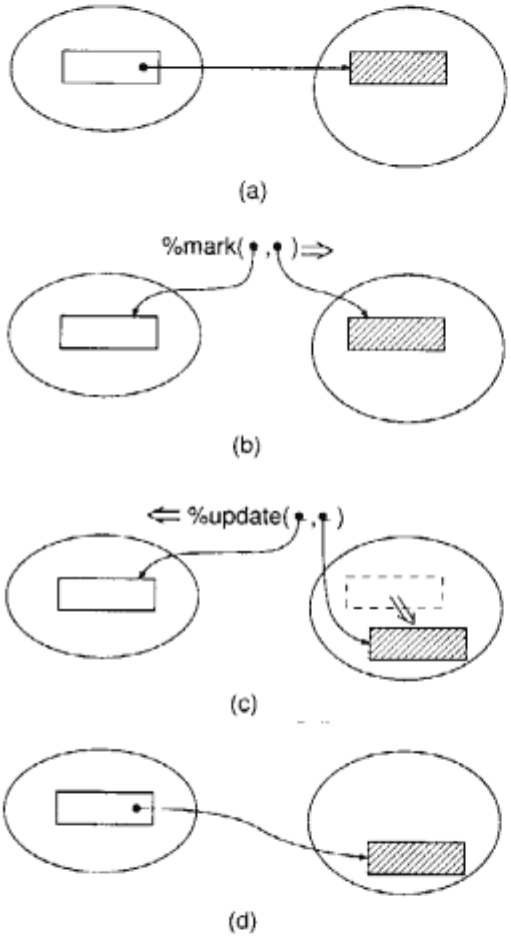

図 2: アドレス更新

各プロセッサは、マーキング・フェーズ中に、マークメッセージを取り込まなくてはいけないが、差し当たってマークすべきデータがなくなった時点で取り込むか、または、マーキング中に時々メッセージ受信をポーリングするか、の選択肢がある。前者では、メッセージ受信バッファが一杯になってもメッセージが取り込まれず、送り手プロセッサが待たされる可能性がある（あるいは、送り手がメッセージ送信を中断しなくてはならない）。一方後者では、取り込んだマークメッセージをソフトウェア・バッファにコピーするか、マーキングルートに付け加えるか、などしなくてはならず、そのためのオーバヘッドがあり、また余分のメモリ領域を確保しておく必要もある。どちらがよいかは、実験的に探るしかないであろう。

[5] コンパクションGCのコンパクションフェーズでのアドレス移動も同様だが、もう少し複雑であろう。

3.4 マーキングの終了検出

次のフェーズに移るためにマーキングの終了という大域的な状態を検出しなければならない。マスタプロセッサがイニシアチブをとって、何らかの分散終了検出アルゴリズムを用いることになろう。（終了検出アルゴリズムについては付録参照。）

3.5 メモリ管理情報の再初期化

マーキングフェーズ終了が確認されると、マスタプロセッサは**メモリ再初期化メッセージ**をブロードキャストする。それぞれのプロセッサで独立にメモリ管理情報を再初期化する（ヒープトップポインタの再設定、フリーリストの再初期化など）、それが終るとマスタプロセッサに**メモリ再初期化終了メッセージ**を送る。

3.6 通常処理の再開

全プロセッサからメモリ再初期化終了メッセージを受けとったマスタプロセッサは**通常処理再開メッセージ**をブロードキャストする。通常処理を再開したプロセッサは、他のプロセッサに通常処理用のメッセージを送信し、それが通常処理再開メッセージの到着前に着くかも知れない。したがって、もし通常処理再開メッセージより前に通常処理用のメッセージを受信したら、通常処理を再開し、後から到着する通常処理再開メッセージは無視するようにする。

4 即時GC

4.1 参照カウント方式

即時GCでは、生きているデータへの参照がなくなったことを通常処理中に検出する方法が要る。そのために、ポインタで指される可能性のあるデータに参照カウント・フィールドを設ける**参照カウント方式**が一般的である

参照カウント方式のカウントフィールドはせいぜい数ビットの固定長であり、参照数がカウントフィールドで表せる数の上限を越えると、無限大として扱われ、参照数が減ってもカウントフィールドは変えられない。そのために、参照数が大きく増えた後に再び減ってゴミとなるデータはゴミとなっても検出できないので、回収されないことになる。大半のデータへの

参照数はごく少数なので、実用上はあまり問題とならないようである。[6]

4.2 MRB 方式

ポインタ側にそれが単一参照であるか否かを示すフラグを付加する多重参照ビット (Multiple Reference Bit (MRB)) 方式 [4. 19] もある。MRB 方式では、データ生成時には MRB は落ちており (図 3(a))、ポインタをコピーする際に MRB を立てる (図 3(b))。ポインタを捨てる時に MRB が落ちていれば、唯一の参照が消えるから参照先はゴミとなることが分かる。

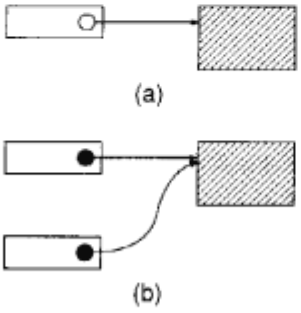

図 3: MRB

MRB 方式でも一旦立った MRB は決して落ちることがない (1 を越えたら無限大)。多くのプログラムでは参照数 2 以上のデータも一定の割合で存在し、それらが徐々にゴミとして溜ってゆく。したがって、MRB 方式だけでは不十分で、一括GC方式と組み合わせる必要がある。MRB 方式の意義は、ゴミの溜る速度を減らすことによって一括GCの頻度を下げることと、さらに重要なこととして、論理型言語のような変数が単一代入である言語においても、単一参照のデータの破壊的更新ができるようになることである。

4.3 分散処理系における参照カウント方式の問題点

プロセッサをまたがるポインタに参照カウント方式を用いるとすると、参照数の増減を伝える 参照増メッセージ、参照減メッセージを導入することになろう。その場合、次のようなレーシングの状況が起こり得る。

プロセッサ PE_i のデータ X へのポインタをプロセッサ PE_j が持っており、X の参照カウントは 1 であるとする (図 4(a))。プロセッサ PE_j が X へのポインタをコピーして、プロセッサ PE_k に渡そうとする。その際、プロセッサ PE_j はプロセッサ PE_i に参照増メッセージを送る (図 4(b))。X へのポインタがプロセッサ PE_k に到着する (図 4(c)) が、PE_k はそれを捨て、参照減メッセージをプロセッサ PE_i に送る (図 4(d))。何らかの原因で、参照増メッセージよりも前に 参照減メッセージがプロセッサ PE_i に到着し、データ X がゴミとして回収されてしまう (図 4(e))。後からプロセッサ PE_i に届いた 参照増メッセージの宛先は既に失われている。

このようなことが起こらない保証をするためには、

(1) 外部参照ポインタのプロセッサ外へのコピーを許さない、または

(2) 先に出した 参照増メッセージの方が参照減メッセージよりも先に到着するように保証する。

のいずれかの対策があろう。(1) は、言語仕様とするのは困難であろうから、外部参照ポインタのコピーの代わりに、PE_k から X への参照を PE_j 経由とする**間接ポインタ方式**が考えられる。間接ポインタ方式には、参照パスが長くなる問題や、PE_k からの参照がある限り PE_j における X への参照を回収できないという問題がある。(2) については、参照増メッセージと参照減メッセージはネットワークの異なる経路を通るのでハード的に到着順を保証することは難しい。確実なのは、参照増メッセージに対するＡＣＫメッセージを受信した後に、PE_k に外部参照ポインタを送ることだが、メッセージ送出の中断やACK メッセージ授受のオーバヘッドに問題がある。

そこで、この問題の一つの解決として、重み付き参照カウント方式が提案された。

4.4 重み付き参照カウント方式

通常の参照カウント方式では、参照ポインタをコピーするときに参照の増加を参照先に報告していたが、参照ポインタのコピーは参照の増加としてではなく、参照の分割と捉えることもできる。すなわち、参照ポインタにはある参照の重みが付随していて、参照のコピーの時に重みを分割するのである。例えば、初めに参照ポインタの重みが 100 だとすると、参照の分割の時に、50 ずつに重みを分ける。一方、参照を捨てる時には参照減メッセージによって参照の重みを返却する。これを重み付き参照カウント方式 (Weighted Reference Counting (WRC)) と呼ぶ [2, 13]。

通常の参照カウント方式では、

(データ X の参照カウント) =
(X への参照ポインタの数)

が成り立ったが、WRC 方式では、

(データ X の参照重み) =
(X への参照ポインタの参照重みの和)

を保つようにする (図 5)。これにより、被参照データ X に関して、X への参照ポインタが存在しない

[6] ただし、非常に長時間に渡ってプログラムを走らせると、ゴミが積もり積もることもあるだろう。

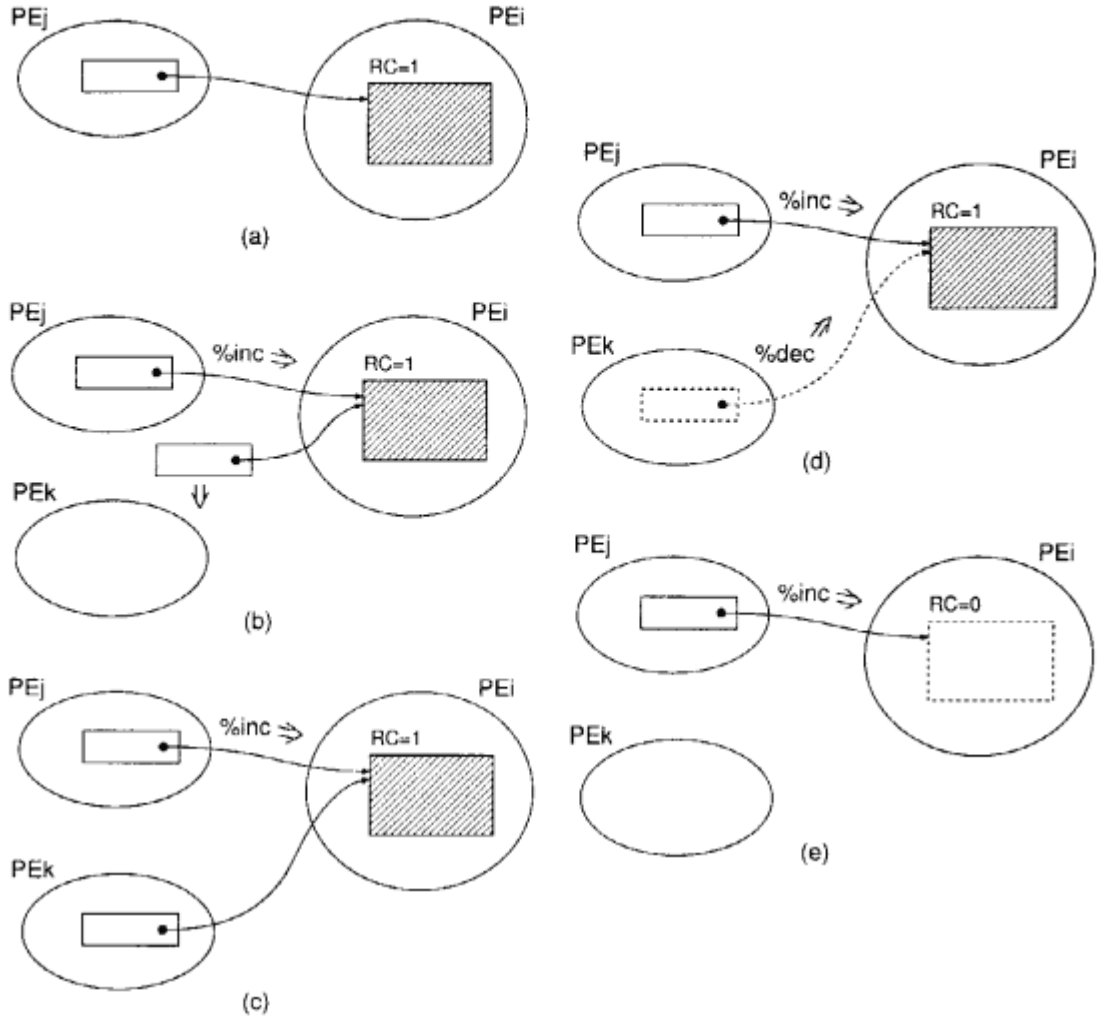

図 4: 分散参照カウント方式におけるレ--シング

ことと、X の WRC の値がゼロであること、は同値になる。通常の参照カウント方式は、WRC 方式においてポインタの WRC の値を 1 に制限したものと見なせる。WRC 方式が通常の参照カウントより優れている点は、参照ポインタを分割する際に被参照側のカウントを更新する必要がないことである。したがって、参照増メッセージも不要となる。

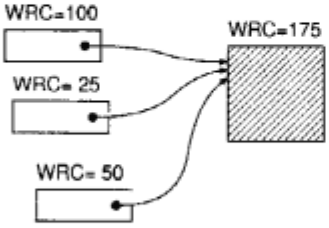

図 5: 重み付き参照カウント

WRC 方式の実装に当たっては、

(1) 重みの表現

(2) 初期の重み

(3) 参照の分割の方法

(4) 重み 1 の参照の分割の方法

を決めなくてはならない。参照の分割は 2 等分が自然であろう。この場合、例えば参照を 3 分割する時は、2 等分した後に、片方をさらに 2 等分することになって重みが偏るが、3 分割等の特殊処理を用意すると処理系が複雑化するであろう。また、重みを必ず 2 等分することにすると、重みの表現に関して次のような最適化ができる。すなわち、初期参照重みを 2 巾とすれば、それを 2 分割したものも 2 巾なので、底を 2 とする対数でもってコンパクトに表現できる。ただし、被参照側は初期重みから返却される重みを引いてゆくので、一般の非負整数を表現できなくてはならない。例えば、初期重みを 2^7 とすると、被参照側は重み（1 ～ 2^7）を 7 ビット、また参照側は重みの対数（0 ～ 7）を 3 ビットで表現できる。

重みが 1 となった参照の分割は、重みがそれ以上分

割できないので、何らかの特別な処理が必要となる。ひとつのやり方は間接ポインタとすることであり、別のやり方は参照先から新たに参照重みを貰うことである。後者には、参照分割の中断処理が必要になること、および、被参照側の重みに予め上限を設けられなくなる、などの問題点があり、単純な前者を採用する方が良いであろう。[7]

4.5 一括GCと即時GCの比較

一括GCと即時GCを比較すると、一括GCでは実行が長く中断してしまい、また生きているデータ全てをアクセスするので局所性が悪いという欠点がある。それに対し、即時型GCは、それによって通常処理が長く中断することがなく（非常に大きなデータ構造を一度に回収しようとしない限り）、通常処理で直前にアクセスしたデータを回収するために局所性に優れるという長所がある。これは、キャッシュやTLBのヒット率の高さやGCによるページングが少ないことやGCのためのメッセージ通信が少なくて済むということにつながる[8]。また、（重み付き）参照カウント情報／MRBをGC以外に有効に使うこともできる（7節参照）。

しかし他方、即時GCではプロセッサをまたがる循環構造は回収できず、また、参照数が参照カウントフィールドで表し切れない大きさになると、無限大として扱われ以後回収できなくなる。さらに、コンパクションせずにデータの割付け、回収、再割付けを繰り返すと、メモリ領域のフラグメンテーションが生じて、通常処理の局所性が徐々に減って行くという問題もある。加えて、通常処理中に参照カウントの手間が加わるために通常処理が遅くなり、また複雑度が増して、デバッグがより難しくなるという問題もある。

GCを設計する際には、このようなことを総合的に判断して、どのようなGCを単独であるいは組み合わせて用いるかを決める必要がある。特に、大きな応用プログラムを長時間走らせると、GC方式が総合性能に大きな影響を与える可能性がある。分散GCでは、GC自身の負荷バランスも重要な問題である。即時GCは通常処理に埋め込まれているので、通常処理の負荷バランスの一部として解決されるが、一括GCでは、プロセッサ間を何度も巡る長いリストが逐次ボトルネックになる可能性がある。基本的に、通常処理におけるデータや負荷が均等に分散されていれば、GCにおいても比較的良い負荷バランスが得られるのではないかと期待されるが、一括GCの所要時間は、GC起動のタイミングや走らせるアプリケーションに依存するであろう。

5 局所GC

分散処理系では、プロセッサ毎のメモリ消費速度のバラツキが避けられないが、GC機構として、前述のような大域的な一括GCのみしかないと、メモリ消費速度が最大のプロセッサの要求する頻度で、全系が中断してGCを行なうことになる。局所GCを実現すれば、全系の中断時間を最小（またはゼロ）にすることができると期待される。特に、メモリ消費速度が最大のプロセッサにおいて局所的に割付けてまたゴミとなるデータが多い場合に効果が大きいであろう。

局所GCでは基本的に、逐次GCの方式をそのまま用いることができるが、次の2点への配慮が必要である。

(1) 外部から参照されているデータをどのように同定するか

(2) 外部から参照されているデータのアドレスが局所GCによって移動した場合に、どのように参照元に伝えるか

(1)については、(a) 外部から参照されていることを示す外部参照フラグを導入したり、(b) 外部からの参照を一括管理する**外部参照管理テーブル**を設けることが考えられる。前者ではメモリ領域をスイープして外部参照フラグの立っているデータをマーキングにルートに加えることになり、また後者では、外部参照管理テーブルをマーキングルートとすることになる。

(2)については、外部プロセッサでは通常処理が進行中なので整合性を持ってアドレス更新を行なうことが非常に困難である。そこで、(a) 局所GCでは生きているデータのアドレスを変えないようにするか、または、(b) 外部からの参照を間接ないし連想テーブル経由とし、直接にアドレスを指せないようにすることが考えられる。外部参照管理テーブルはこの目的に用いることができる。マルチPSI上KL1処理系では外部参照管理テーブル方式を採用している。

（一括に限らず）局所GCによって、外部参照ポインタがゴミとして回収されることが当然あり得るが、プロセッサ間で即時GCを行なっている場合は、それを伝えることで参照先プロセッサでのゴミの回収につながる。

なお、外部参照管理テーブル方式のもとで局所的な一括GC（**一括局所GC**と呼ぶ）と大域的な一括GC（**一括大域GC**と呼ぶ）とを併用した場合は、一括大域GCの開始時点で新たな空の外部参照テーブルを割付け（新旧入れ替えでテーブルが2つあればよい）、マーキングメッセージに対応して新しいテーブルにアドレスを登録し直し、アドレス更新メッセージによって新しいエントリIDを知らせることになろう。

---

[7] 重み1の参照は参照カウント方式での通常の参照と類似している。参照重みの補給を要求するメッセージは参照増メッセージに対応する。

[8] 専用の %release メッセージの他に、通常処理メッセージに参照重みを載せることでも参照重みは返却できる。

6 事例： マルチPSI上KL1処理系におけるGC

分散GCの事例として、並列推論マシン実験機マルチPSI上の並列論理型言語KL1の処理系 [10]（以下単に、KL1処理系）におけるGCの方式を紹介する。KL1処理系では、以下の複数のGCを実装している。

- MRB方式の局所GC
- 一括局所GC（コピー方式）
- プロセッサ間即時GC（WEC方式）

前述のように、一括大域GCは方式が検討されたが、実装はされなかった。また、プロセッサ間MRB GCを行なわない理由は、一旦MRBが立ったポインタの参照先データの回収のために一括大域GCが必要となるからである。以下、KL1言語の概要とマルチPSIのアーキテクチャを紹介した後、外部参照方式とWEC方式について述べる。

6.1 KL1言語の概要

並列論理型言語KL1 [12]の概要を簡単に紹介する。KL1のプログラムはガード付きホーン節の集まりである。ガード付ホーン節は、Prologでお馴染みのホーン節を縦棒で区切った

$$H :- G_1, \ldots, G_m \mid B_1, \ldots, B_n.$$

という形をしている。ここで、"|"をコミットバー、その左側をガード部、右側をボディ部と言う。ヘッドHでは入力引数とのパターンマッチングを行ない、ガードゴールG_iたちは入力引数に関するテストを行なう。ボディ部は並列に実行されるゴール群である。ガード部では入力引数の観測のみが許されており、テストの際に対象となる引数が未定義だとそのテストは入力値が定まるまで中断する。ボディゴールには、アノテーションとして、それを実行すべきプロセッサや実行優先度を指定できる。

ガード付ホーン節は非常に単純な形をしているが記述力は高い。

```
filter([N|Ns],Ms) :- N mod 2 =:= 0 |
    filter(Ns,Ms).
filter([N|Ns],Ms) :- N mod 2 =\= 0 |
    Ms = [N|Ms1], filter(Ns,Ms1).
```

のように偶数を除去するフィルタを表現したり、

```
counter([increment|Ms],N) :- true |
    N1 := N + 1, counter(Ms,N1).
counter([read(X)|Ms],N) :- true |
    X = N, counter(Ms,N).
    ...
```

のようにカウンタ・プロセスを表現でき、また、

```
make_morning_set(X) :- true |
    get_hamburger(H)@node(Fridge),
    heat_hamburger(H,HH)@node(Oven),
    get_coffee(C)@node(CoffeePot),
    X = tray([HH,C,sugar,cream,spoon]).
```

のように並列実行も表現できる。@node(*Where*)はゴールを実行すべきプロセッシングノードの指定である。もう少し詳しくは、筆者による平易な解説 [15] などを参考のこと。

6.2 マルチPSI/PIMのアーキテクチャ

マルチPSIは第五世代プロジェクトの中期（1985～1988）に開発されたマルチコンピュータで、逐次推論マシンPSI-IIのプロセッサを要素プロセッサとし、それらを最大8×8のメッシュ型ネットワークで結合している。ネットワークは専用のルータによってワームホール・ルーチング（worm-hole routing）を実現しており、いわゆる、第2世代マルチコンピュータである。諸元を表1に示す。

より最近開発された並列推論マシンPIMにはハードウェアの詳細の異なる幾つかのモデルがあるが、そのうちのPIM/mはマルチPSIの高性能化版といえ、最大256プロセッサが2次元メッシュ・ネットワークで結合されている。PIM/m上のKL1処理系はマルチPSI上処理系をほぼそのまま移植したものである。他のPIMでは、プロセッシングノード自身が共有メモリマルチプロセッサ（クラスタ）となっているなどの違いはあるが、プロセッシングノード間の処理はマルチPSI上処理系とほぼ同じである。

6.3 外部参照の表現

5節で述べたように、KL1処理系では外部参照管理テーブル（**輸出表**と呼んでいる）を採用しており、外部参照はプロセッサID（8ビット）と輸出表のエントリID（24ビット）の対として表現している。

同じ外部データを参照する複数の外部参照ポインタを輸出表のそれぞれ異なるエントリで登録することが可能だが、そのようにすると間接テーブルのエントリ数は輸出するデータ数でなく、輸出回数分だけ必要となる。また、一つのデータに関する1対のプロセッサ間の外部参照パスを一つに限定しておくと、プロセッサ間でデータの読出しを重複して行なわずに済むメリットがある。このためにKL1処理系では、外部への参照を一括管理する**輸入表**を設け、同一の外部参照を輸入表の唯一のエントリに登録するようにしている。処理系データタイプとしての外部参照（EXREFセル）は、輸入表エントリへのポインタである（図6）。

表 1: マルチ PSI 諸元

タグ・アーキテクチャ	8 ビットタグ + 32 ビットデータ
要素プロセッサ制御	水平型マイクロプログラム方式 (53 ビット / 命令)
サイクルタイム	200 ns (プロセッサ、ネットワークとも)
主記憶	80 MB (16 MW) / プロセッサ
ネットワークチャネル	双方向 5 MB/s

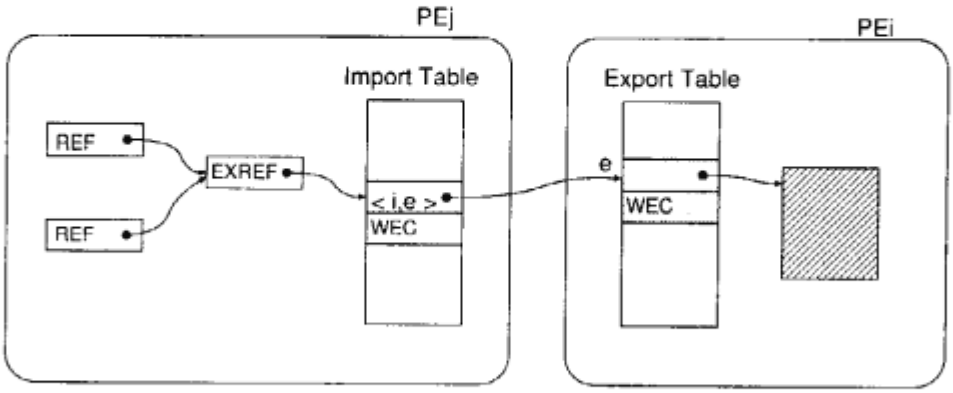

図 6: KL1 処理系における外部参照

図 6の状態において、PE_j から外部参照先を読みに行くと、データが PE_j にコピーされ、PE_j からの外部参照は消滅する。また、もし参照側の WEC と被参照側の WEC が等しいとすると (すなわち、PE_i のデータが PE_j のみから参照されていた)、PE_j からの外部参照の消滅に伴って PE_i 側のデータが回収される (図7)。[9]

6.4 WEC 方式

KL1 処理系では、重み付き参照カウントの一種である Weighted Export Counting (WEC) 方式をプロセッサ間即時GCに用いている [16, 5] [10]。4.4節で述べた WRC 方式との違いは以下の通りである。

- WEC 方式では、外部参照は直接的に外部データのアドレスを指さずに、輸出入表経由で指す。

  参照側の参照重みは EXREF セルでなく、輸入表側で管理する。

- 参照側の重みは分割されるだけでなく、足し込まれることがある。

  既に輸入されたものと同一の外部参照が輸入されると、参照重みが足し込まれる。(そのため、参照側で重みが分割されるだけでないので、対数表現はできない。)

KL1 処理系では、参照重みフィールドとして、参照側で 32 ビット、被参照側で 64 ビットを割当てており、被参照側は決してオーバフローしないと仮定している [11]。輸出時の重みはパラメータで変えられるが、デフォルトでは 2^{24} を与えている。したがって、連続 24 回の参照分割が可能である。参照重みが 1 になった場合は、間接参照としている。

7 その他の話題

7.1 参照カウント／MRB 情報の利用

論理型言語のように変数値の書き換えのできない (単一代入) 言語では、例えば配列の 1 要素を更新するにも、一般には、新たな配列を割付けなくてはならない (上書きするとその配列を参照している変数の値が変わってしまうことになる)。ところが、参照カウントや MRB 情報によって、参照先の配列へのポインタが他にないことが分かると、配列要素の上書きができる。

KL1 処理系では、MRB が OFF の場合の最適化と

[9] 現在の処理系では、メッセージが 1 往復半 (%read、%anser_value、%release) してはじめて PE_i のデータが解放される。最初の %read メッセージ到着時点で解放した方が効率が良いが、後から PE_j から書き込みメッセージが送られる可能性 (通常のプログラムはそのような振舞いをしないが) があるのでそのようにできないのである。静的解析によって書き込みのないことが分かれば、%read メッセージ到着時点の解放ができるであろう。

[10] WRC と同時期に独立に考案された。

[11] オーバフローするためには、最低 2^{32} 個の参照ポインタが必要だが、輸入表機構により 1 プロセッサには 1 つの参照しかなく、ネットワークおよびバッファ中のメッセージからの参照数も、PIM/m 256 プロセッサで最悪でも高々 2^{18} 個程度。

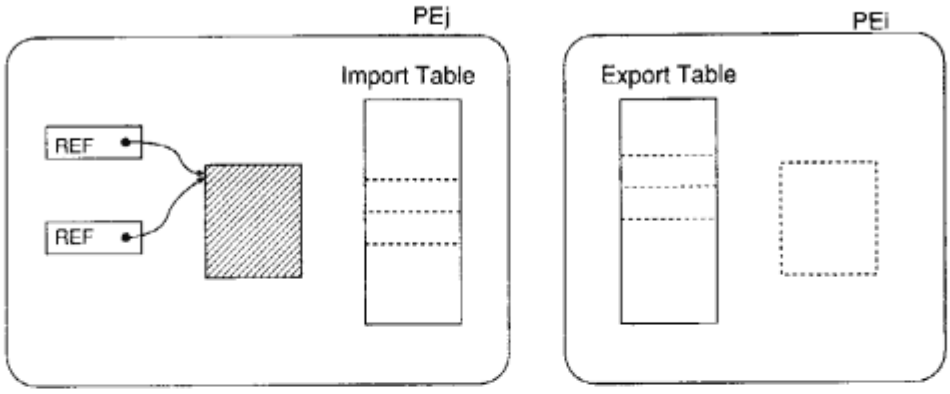

図 7: 外部参照の読出し後の状態

してその他にも、ストリームマージの最適化なども行っている [4, 6]。

7.2 GCによる永久中断の検出

KL1 言語においては、具体化していない変数の値を読もうとしたゴールは中断するが、その際処理系では、変数具体化時にゴールを再開するために変数からゴールへポインタを張るようにしている。

中断しているゴールはGCにおけるマーキングルートとならないが、中断の原因となっている変数からのポインタ経由でマークできる。ところが、その変数に値を与えるべきゴールが変数への参照を捨ててしまうと、中断ゴールは永久に再開されない。そのようなゴールは、GCにおいてマークされずゴミとなるので処理系が検出することができる [7]。

KL1 処理系では一括局所GC時でゴミとなったゴール（のうちのデータフロー関係で極大のもの）をユーザに報告しているが、永久中断ゴールが循環構造をなしており（デッドロック）、それがプロセッサをまたがっている場合、検出できない。一括大域GCを実装すると、全ての永久中断を検出できるようになる。

7.3 世代別GC

前述のように KL1 処理系では、一括大域GCに所要時間がかかるという前提で、一括局所GCとプロセッサ間即時GCの組合せを取っている。しかし、そのために外部参照におけるインディレクション段数の増加やまた輸出入表操作などのオーバヘッドを招いている。そこで、大域GCのオーバヘッドを減らすために、一旦割付けられた後に長時間ゴミとならないデータをマークせずに済むような世代別GCが提案されている [17, 9, 18]。ただし、世代別GCでは、旧世代領域からカレント世代領域へのポインタを管理することによる通常処理の複雑化やオーバヘッドがあり、また、カレント世代領域が尽きた時のための（普通の）一括大域GCを別に実装しなければならない問題がある。本格的な並列処理系における実装評価はこれからのようである。

7.4 共有メモリマルチプロセッサにおけるGC

共有メモリマルチプロセッサにおける並列GCに関しては [14] などの研究がある（その他の研究については [14] の参考文献を参照のこと）。

8 おわりに

分散GCは実現例が乏しいが、今後、本格的な分散処理系が開発されて行くにつれて増えて行くであろう。これまでに見てきたように、外部参照方式はGC方式を左右する。また、ハード的／ソフト的な非局所参照のオーバヘッドをどれくらいに抑えられるかも大きなファクタである。効率の良いGC方式実現のために外部参照方式を設計することもあろう。そうした全体 — できれば応用プログラムの性質も — を見通して処理系全体を設計できたら素晴らしいであろう。本稿がその時の検討の出発点になるとしたら幸いである。

最後にご注意、前述のようにマルチ PSI 上 KL1 処理系では一括大域GCは実装されなかったので、3節の記述にはバグがあるかも知れません！

参考文献

[1] W. C. Athas and C. L. Seitz. Multicomputers: Message-passing concurrent computers. *IEEE Computer*, 21(8):9–25, 1988.

[2] D. I. Bevan. Distributed garbage collection using reference counting. In *Proceedings of Parallel Architectures and Languages Europe*, pp. 176–187, June 1987. Also in *Parallel Computing*, Vol.9, No.2, pp.179–192, 1989.

[3] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[4] T. Chikayama and Y. Kimura. Multiple reference management in Flat GHC. In *Proceedings of the Fourth International Conference on Logic Programming*, pp. 276–293, 1987.

[5] N. Ichiyoshi, K. Rokusawa, K. Nakajima, and Y. Inamura. A new external reference management and distributed unification for KL1. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1988*, pp. 904–913, 1988.

[6] Y. Inamura, N. Ichiyoshi, K. Rokusawa, and K. Nakajima. Optimization techniques using the MRB and their evaluation on the Multi-PSI. In *Proceedings of NACLP'89*, pp. 907–921, 1989.

[7] Y. Inamura and S. Onishi. A detection algorithm of perpetual suspension in KL1. In *Proceedings of the Seventh International Conference on Logic Programming*, pp. 18–30, 1990.

[8] F. Mattern. Algorithms for distributed termination detection. *Distributed Computing*, 2(3):161–175, 1987.

[9] K. Nakajima. Piling gc — efficient garbage collection for ai languages —. In *Proceedings of IFIP WG 10.3 Working Conference on Parallel Processing*, pp. 201–204, 1988.

[10] K. Nakajima, Y. Inamura, N. Ichiyoshi, K. Rokusawa, and T. Chikayama. Distributed implementation of KL1 on the Multi-PSI/V2. In *Proceedings of the Sixth International Conference on Logic Programming*, pp. 436–451, 1989.

[11] K. Rokusawa, N. Ichiyoshi, T. Chikayama, and H. Nakashima. An efficient termination detection and abortion algorithm for distributed processing systems. In *Proceedings of the 1988 International Conference on Parallel Processing, Vol. I Architecture*, pp. 18–22, 1988.

[12] K. Ueda and T. Chikayama. Design of the kernel language for the parallel inference machine. *The Computer Journal*, 33(6):494–500, 1990.

[13] P. Watson and I. Watson. An efficient garbage collection scheme for parallel computer architectures. In *Proceedings of Parallel Architectures and Languages Europe*, pp. 432–443, June 1987.

[14] 今井明, エヴァン・ティック, 中島克人, 後藤厚宏. 共有メモリマルチプロセッサにおけるガーベジコレクションの 並列実行と評価. 情報処理学会論文誌, 33(3):298–306, 3 1992.

[15] 市吉伸行. 並列論理型言語 KL1. 数理科学, (313):11–17, 7 月 1989 年.

[16] 市吉伸行, 六沢 一昭, 近山, 中島克人, 宮崎敏彦, 杉野栄二. 並列処理における重みつき参照カウントを用いた実時間 GC. 第 36 回情処全大 7H-3, pp. 813–814, 1988.

[17] 小沢年弘, 細井聡, 服部彰. FGHC 処理システムのメモリ使用特性と世代別ガーベジコレクション. 情報処理学会論文誌, 30(9):1182–1188, 9 1989.

[18] 小池汎平, 田中英彦. 分散メモリ並列計算機上でのジェネレーションスキャベンジング GC. In *JSPP'90*, pp. 273–280, 1990.

[19] 木村康則, 近山隆. 並列論理型言語 KL1 の多重参照管理によるガーベジコレクション. 情報処理学会論文誌, 31(2):316–327, 2 1990.

付録：分散終了検出アルゴリズム

分散計算においては、メッセージ到着が新たな局所的な計算を起動するので（例えば、マーキングの終わったプロセッサにマーキングメッセージが到着して、新たなマーキングをトリガーすることがある）、全てのプロセスの局所的終了が計算の終了と同値でない。したがって分散計算の終了検出アルゴリズムは、全プロセッサの終了およびネットワーク内のメッセージの不在を検出しなければならない。

プロセッサ当たりのメモリ必要量が数ワード程度で、メッセージ当たりの処理オーバヘッドも小さいものとして**メッセージ数え上げ方式**がある。これは、ネットワーク内にメッセージが残っていないならば、送信メッセージ総数と全受信メッセージ総数が等しくなければならない、というアイデアに基づいている。そのために、各プロセッサにおいて、送信したメッセージ数および受信したメッセージ数を数えておき、マスタプロセッサから全プロセッサを訪れるトークンを繰り返し流して、送受信それぞのメッセージ総数を集める。基本的には、全プロセッサが局所的に計算を終了していて、かつ、送受信メッセージ数それぞれの和が一致していれば、分散計算は終了している筈だが、トークンが既に訪れたプロセッサからまだ訪れていないプロセッサへメッセージが到着する可能性のためにこれだけでは不十分である。しかし、上記の判定条件が成り立った後、もう一度トークンを流して送受信メッセージ数が変わっていなければ計算が終了していることが保証される。分散計算モデル、分散終了判定の詳細については、[3]、[8] などを参照のこと。

WRC の原理を応用した終了検出法も考案されており、KL1 処理系におけるタスク（「荘園」と呼んでいる）の終了検出に実装している [11]。