TR-649

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

今井　明、Evan Tick、中島　克人、
後藤　厚宏

May, 1991

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

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

今井　明
新世代コンピュータ技術開発機構 (ICOT)

Evan Tick
University of Oregon

中島　克人
三菱電機　情報電子研究所

後藤　厚宏
NTT　ソフトウェア研究所

概要

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(GC)の並列実行方式について提案する。本方式は Baker のコピー方式を並列に実行できるように拡張したもので, GC中の負荷分散に関しても考慮されている。本方式は並列論理型言語KL1システムであるVPIMに実装され,広範囲のベンチマークプログラムにより評価された。この結果, (1)GC中に共有される変数の更新頻度の抑止, (2)並列動作による高速化の達成, (3)プロセッサ間の負荷の均等化の実現,が確認された。

1　はじめに

動的に構造の割付を行なう記号処理言語を実行する並列システムにとって,割り付けた構造を動的に回収するガーベジコレクション(GC)の効率は重要な意味を持つ。特に, ICOTで開発を行なっている並列推論マシンPIM[4]のような並列論理型言語KL1[10]を実行するシステムでは, KL1が構造の破壊的上書きを許さないことや,バックトラックによるメモリ回収が不可能なことから,単純に実装すると1リダクション当たり5語程度のゴミを生成すると見積もっており, 8台の要素プロセッサ(PE)で構成するPIMのクラスタでは, 1秒間に10M語ものゴミを生成する。このようなシステムでは,特にGCの効率が全体性能を支配する。

このような急激なメモリ消費の問題を解決する方法として, MRB方式[2]という即時GC方式が提案されているが, MRB方式ではすべてのゴミを回収できず, 一括GCとの併用が前提となる。そこで, GC操作自体を効率良く実行することを目標に, コピー方式による一括GC処理を複数のPEで並列実行できるように拡張し, その評価を行なった。

2　Bakerの逐次アルゴリズム

本並列方式のベースとなった Baker の逐次アルゴリズム[1]について簡単に述べる。Baker のアルゴリズムでは, ヒープを2面用意し, 通常実行中はそのうち一方のみを使用する。一方のヒープ(旧領域)を使い切った時点で, その時にアクティブな(生きている)データオブジェクトのみをもう片方のヒープ(新領域)にコピーする。ゴミとなったオブジェクトにアクセスしないため高速であるという特徴を持つ。

図1に, このアルゴリズムを示す。 なお, 言葉の合意として, ヒープは上(top)から底(bottom)へ向けて伸ばして使ってゆくこととする。

このアルゴリズムは, 新領域の走査(スキャン)ポイントを指すS(scan)と,新領域の底を示すB(bottom)の2つのポインタを用いて実行される。まず, ルートポインタの指すオブジェクトを新領域にコピーしたあと, Sを底に向けて走査する。Sの指している先が旧領域へのポインタであった場合で, まだそのオブジェクトがコピーされていなければ, 新領域にコピーし, さらに旧領域には, コピー先新領域のアドレスを記録する。既にコピー先アドレスが記録されていた場合は, コピー先アドレスをSの指している先に書き込む。このような処理を続け, S = B になったところで GC は終了する。

3　並列実行のための拡張

本章では, 前章で述べたBakerのアルゴリズムを共有メモリモデルで並列に実行する方法を述べる。

3.1　並列性の確保

単純に並列実行を行なう一つの方法として, 複数のPEがS およびB のポインタを共有し, S の指すセルをスキャンし, B へコピーする方法がある。しかしこのような方法では, S および B への排他的アクセ

Parallel Garbage Collection on a Shared Memory Multi-Processor and its Evaluation
Akira Imai (ICOT), Evan Tick (Univ. of Oregon), Katsuto Nakajima (Mitsubishi Electric Co.) and Atsuhiro Goto (NTT)

スがネックとなり,並列実行による高速化が望めないことは明らかである。

この問題を解決する一つの方法として,静的にヒープを分割し,PE毎のローカルなヒープとして割り付ける方法が考えられる。こうして,SおよびBのポインタをローカルに管理すれば,前述の問題は回避でき,実際,このような方式はMultilisp[5]やJAM Parlog[3]のガーベジコレクタに採用されている。この方式でも,移動先アドレスを書き込む処理を排他的に行なえば,多重コピーなどの問題を生じることなく,並列にコピーを行なうことができる。しかし,このようなローカルな管理方法には,次のような重大な問題がある。

- 大きなオブジェクトをコピーしようとした時,空き領域の合計はそのオブジェクトの大きさよりも小さいのに,フラグメンテーションによりコピーする場所が確保できないことがある。
- あるPEが自分の領域を使い切った場合の処理が複雑である。
- あるPEがアイドルになった時に,他のPEから仕事を分配してもらうのが難しい。

これらの問題を解決するため,PE毎の新領域を静的に割り付けておく代わりに,必要に応じて動的に割り付けることにする。ただし,新領域の割り付けに際しては,次の二点を考慮する必要がある。

- 動的な割り付け処理の頻度を極力抑えること
- 割り付けが原因となるフラグメンテーションを発生させないこと

第一の点を解決するため,新領域をある一定の大きさで拡張してゆくことにする。この大きさを **ヒープ拡張ユニット**(**HEU**: Heap Extension Unit)と呼び,HEUを単位とした連続領域を**ページ**と呼ぶ。

最初に,簡単なモデルを考える。即ち,各PEは一組のSとBのポインタで管理されるページを持ち,SとBはページの先頭を指すように初期化する。B_{global} は,全PEで共有されるポインタで,新領域の底を指している。各PEは,必ずこの B_{global} を更新することにより,ローカルなページを確保する。

ローカルなページにコピーし尽くした時,即ちBが次のページ[1]の先頭に達した時の状況には2通りある。1つはSとBが同じページを指していた場合で,もう1つは,SとBが異なるページを指していた場合である。いずれの場合も,新しいページを割り付けてコピーを続ける。ただし,後で漏れなくスキャンを行なうために,後者の場合のみ,Bが指していた(未スキャン領域だけを保持した)ページを共有プールに入れる。この未スキャン領域を,アイドルになったPEが取り出しスキャンすることで負荷分散が行なわれる。Sがページをスキャンし尽くした時は,SはBの指すページの先頭にセットされる。

次に,第二点目のフラグメンテーションの問題の解決法を述べる。即ち,オブジェクトにはいろいろなサイズのものがあり,これらを無作為に固定サイズのページに詰めてゆくと,ページの底に使えない大きさの領域が残されてしまうという問題がある。この解決のため,簡単なモデルを拡張する。

通常実行時にオブジェクトを割り付ける際に,サイズを 2^n に丸める[2]。ただし,HEUを越えるオブジェクトは,HEUの整数倍($n\times$ HEU)[3]に丸める。HEUが2の冪乗で,オブジェクトをコピーする時「一つのページには同じ大きさのオブジェクトしか置かない」という規則を守れば,GCに起因するフラグメンテーションは生じない。

この拡張により,各PEはオブジェクトサイズの種類に応じた複数の $\{S,B\}$ 即ち,$\{S_1,B_1\}$, $\{S_2,B_2\}$, $\{S_4,B_4\}$, ..., $\{S_{\mathrm{HEU}},B_{\mathrm{HEU}}\}$ というポインタの組($log(\mathrm{HEU})+1$組)を管理することになる。GCに先だって,各サイズ毎にページを割り付け,$\{S_i,B_i\}$ に各ページの先頭を指させる。ただし,HEUより大きなサイズのオブジェクトに関しては,予め連続する

```
copy(P) {
    — P は旧領域へのポインタ —
    Temp := oldheap(P);
    if (Temp が新領域の移動先であれば)
        newheap[S] := Temp;
    else {
        newheap[S] := B;
        Arity := arity(Temp);
        — 旧領域に移動先を書き込む —
        oldheap[P] := B;
        for (i = 1; i ≤ Arity ; i++)
            — 旧領域の内容を、新領域にコピー —
            newheap[B++] := oldheap[P++];
    }
}

main() {
    S := B := 新領域の先端アドレス;
    for (i := 1; i ≤ # roots; i++ )
        copy( root(i) );
    while (S < B) do {
        P := newheap[S];
        if (P が旧領域へのポインタ)
            copy(P);
        S++;
    }
}
```

図 1: Bakerの逐次アルゴリズム

[1]他のPEが管理しているページかもしれない。

[2]2^{n-1} < サイズ ≤ 2^n : nは整数

[3](n-1) × HEU < サイズ ≤ n× HEU : nは整数

ページを割り付けておくことができないので，必要に応じて割り付け，$\{S_{\mathrm{HEU}}, B_{\mathrm{HEU}}\}$ の組で管理する。また，共有プールから取り出された領域の管理にも $\{S_{\mathrm{HEU}}, B_{\mathrm{HEU}}\}$ を用いる。

なお全てのPEで，$\forall k(S_k = B_k)$ を満たし，かつ共有プールが空の時点でGCは終了する。

3.2 負荷分散のための最適化

前節で述べた並列アルゴリズムでは，HEUの最適値を求めることが難しい。即ち，HEUを大きくすると，共有変数である B_{global} の更新頻度が抑えられる代わりに，SとBとの(ページを単位とする)距離が離れにくくなり，負荷分散の機会が減少してしまう。そこで，この相反する要求を満たすため，HEUとは独立の，負荷分散のための単位を導入する。この大きさを**負荷分散ユニット**(LDU: Load Distribution Unit)と呼び，そのLDUを単位とした連続領域を**サブページ**と呼ぶ。なお HEU mod LDU = 0とする。

サブページの導入に伴い，アルゴリズムを次のように変更する(図2参照)。

- Sがサブページを越えた時に，SとBに挟まれたサブページ群を共有プールに入れ，SをBが指すサブページの先頭にセットする

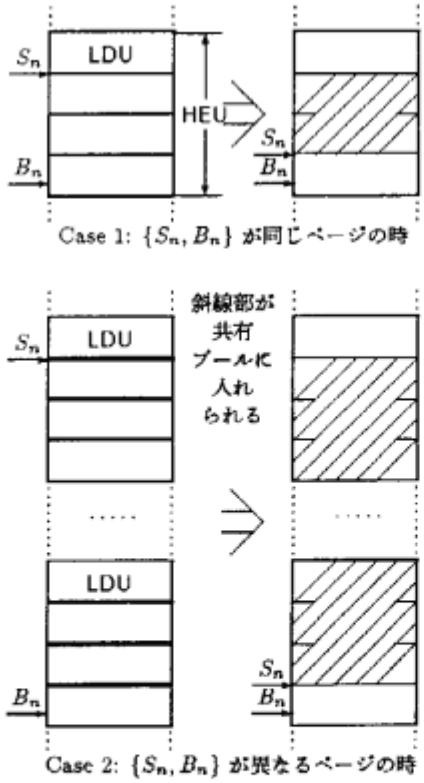

図 2: S が LDU 境界に達した時の負荷分散

3.3 並列実行アルゴリズム

本節では，前節までに述べた方式をまとめ，並列実行アルゴリズムを示す。

まず図3に，全てのサイズのページをスキャンするscan-all() と，サイズ n のオブジェクトが入ったページをスキャンするscan() のアルゴリズムを示す。スキャン処理がBakerのアルゴリズムと異なる理由は，S_n の指す先をスキャンした時に n と異なるサイズ(m)のオブジェクトをコピーし，B_m が更新されることがあるためである。

図 4は旧領域中のオブジェクトをコピーするcopy()のアルゴリズムである。ここでは，旧領域のオブジェクトをロックしておいて ($P()$)，コピー先アドレスを書き込み，ロックを解除し ($V()$)，オブジェク

```
scan-all() {
    repeat {
        — ある PE のページを全てスキャンする —
        repeat {
            スキャン中 := false;
            for (i := HEU; i ≥ 1 ; i := i/2)
                if (S_i < B_i) {
                    scan(i);
                    スキャン中 := true;
                }
        } until not(スキャン中);

        — 共有プールからのページ取り出しを試行する —
        if (get-from-pool(S_HEU, B_HEU)) {
            — ページ取り出しに成功 —
            — S_HEU と B_HEU をこのページ管理に使う —
        } else
            アイドル状態であると宣言する;
    } until (全てのPEがアイドルになる);
    — GC はここで終了する —
}

scan(n) {
    — サイズ n のオブジェクトのページをスキャンする —
    while (S_n < B_n) do {
        P := newheap[S_n];
        if (P が旧領域へのポインタならば) {
            — m は更新されたページのサイズ —
            m = copy(P, S_n);
            if( (m ≠ 0) ∧ (m ≠ HEU) )
                checkB( m );
        }
        S_n := S_n + 1 ;
        checkS( n );
    }
}
```

図 3: 全ヒープのスキャン

トをコピーする。ページ境界を越えた時の処理が後に必要となるため(更新されたページを特定するため),コピーしたオブジェクトのサイズを返す。

また,サイズがHEUを越えるオブジェクトのコピーする場合,そのオブジェクトのコピー先領域をB_{global}を更新して確保する。HEU以下のサイズのオブジェクトの場合,コピー先領域はGCの初期化の段階か図5のcheckB()で予め割り付けられている。

最後に,Sがページの境界を越えたかどうかの検査を行なうcheckB()と,Bがサブページの境界を越えたかどうかの検査を行なうcheckS()を図5に示す。いずれの場合も,境界を越えていない限り何も行なう必要はなく,越えていた場合には,必要ならば,未スキャン領域を共有プールへ入れたり,新しいページの割付を行なったりする。

```
copy(P,S_c) {
  — P は旧領域へのポインタ, S_c はスキャンポイント —
  P( oldheap[P] );
  Temp := oldheap[P];
  if (Temp が新領域の移動先アドレス) {
    V( oldheap[P] );
    newheap[S_c] := Temp;
    — どのサイズの B も更新されなかった —
    return(0);
  } else {
    Arity := arity(Temp);
    if ( Arity < HEU )
      m := Arity;
    else {
      m := HEU;
      — B_global を更新して,連続ページを獲得する —
      P (B_global);
      from := B_HEU := B_global;
      B_global := B_global + Arity;
      V (B_global);
    }
    newheap[S_c] := B_m;
    — 旧領域へ移動先アドレスを書き込む —
    oldheap[P] := B_m;
    V( oldheap[P] );
    for (i := 1; i ≤ Arity; i++)
      — 旧領域の内容を全て新領域にコピー —
      newheap[B_m++] := oldheap[P++];
    if ( Arity ≥ HEU ) {
      — 直ちに共有プールに入れる —
      add-to-pool(from, B_HEU);
    }
    — B_m が更新された —
    return(m);
  }
}
```

図 4: オブジェクトのコピー

4 評価

前章で提案した並列GCのアルゴリズムを,Sequent社のSymmetryで並列に稼働するKL1処理系VPIM[11]に実装し評価した。評価に用いたベンチマークプログラム([9]他)を表1に示す。ここで,ヒー

```
is-at-top-of-HEU(x) ≡ ((x mod HEU) = 0)
is-at-top-of-LDU(x) ≡ ((x mod LDU) = 0)
Top-of-HEU(x) ≡ (x div HEU) × HEU
Top-of-LDU(x) ≡ (x div LDU) × LDU
Bottom-of-HEU(x) ≡ (Top-of-HEU(x) + HEU − 1)
Top-of-prev-HEU(x) ≡ (Top-of-HEU(x) − HEU)
Bottom-of-prev-HEU(x) ≡ (Top-of-HEU(x) − 1)
Bottom-of-prev-LDU(x) ≡ (Top-of-LDU(x) − 1)

checkB(m) {
  — B_m がページ境界を越えたかをチェック —
  if (is-at-top-of-HEU(B_m)) {
    if (m ≠ HEU) {
      if (Top-of-HEU(S_m) ≠ Top-of-HEU(B_m-1))
        add-to-pool(Top-of-prev-HEU(B_m),
                    Bottom-of-prev-HEU(B_m));
      — B_global を更新して,1ページを獲得する —
      P(B_global) ;
      B_m := GlobalB;
      GlobalB := GlobalB + HEU;
      V(B_global) ;
    }
    — (m = HEU) ならば,単に
        S_HEU = B_HEU まで繰り返す —
  }
}

checkS(n) {
  — S_n がサブページ境界を越えたかのチェック —
  if (is-at-top-of-LDU(S_n)) {
    if (n ≤ LDU) {
      switch (距離(S_n,B_n)) {
        case (同じ HEU 内): {
          add-to-pool(S_n,Bottom-of-prev-LDU(B_n));
          S_n := Top-of-LDU(B_n);
        }
        case (異なる HEU): {
          add-to-pool(S_n,Bottom-of-HEU(S_n));
          add-to-pool(Top-of-HEU(B_n),
                      Bottom-of-prev-LDU(B_n));
          S_n := Top-of-LDU(B_n);
        }
      }
    } else if (LDU < n < HEU)
      if (is-at-top-of-HEU(S_n))
        S_n := Top-of-HEU(B_n);
    — else (HEU = n) 何もしない —
  }
}
```

図 5: 領域オーバフローのチェック

表 1: 評価に用いたベンチマーク (†は全解探索)

ベンチマーク	ヒープ (K語)	GC 回数	リダクション (K)	サスペンション (K)	平均仕事量 (K)	
BestPath	192	6	394	57	165	最短経路問題 (30×30 ノード)
Boyer	128	4	529	18	47	簡単な定理証明
Cube	128	5	291	6	139	制約充足問題 (7 キューブ) †
Life	128	4	353	236	101	ライフゲームのシミュレーション (38×38ノード)
MasterMind	128	8	1,525	5	4	ゲーム †
MaxFlow	128	3	80	35	95	最大流量問題 (80 ノード, 123 リンク)
Pascal	64	13	285	1	5	パスカルの三角形 (250 行)
Pentomino	64	7	188	9	3	二次元の詰込パズル (6 ピース) †
Puzzle	128	19	1,254	145	17	三次元の詰込パズル (7 ピース) †
SemiGroup	448	6	732	12	496	部分群問題 (5 タプル)
TP	64	23	564	47	17	定理証明
Turtles	320	1	1,178	62	203	制約充足問題 (カード12枚) †
Waltz	128	6	1,207	19	32	三次元の色塗り問題 (38 ノード) †
Zebra	320	9	405	2	167	制約充足問題 †

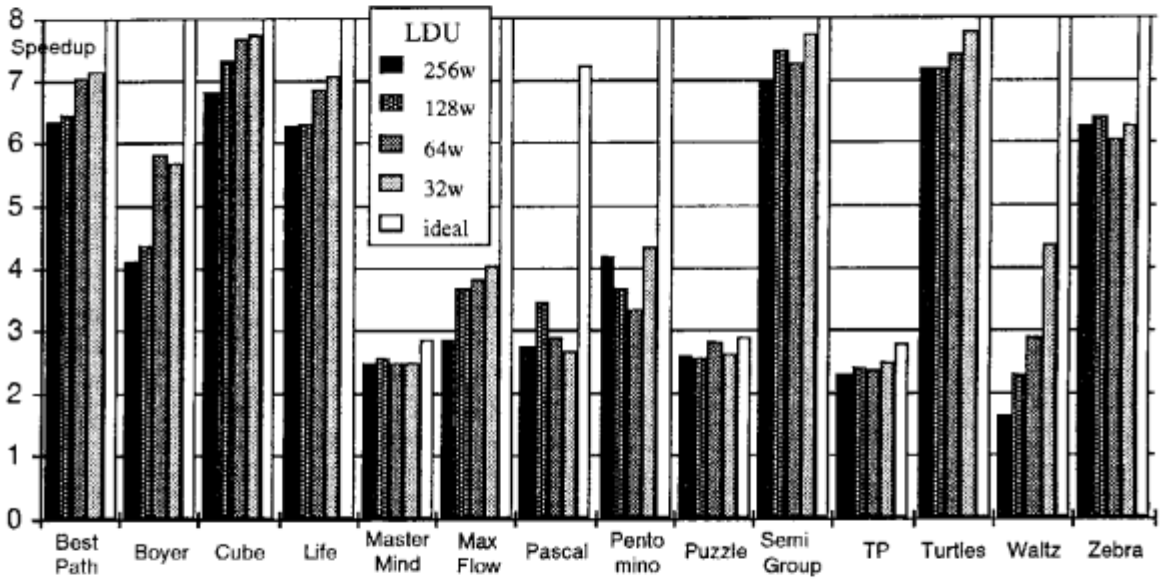

図 6: 速度向上と速度向上限界

ブサイズは, 静的に割り付けられた旧領域(= 新領域)の大きさである。

すべてのプログラムは一度だけ, 8台のPEで実行された。これ以降の表中に現れる「平均」は, 一度の実行中に起きた複数のGCの平均値である。また, 実際に並列に動作させたため, スケジューリングの差が, GC回数, リダクション数などに影響を与えているが, 表中の数値は, 特に断らない場合, PE=8台, HEU=256語, LDU=32語での測定値である。

4.1 負荷分散と速度向上

GC中の負荷分散状況を評価するための指標として, PE毎の仕事量と, システムの速度向上を次のように定義する。

$$\text{仕事量} \equiv \text{コピー語数} + \text{スキャン語数}$$
$$\text{速度向上} \equiv \sum \text{仕事量} \,/\, max(\text{仕事量})$$

仕事量はGC時間の近似値である。表1中の仕事量は, 全PEの仕事量の合計を1回のGC当たりの平均にしたものである。速度向上は実行時間を支配するのは最大の仕事量を持つPEであるという仮定に基づいて計算された値である。実際の実行時間を計測しなかった理由は, Symmetry のスケジューリングによる影響を排除するためである。なお, ここでの速度向上は, あくまでも負荷分散状況を評価するために導入した定義であり, 並列化のためのオーバヘッドは含まれていないことに注意が必要である。オーバヘッドは, 別に4.2節, 4.3節で見積る。

表 2: 全GCでの B_{global} の更新回数合計

ベンチマーク	LDU (語) 32 naive	32 smart	32 n/s	64 n/s	128 n/s	256 n/s
BestPath	124,569	2,305	54	60	56	57
Boyer	22,779	587	39	44	31	38
Cube	158,923	1,686	94	90	83	93
Life	68,687	1,326	52	52	52	52
MasterMind	3,427	522	7	6	6	7
MaxFlow	55,639	699	80	73	72	62
Pascal	14,437	917	16	15	16	16
Pentomino	9,480	305	31	32	29	30
Puzzle	38,831	1,486	26	29	27	27
SemiGroup	708,183	6,229	114	114	114	115
TP	40,455	1,738	23	23	23	22
Turtles	28,209	566	50	49	49	55
Waltz	39,476	715	55	62	56	60
Zebra	126,761	3,885	33	32	33	32

表 3: 共有プールの平均アクセス回数

ベンチマーク	LDU (語) 32	64	128	256
BestPath	421.0	139.6	84.4	45.8
Boyer	208.8	131.3	24.3	12.8
Cube	609.4	241.6	96.3	55.5
Life	145.8	66.5	29.8	14.8
MasterMind	3.9	1.5	1.1	1.0
MaxFlow	211.3	75.0	37.0	10.0
Pascal	1.6	1.0	1.0	1.0
Pentomino	134.3	65.3	21.0	7.5
Puzzle	51.6	30.6	10.5	4.9
SemiGroup	1,700.7	910.8	439.3	29.6
TP	44.4	19.8	8.8	4.6
Turtles	1,427.0	640.0	314.0	136.0
Waltz	76.0	36.0	11.5	1.4
Zebra	2,127.9	920.2	467.7	222.4

実際には, n台のPEを用いても n倍を達成することができないことが明らかな場合がある。即ち, 一つのオブジェクトのための仕事量が, 全体の仕事量の $1/n$ より大きい場合で, このような場合は, 最大のオブジェクトのための仕事量が全体のGC時間を支配することになる。これを考慮に入れた速度向上限界を次のように定義し, 完璧な負荷分散が達成できた時の速度向上の目安とする。

$$\text{速度向上限界} = min(\frac{\sum \text{仕事量}}{max(\text{1オブジェクトのための仕事量})}, \text{PE台数})$$

図6に, 速度向上および速度向上限界 (ideal) を示す。この図から, 仕事量が大きいプログラムほど速度向上の値が高くなるという実用的な結果が得られたことがわかる。即ち, 仕事量が100,000を越えるプログラムは, どのLDUでも6以上の速度向上を得ている。また, 予想通り大半のプログラムで, サブページの大きさ (LDU) を小さくすると速度向上値が向上している。

MasterMind,Puzzle,TPでは, 速度向上限界が2–3程度に抑えられているが, これは, 前述のように大きなオブジェクトが原因である。これらのプログラムで最も大きいオブジェクトはコードモジュールである。コードモジュールをGC対象にしたのは, VPIMをKL1のセルフシステムとするために, 動的にコードモジュールを割り付ける必要があるからである。実際には, KL1コンパイラが分割コンパイルを可能としているため, アプリケーションは多くのモジュールで構成されている。この現象は小規模プログラムに特有なものと考えられる。

PascalやWaltzでは, 測定された速度向上値が, 速度向上限界と比べて特に小さい。これは, これらのプログラムが長い平坦なリスト[4]を生成したためである。このようなリストの場合, SとBが同じ速度で進むため, SとBの間の領域をプールに入れることができなかったためと考えられる。

4.2 共有ポインタの更新削減

並列実行のオーバヘッドの見積りとして, 新しいページを獲得するために, 新領域の底を指している共有ポインタ B_{global} の更新頻度についても評価を行なった。このポインタの更新には, 排他制御が必要なため, 更新はできるだけ少ないことが望まれる。

Zebra (LDU = 32語) の場合, B_{global} は, 9回のGCの合計で3,885回更新された。もし, B_{global} が一つのオブジェクトをコピーする度に更新されていたとすると, 126,761回更新されていたことになり, 更新頻度を約 1/32 に抑えることができた。

これ以外のプログラムでの結果を表2に示す。B_{global} の更新は, 仕事量とHEUとオブジェクトの平均サイズに依存し, LDUの大きさには依存しないことがわかる。表中の "naive" は, オブジェクトをコピーする度に更新した場合, "smart" は, ページの導入により B_{global} の更新を抑えた場合で, n/s は, 両者の比である。MasterMindはこの比が特に小さく, 6–7程度に抑えられたが, これは仕事量が小さいためである。3.1節で述べたように, 初期化時点で log(HEU)個のページが割り付けられる。MasterMindの場合, この初期割付が, 8 (PE) × 8 (ページ) × 8 (GC回数) = 512回行なわれ, このための B_{global} の更新が全体の98%を占めたためである。この例を除くと, B_{global} の更新は 1/15 から 1/115 に抑えられている。

[4] Carがアトミックで, Cdrがリストへのポインタを保持しているようなリスト。

表 4: オブジェクトのサイズと種類別割合

ベンチマーク	サイズ		種類別割合 (%)					
	平均	σ^2	VR†1	LS†2	VT†3	GL†4	MD†5	MS†6
BestPath	4.13	569	11.3	6.4	15.9	44.6	14.3	7.5
Boyer	4.26	4,056	1.8	0.1	69.0	12.6	16.4	0.1
Cube	2.19	34	1.0	86.7	1.4	9.4	1.5	0.0
Life	2.94	87	14.5	32.9	0.3	50.1	2.1	0.1
MasterMind	6.50	3,488	1.3	21.5	6.2	16.8	52.3	1.9
MaxFlow	2.68	2,271	1.2	24.1	17.9	10.6	13.8	32.4
Pascal	2.86	1,326	0.8	65.3	0.8	7.6	23.8	1.7
Pentomino	5.78	4,639	3.3	12.0	17.3	22.2	33.4	11.8
Puzzle	5.60	16,786	0.9	23.2	14.6	8.9	52.0	0.4
SemiGroup	2.10	57	0.7	91.3	3.1	4.0	0.9	0.0
TP	6.60	56,583	0.8	22.3	15.7	6.9	53.9	0.4
Turtles	3.54	172	5.6	32.9	3.2	56.2	2.0	0.1
Waltz	2.56	366	1.1	72.6	1.5	11.6	12.7	0.5
Zebra	5.64	582	0.1	6.9	88.3	0.7	3.9	0.1

†1 **VaRiable**
1語, 未定義変数
†2 **LiSt**
2語, リスト
†3 **VecTor**
1-N 語, 配列
†4 **GoaL**
16,32語, ゴール環境(引数など)
†5 **MoDule**
1-N 語 (通常大), コードモジュール
†6 **MiSc**
1-N 語, 他の制御用レコード

4.3 共有プールのアクセス

表3に, 1回のGCあたりの共有プールの平均アクセス回数を示す。この表から, PascalとMasterMindを除き, LDUが小さくなるほど, 未スキャン領域の分配のためのアクセスが増えていることがわかる。これは, 4.1節での速度向上の裏付けとなっている。

更に, アクセス状況を細かく調べてみると, 例えばZebraの場合, 最も仕事量の多かったPEは, 平均で142回プールに入れ, 96回取り出していた。また, 最も仕事量の少なかったPEは, 平均で293回プールに入れ, 302回取り出していた。このような傾向は他のプログラムでも観察され, 負荷の高いPEの未スキャン領域を, 負荷の低いPEがスキャンすることで負荷の均等化が行なわれたことが確認された。

負荷分散のためのコストを見積もると, 仕事量当たりのプールアクセス頻度が最も高かったZebra (LDU=32語) の場合で, 平均78仕事量当たり1回の割合でアクセスしていた。1仕事量当たり約2回のメモリアクセスが, またプールのアクセスには2回のメモリアクセスが必要であるため, 共有プールのアクセス頻度は, 特に問題のない程度であると言える。

4.4 オブジェクトの種類による影響

GC時にアクティブであったオブジェクトの割合が, GC性能にどのように影響を与えるかについて調べた。表4に, GC時にアクティブであったオブジェクトのサイズと種類別割合を示す。

一般に, 高い速度向上を得たプログラムは, オブジェクトサイズの平均値が高く, また分散が低いという傾向がある。この観点から, プログラムを4つのグループに分類し, 表5に示した。表5中の括弧内の数値は, 図6に示した速度向上値の幅を示している。これらのグループの境界は 3.0 (平均) と 1000 (分散) である。種類別割合と合わせて考えると, ゴール環境とベクタは平均値を上げる方向にあり, これらは多くのポインタを含んでいるためSとBの距離が離れやすく, 負荷分散に良い影響を与えると考えられる。また, コードモジュールは特にサイズが大きいため分散を大きくする方向にあり, 一つのオブジェクトは単一PEでコピーされることから, 負荷分散に悪影響を与えることも考えられる。

表 5: オブジェクトサイズによる分類

		分散			
		低		高	
平均	低	Cube	(7.7–6.8)	Pascal	(3.5–2.7)
		Life	(7.2–6.3)	MaxFlow	(4.1–2.9)
		SemiGroup	(7.8–7.0)		
		Waltz	(4.4–1.6)	(悪いグループ)	
	高	BestPath	(7.2–6.4)	Boyer	(5.8–4.1)
		Turtles	(7.8–7.2)	MasterMind	(2.6–2.5)
		Zebra	(6.4–6.0)	Pentomino	(4.3–3.3)
				Puzzle	(2.8–2.6)
		(良いグループ)		TP	(2.5–2.3)

4.5 ページサイズと性能

最後に, ページサイズがGC性能に与える影響について評価した。表6にHEUと速度向上の関係を示す。この表からわかるように, 速度向上はLDUに影響され, HEUにはあまり影響されていない。これは, Bが移動する時にプールに入れる回数より, Sが移動する時にプールに入れる回数の方が多いからである。また, もしサブページの導入による最適化がなされなかった (HEU=LDU) とすると, 一番左の太字の数値が速度向上の値であり, サブページの導入が B_{global} の更新を抑えたまま, GC性能を向上させることを可能としたことが確認できる。

表 6: HEUの変化と速度向上

ベンチマーク	HEU (語)	LDU (語) 32	64	128	256	512
Boyer	128	5.93	5.73	**4.80**	—	—
	256	5.67	5.83	4.38	**4.12**	—
	512	5.82	5.81	5.88	4.88	**3.90**
MaxFlow	128	3.12	2.70	**4.23**	—	—
	256	4.06	3.84	3.70	**2.86**	—
	512	4.47	2.42	4.11	2.50	**2.18**
Pascal	128	3.31	2.78	**3.19**	—	—
	256	2.67	2.91	3.45	**2.77**	—
	512	3.02	2.93	2.82	2.63	**2.68**
SemiGroup	128	7.18	7.76	**7.46**	—	—
	256	7.75	7.28	7.49	**7.02**	—
	512	7.77	7.61	7.53	6.31	**6.88**
Zebra	128	6.07	6.66	**6.65**	—	—
	256	6.27	6.04	6.42	**6.28**	—
	512	6.44	6.18	6.44	6.49	**6.49**

5 おわりに

コピー方式をベースとした並列実行GC方式を提案し, KL1システム上に実装し, その評価を行なった。本方式は, KL1システムに限らず, 他の言語の実装にも広く適用可能である。本方式の特長は, 旧領域に移動先アドレスを書き込む操作, ページの獲得操作, 共有プールのアクセスを除く処理が排他制御なしで実現でき, 負荷分散も可能としている点である。本方式を, 8台のPEを用いた並列システムで多種のベンチマークを用いて評価し, 2.5倍 (MasterMind) から7.8(Cube) 倍の速度向上値を得た。速度向上限界を考慮に入れると, 限界値の51%(MaxFlow) から97% (Cube) の性能を達成することができた。また, 並列実行のためのコストについても見積り, 負荷分散のためのコストも, ページ獲得のためのコストも非常に小さいことを確認した。なお, 紙面の都合上アルゴリズムおよび評価結果に省略した点があり, 詳細は[6]を参照して頂きたい。

今後, この並列化技法を, オブジェクトのライフタイムに着目した「世代別GC」[7][8]に適用することも, 今後検討してゆきたい。

謝辞

本研究に関して, 有益な助言を頂いた平田圭二研究員を始めとする, ICOT第1研究室及び関係会社の方々に感謝する。また, 本研究の機会を頂いたICOTの渕一博研究所長, 淵和男第1研究室長に深く感謝する。また, Evan Tick の研究は米国科学財団 Presidential Young Investigator Awardの支援によって実現された。

参考文献

[1] H. G. Baker. List Processing in Real Time on a Serial Computer. *Communications of the ACM*, 21(4): pp 280–294, 1978.

[2] T. Chikayama, Y. Kimura. Multiple Reference Management in Flat GHC. In *Fourth International Conference on Logic Programming*, pp 276–293. MIT Press, 1987.

[3] J. A. Crammond. A Garbage Collection Algorithm for Shared Memory Parallel Processors. *International Journal of Parallel Programming*, 17(6): pp 497–522, 1988.

[4] A. Goto et al. Overview of the Parallel Inference Machine Architecture (PIM). In *International Conference on Fifth Generation Computer Systems*, pp 208–229, ICOT, 1988.

[5] R. H. Halstead Jr. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7(4): pp 501–538, 1985.

[6] A. Imai, E. Tick. Evaluation of Parallel Copying Garbage Collection on a Shared-Memory Multiprocessor. *ICOT Technical Report*, ICOT, 1991.

[7] K. Nakajima. Piling GC: Efficient Garbage Collection for AI Languages. In *IFIP Working Conference on Parallel Processing*, pp 201–204. North Holland, 1988.

[8] T. Ozawa, et al. Generation Type Garbage Collection for Parallel Logic Languages. In *North American Conference on Logic Programming*, pp 291–305. MIT Press, 1990.

[9] E. Tick. *Parallel Logic Programming*. Logic Programming. MIT Press, 1991.

[10] K. Ueda, T. Chikayama. Design of the Kernal Language for the Parallel Inference Machine. *The Computer Journal*, 33(6): pp 494–500, 1990.

[11] 山本 礼己 他 "並列推論マシン PIM における抽象機械語 KL1-B の実装 – 高級機械語を実装するための道具立 –", 信学技報 CPSY 89(168), 1989.