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†† 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，一括形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 の移動法を，フラグメンテーションの発生を抑えたまま，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．本方式は並行論理型言語 KL1 の並列実行システムである 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．特に並列推論マシン PIM[5] のような並行論理型言語 KL 1[12] を実行するシステムでは，KL 1 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などから，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いる．すなわち，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 10 M 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．

GC 方式については，LISP 言語の実装の研究が始まって以来，様々な方式が提案されてきた[3),7)]．また，KL 1 言語向きの GC 方式として MRB 方式[2)]という即時形の GC 方式が提案されている．しかし，MRB 方式ではゴミの回収が完全でないため，残りのゴミを回収する一括形の GC との併用が前提となる．また，移動法による一括形の GC (以下コピー方式と呼ぶ) を共有メモリマルチプロセッサ上で並列実行する方式[4),6),10)]も研究されているが，並列に実行するように拡張したことにより，フラグメンテーションを発生させるという問題が残されている．

本稿では，まずコピー方式による GC 処理を，フラグメンテーションの発生を抑えたうえで，複数の PE で並列実行する方式を提案する．本方式は，文献 4), 6), 10) と同様に，使用可能なメモリ領域がなくなった時点で通常実行を停止し，コピー方式の GC を複数の PE によって並列に実行する方式であり，通常実行と GC の実行を並列に行う並列形[7)]とは異なる．次に，この並列実行方式を，汎用の共有メモリマルチプロセッサ上に実装して評価した結果について報告する．

2. Baker の逐次方式

本並列方式のベースとなった Baker の逐次コピー方式[1)]について簡単に述べる．この方式では，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．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．

この方式は，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，B を進めてコピー先新領域を確保してからコピーするとともに，旧領域に

---

† Parallel Garbage Collection on a Shared Memory Multiprocessor and Its Evaluation by AKIRA IMAI (Institute for New Generation Computer Technology (ICOT)), EVAN TICK (University of Oregon), KATSUTO NAKAJIMA (Computer & Information Systems Laboratory, Mitsubishi Electric Corp.) and ATSUHIRO GOTO (Nippon Telegraph and Telephone Corp. Software Laboratories).

†† (財)新世代コンピュータ技術開発機構
††† オレゴン大学
‡ 三菱電機(株)情報電子研究所
‡‡ NTT(株) ソフトウェア研究所

は移動先の新領域のアドレスを記録する．既に移動先アドレスが記録されていた場合は，移動先アドレスを S の指す先に書き込む．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. GC の並列実行方式

前章で述べた Baker の逐次方式を，共有メモリマルチプロセッサで効率良く並列に動作させるにあたっては，特に以下の3点に注意が必要である．

- 排他制御の必要な共有ポインタの更新頻度を減らす．
- 並列にメモリ管理を行うことが原因となるフラグメンテーションの発生を避ける．
- GC 処理の負荷を均等化する．

本章では，これらの項目の解決方法を示しながら，並列実行方式を提案する．

3.1 共有ポインタの更新頻度削減

単純に並列実行を行う1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 の指すデータオブジェクトを走査し，B の指す新領域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自明である．

この問題を解決する1つの方法として，新領域を静的に等分割し，PE ごとのローカルな移動先として使用す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る．実際，この方式は Multilisp[6] や JAM Parlog[4] のガーベジコレクタに採用されている．また，佐藤らの方法[10]では，新領域を PE 数より多い数のブロックに分割しているが，コピーのためのポインタの管理の方法は，文献 4) や 6) と同様である．この方式でも，移動先アドレスを書き込む処理さえ排他的に行えば，多重コピーなどの問題を生じることなく，並列にコピーを行うことができる．しかし，この方式には，次に挙げる重大な問題がある．

- 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，新領域を静的に等分割する代わりに，必要に応じて動的に割り付ける．ただし，動的な割付処理には共有ポインタの更新が必要となり，この頻度を極力抑えるため，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．すなわち，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，すなわち B が次のページ* 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，後で漏れなく走査を行うために，後者の場合のみ，B が指していた（未走査領域だけを保持した）ページを共有プールに入れる (**図 1**)．この未走査領域を，アイドルになった PE が取り出し走査することで負荷分散が行われる．S がページを走査し尽くした時は，S は B の指すページの

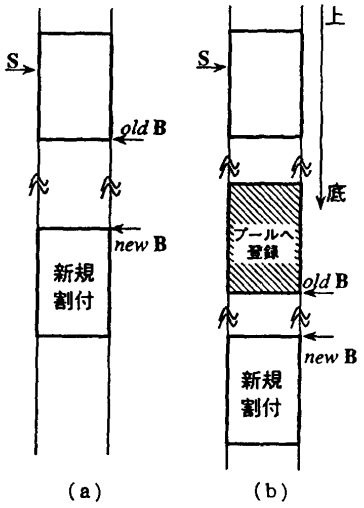

図 1 B の進め方
(a) S と B が同じページを指していた，(b) S と B が異なるページを指していた
Fig. 1 How to proceed B.
(a) S and B within the same page.
(b) S and B in different pages.

* 他の PE が管理しているページかもしれない．

先頭にセットされる.

次に，フラグメンテーションの問題の解決法を述べる．すなわち，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．この解決のため，前出のモデルを拡張する．

通常実行時にオブジェクトを割り付ける際に，そのサイズを 2^n に丸める*．ただし，HEU を越えるサイズのオブジェクトは，HEU の整数倍 $(n\times\mathrm{HEU})$** に丸める．HEU が2のべき乗で，オブジェクトをコピーする時「1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 $\{S, B\}$ すなわち，$\{S_1, B_1\}$, $\{S_2, B_2\}$, $\{S_4, B_4\}$, …, $\{S_{\mathrm{HEU}}, B_{\mathrm{HEU}}\}$ というポインタの組 (log (HEU)+1組) を管理することになる．GC に先だって，各サイズごと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，あらかじめ連続する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．

なおすべての PE のすべてのサイズ k で $(S_k=B_k)$ 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 GC 中の負荷分散のための最適化

前節で述べた並列実行方式では，HEU の最適値を求めることが難しい．すなわち，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 は LDU の整数倍とする．

サブページの導入に伴い，S の進め方を次のように変更する (**図2**参照)．

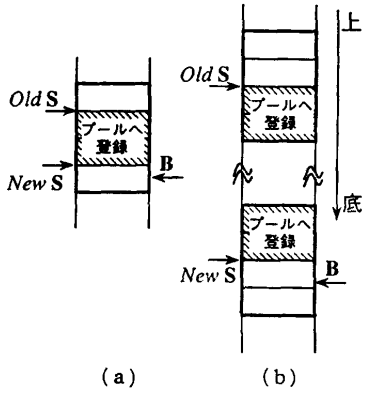

図2 S がサブページ境界に達したときの負荷分散
(a) S と B が同じページを指していた，
(b) S と B が異なるページを指していた
Fig. 2 Load distribution when S reaches sub-page boundary.
(a) S and B within the same page.
(b) S and B in different pages.

- 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

4. 評　価

前章で提案した GC の並列実行方式を，共有メモリマルチプロセッサである Sequent 社の Symmetry 上で並列に稼働する KL1 処理系 VPIM[13] 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 (文献11) ほか) を**表1**に示す．ここで，ヒープサイズは，旧領域 (＝新領域) の大きさ (語) 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 ごとの**仕事量**と，システムの**速度向上**を次のように定義する．

仕 事 量≡コピー語数＋走査語数

速度向上≡Σ仕事量／max(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

---

* $2^{n-1}<$ サイズ $\leq 2^n$: n は整数
** $(n-1)\times$HEU$<$ サイズ $\leq n\times$HEU : n は整数

表 1 評価に用いたベンチマーク († は全解探索)
Table 1 Summary of the benchmarks († All solution search).

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†

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，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．

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．すなわち，1 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，こ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．

速度向上限界

$$= \min\left(\frac{\Sigma \text{ 仕事量}}{\max\left(\begin{matrix}\text{1 オブジェクトの}\\ \text{ための仕事量}\end{matrix}\right)},\ \text{PE 台数}\right)$$

図 3 に，速度向上および速度向上限界を示す．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．すなわち，仕事量が 100,000 を越えるプログラム (図 3 中，枠で囲まれたもの) は，どの LDU でも 6 以上の速度向上を得ている．また，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．

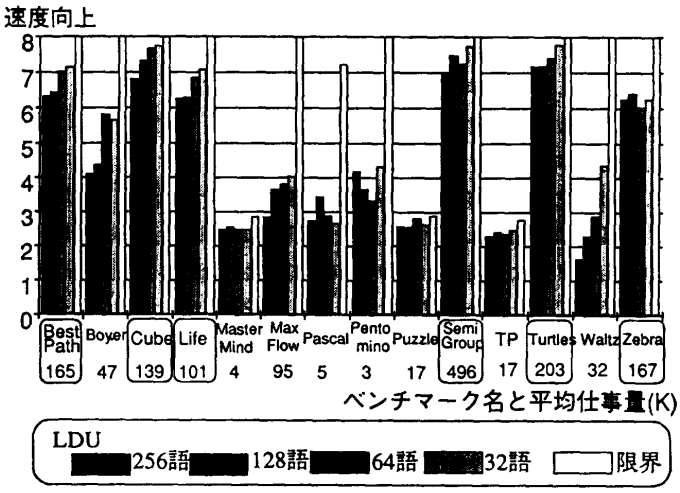

図 3 速度向上と速度向上限界
Fig. 3 Speedup and ideal speedup.

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ためである*.

* 例えば，VPIM 上で動く KL1 で書かれたコンパイラによって，ユーザプログラムが再コンパイルされた時には，同一モジュール名で異なるモジュールの実体が必要となる．これは，論理変数で指されるモジュールというオブジェクトも，KL1 のような純粋な言語では，破壊的な書き換えが禁止されているためである．なお，再コンパイルにより参照されなくなった古いモジュールは，本稿で述べる GC ではコピーされない．

実際には，KL 1 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サブ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* 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 (HEU＝256 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 1 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 1/33 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．なお，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．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 回行われ，このための B_{global} の更新が全体 (522 回) 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．

表 2 全 GC での B_{global} の更新回数合計
Table 2 Total number of updates of B_{global} over all GCs.

ベンチマーク	naive	smart	n/s
BestPath	124,569	2,305	54
Boyer	22,779	587	39
Cube	158,923	1,686	94
Life	68,687	1,326	52
MasterMind	3,427	522	7
MaxFlow	55,639	699	80
Pascal	14,437	917	16
Pentomino	9,480	305	31
Puzzle	38,831	1,486	26
SemiGroup	708,183	6,229	114
TP	40,455	1,738	23
Turtles	28,209	566	50
Waltz	39,476	715	55
Zebra	126,761	3,885	33

HEU＝256 (語)

* 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．

4.3 共有プールのアクセス

表 3 に，1 回の GC 当たりの共有プールの平均アクセス回数を示す．共有プールへのアクセスには排他制御が必要となるため，アクセス回数は少ないことが望まれる．この表から，Pascal と MasterMind を除き，LDU が小さくなるほど，未走査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．

さらに，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．このように，負荷の高い PE の未走査領域を，負荷の低い PE が走査することによる負荷の均等化は，他のプログラムでも観察された．

負荷分散のためのコストを見積もると，仕事量当た

表 3 共有プールの平均アクセス回数
Table 3 Average accesses of the global pool.

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

HEU＝256 (語)

表 4 オブジェクトのサイズと種類別割合
Table 4 Active objects size and distribution by type.

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

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．

表 5 オブジェクトサイズによる分類
Table 5 GC performance groups, categorized by object size.

		分散 低		分散 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)

ゴール環境 (**GL**) とベクタ (**VT**) 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．これらのオブジェクトの比率が高いプログラムでは，速度向上の値が高い結果が得られている．また，1 つのオブジェクトのコピーは単一の PE で行われるため，コードモジュール (**MD**) のような特に大きなオブジェクトの比率が高くなると，負荷分散が行われにくくなり，速度向上値が低いという結果も得られている．

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 3 に示した速度向上の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 1,000 (分散) である．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，また分散値が低いという傾向がある．種類別割合と合わせて考えると，ゴール環境およびベクタが平均値を上げ，コードモジュールが分散値を上げていることにより説明できる．

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

表 6 HEU の変化と速度向上
Table 6 Average speedup varying HEU.

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**

響され，HEU にはあまり影響されていない．これは，B を移動する時にプールに入れる回数より，S を移動する時にプールに入れる回数のほうが多いからである．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，一番右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．

5. おわりに

コピー方式に基づいた GC の並列実行方式を提案し，KL1 システム上に実装し，その評価を行った．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，(1)フラグメンテーションの発生を抑え，(2)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，(3)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 KL1 の並列実行システムで多種のベンチマークプログラムを用いて評価した結果，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) の性能が達成できた．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．

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」[8),9)] に適用することも，今後の課題である．

謝辞 本研究に関して，有益な助言をいただいた平田圭二研究員を始めとする，ICOT 第1研究室および関係会社の方々に感謝する．また，本研究の機会をいただいた ICOT の渕一博研究所長，瀧和男第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) Baker, H. G.: List Processing in Real Time on a Serial Computer, *Comm. ACM*, Vol. 21, No. 4, pp. 280-294 (1978).
2) Chikayama, T. and Kimura, Y.: Multiple Reference Management in Flat GHC, *Fourth International Conference on Logic Programming*, pp. 276-293 (1987).
3) Cohen, J.: Garbage Collection of Linked Data Structures, *ACM Comput. Surv.*, Vol. 13, No. 3, pp. 341-367 (1981).
4) Crammond, J. A.: A Garbage Collection Algorithm for Shared Memory Parallel Processors, *International Journal of Parallel Programming*, Vol. 17, No. 6, pp. 497-522 (1988).
5) Goto, A., Sato, M., Nakajima, K., Taki, K. and Mastumoto, A.: Overview of the Parallel Inference Machine Architecture (PIM), *International Conference on Fifth Generation Computer Systems*, pp. 208-229 (1988).
6) Halstead, R., Jr.: Multilisp: A Language for Concurrent Symbolic Computation, *ACM Trans. Prog. Lang. Syst.*, Vol. 7, No. 4, pp. 501-538 (1985).
7) 日比野：ガーベジコレクションとそのハードウェア，情報処理，Vol. 23, No. 8, pp. 730-741 (1982).
8) Nakajima, K.: Piling GC: Efficient Garbage Collection for AI Languages, *IFIP Working Conference on Parallel Processing*, pp. 201-204 (1988).
9) 小沢，細井，服部：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，Vol. 30, No. 9, pp. 1182-1188 (1989).
10) 佐藤，後藤：KL1 並列処理系の評価—メモリ消費特性とゴミ集め—，情報処理学会論文誌，Vol. 30, No. 12, pp. 1620-1627 (1989).
11) Tick, E.: *Parallel Logic Programming*, p. 486, MIT Press (1991).
12) Ueda, K. and Chikayama, T.: Design of the Kernel Language for the Parallel Inference Machine, *Comput. J.*, Vol. 33, No. 6, pp. 494-500 (1990).
13) 山本，今井，中川，川合，仲瀬，中越，宮崎，堂前：並列推論マシン PIM における抽象機械語 KL 1-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).

付録 並列 GC アルゴリズム

図 4～6 に本稿で述べた GC の並列実行方式のアルゴリズムを示す．

図4は，すべてのサイズのページを走査する scan-all() と，サイズ n のオブジェクトが入ったページを走査する scan() のアルゴリズムである．走査処理が Baker のアルゴリズムと異なる理由は，S_n の指す先を走査した時に 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5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())，移動先アドレスを書き込み，ロックを解除し (V())，オブジェクトをコ

```
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 4 全ヒープのスキャン
Fig. 4 Scanning all the heaps.

```
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 5 オブジェクトのコピー
Fig. 5 Copying objects.

```
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 6 領域オーバフローのチェック
Fig. 6 Checking overflow.

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をコピーする場合，そのオブジェクトをコピーするための領域を 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 6 の check $B()$ であらかじめ割り付けられている．

最後に，B がページの境界を越えたかどうかの検査を行う check $B()$ と，S がサブページの境界を越えたかどうかの検査を行う check $S()$ を図 6 に示す．いずれの場合も，境界を越えていた場合には，必要ならば，未走査領域を共有プールへ入れたり，新しいページの割付を行う．

(平成 3 年 7 月 31 日受付)
(平成 3 年 11 月 5 日採録)

今井　明（正会員）

1963 年生．1987 年 3 月，大阪大学工学部通信工学科卒業．同年シャープ(株)入社．同年(財)新世代コンピュータ技術開発機構 (ICOT) へ出向．現在，同機構第 1 研究室．並列推論マシン上の言語処理系に関する研究開発に従事．並列処理，信号処理言語の実装方式に興味を持つ．

エヴァン　ティック

マサチューセッツ工科大学電気工学科卒業 (1982 年)．同修士．1987 年スタンフォード大学電気工学科 Ph. D. 取得．1990 年より，オレゴン大学計算機情報科学科助教授．並列コンピュータアーキテクチャ，並行言語のコンパイル技法，論理プログラミングに興味を持つ．

中島　克人（正会員）

1953 年生．1977 年京都大学工学部電気工学第 2 学科卒業．1979 年同大学院修士課程修了．同年三菱電機(株)に入社．1985 年から 1989 年まで(財)新世代コンピュータ技術開発機構に出向．現在，同社情報電子研究所次世代コンピュータ技術開発部主事．主に高級言語マシンのアーキテクチャ研究に従事し，これまでに PSI, PSI-Ⅱ などの Prolog マシン，マルチ PSI, PIM などの並列推論マシンを研究開発．

後藤　厚宏（正会員）

1956 年生．1979 年東京大学工学部電子工学科卒業．1981 年同大学院情報工学専門課程修士課程修了．1984 年同博士課程修了．工学博士．同年日本電信電話公社(現 NTT) 研究所入社．1985 年より (財) 新世代コンピュータ技術開発機構へ出向．並列推論マシンの研究開発に従事．現在，NTT ソフトウェア研究所に勤務．電子情報通信学会，IEEE，ACM 各会員．