記 号 処 理 67－5
(1 9 9 3. 1. 2 2)

並列 GC Lisp の UNIX 上への実装と評価

松井祥悟
神奈川大学理学部

田中良夫、前田敦司、 高橋尚子、 中西正和
慶應義塾大学理工学部

慶應義塾大学理工学部で開発された Lisp 専用機 Synapse で用いた並列ガーベジコレクションアルゴリズムを、汎用 OS で用意されている並列処理機能を用いて実装した。実装法、効率の評価、改善法について検討した。実装上の問題点である root の挿入およびセル書き換え時のポインタの受け渡しは、セマフォおよびコンディションウェイトの並列処理の基本操作だけで実装できた。効率の評価は、従来の停止型マーク・スイープ法との比較により行った。アルゴリズムの改善の方法として、partial marking によるマーキングコストの低減法を提案した。従来法の 1/2 であった GC 効率が同等まで改善された。

An implementation and evaluation of parallel garbage collector on Unix workstations

Shogo MATSUI
Faculty of Science, Kanagawa University

Yoshio TANAKA, Atsushi MAEDA, Naoko TAKAHASHI,
and Masakazu NAKANISHI
Faculty of Science and Technology, Keio University

We describe a parallel garbage collector implemented on Unix workstations, based on the algorithm used in Lisp machine SYNAPSE. Implementation issues, evaluation of efficiency, and improvement in algorithm are discussed in order. We show that a limited set of parallel processing primitives suffices to implement root insertion and passing of pointers on modifying cell contents. Efficiency of the collector is evaluated in comparison with ordinary (sequential) mark and sweep collector. We propose a method called partial marking, which improves efficiency by decreasing marking cost. Almost the same efficiency as the sequential collector is achieved, which is twice as efficient as the original algorithm.

1 まえがき

慶應義塾大学数理科学科中西研究室では、Synapse という共有メモリ型マルチユーザマルチプロセッサ Lisp マシンの開発を行った [4, 5, 8, 9]。この Synapse の特徴の1つに、複数個の専用プロセッサによる並列ガーベジコレクション（以下並列 GC）があげられる。この並列 GC は、mark & sweep 法を並列化したものである。

本稿は、この並列 GC アルゴリズムの汎用 OS 上の並列処理環境への実装についての報告である。目的は、実装方法の検討、効率の評価、リストプロセスの無停止性（実時間性）の評価である。前提として、リストプロセス (LP) とガーベジコレクションプロセス (GC) をそれぞれ1プロセス (thread) ずつ割り当てる場合だけを考えた。実際には、SunOS の light weight process、LUNA の machOS 上の C-Thread libraly を用いて実装した。

2 Synapse のアルゴリズム

Synapse の開発開始当時、並列 GC のアルゴリズムは数種類発表されていたが、実際に実装され、動作の報告が行われているものは少なかった。

Synapse の GC は、当時もっとも効率がよいと思われた Kung らのアルゴリズム [2] に改良を加えたものである。Synapse ではこの GC アルゴリズムを複数の GC プロセッサで動作させた。また、ハードウェアのサポートを行い高効率化を図った。

2.1 Kung らのアルゴリズムの改良

Kung らの方法は、Dijkstra らの方法 [1] を改良したものである。走査と影付けという印付け方法 (marking) を、双頭待ち行列 (deque) をつかったプロセス間のポインタの受け渡しによるリストたどり法に改め、GC 効率の改善を図った。非可分処理が不要であることが特長であった。

Kung らの方法には、deque の実現法と、root の書き換え時の処理に問題が生じる。Synapse の並列 GC アルゴリズム（以下 Synapse GC） では、次のような改良を加えた。

deque の実現法

Kung らの方法では、非可分処理をなくすために deque と gray のタグを導入した。この deque は、実現が困難である。

Synapse GC では、これを、非可分処理を行う stack で実現した。stack が空である場合は、LP が新たなセルの書き換えを行わなかったことが保証される。これにより、GC タグは `black`, `white`, `off-white` という3つで済んだ。LP 側で行っていたタグの操作を省略することができた。

root の書き換え

Kung らのアルゴリズムでは、root ポインタは1つであり、生きているすべてのセルはそのポインタから到達可能であると仮定している。

実際には、stack や register からだけ到達できるセルが存在する。このため、stack への push, pop 動作や register の書き換えも、セルの書き換えとみなさなければ、正当性が保証されない。（明らかに、生きているセルが回収される例が存在する）。

したがって、上記操作の際にもポインタを deque へ積む必要がある。しかし、これらの操作は非常に頻繁に起こり（単なるセルの内容の読み出しを行っただけでも起こる）、これをこのとおり行うことは LP のオーバヘッドの極端な増大を招き、実用的でない。

Kung らの方法では、セル書き換え時のポインタの受け渡しは、上に書く新しいポインタを GC へ渡す方法をとっている。これに対して、日比野のアルゴリズム [3] では、書き換えられる古いポインタを受け渡す。Kung らのアルゴリズムでもこの日比野の受け渡し方法を用いると、セルのポインタを切断しても、そのポインタは GC に受け渡され必ず marking されることになる。これは、root insertion を行った時点の root から到達可能なセルは、セルの書き換えが起こった場合でも、必ず marking されることを意味する。この方法の場合、register や stack 上へのポインタのコピーでも不都合は生じず、deque への登録は不要になる。

複数個のプロセッサのサポート

複数個の LP と複数個の GC による動作をサポートするために、free list アクセス部分を非可分処理化した。

2.2 Synapse の並列 GC アルゴリズム

Synapse の GC アルゴリズムを図 1,2 に示す。セルは2つの pointer field (`left`, `right`) と tag field (`color`) を持つ。セルの総数は `M` である。 free list は `FREE` から指される。 `FREE.left` は free list

```
[Root insertion phase]
  begin
    MARKING := true ;
    << push all roots onto the stack >>
  end

[Marking phase]
  begin
    while << the stack is not empty >> do
      begin
        n := pop ;
        while (n <> NIL) and
              (n.left <> f) and
              (n.color <> black) do
          begin
            n.color := black ;
            push(n.right) ;
            n := n.left ;
          end
      end ;
    MARKING := false
  end

[Collecting phase]
  begin
    for i := 1 to M do
      if i.color = white then
        APPEND(i)
      else if i.left <> f then
        i.color := white
  end

procedure APPEND(n)
  begin
    n.color := off-white ;
    n.left := f ;
    n.right := NIL ;
    region _append do
      begin
        FREE.right.right := n ;
        FREE.right := n
      end
  end
```

図 1: Synapse GC アルゴリズム (GC)

```
Procedure LPa

  begin
    if MARKING then
      push(m.left) ;
    m.left := n
  end

Procedure LPc

  begin
    region _lpc
    when FREE.left <> FREE.right do
      begin
        NEW := FREE.left ;
        FREE.left := FREE.left.right ;
        NEW.left := m ;
        NEW.right := n
      end
  end
```

図 2: Synapse GC アルゴリズム (LP)

の先頭のセルを、FREE.right は free list の最後のセルを指す。また、free cell の left 部分には f という特殊なポインタが格納されている。

Synapse の GC アルゴリズムでは、root insertion phase において LP から集めた root につながるすべてのセルに対し、marking を行う。LP がセルの接続構造を変えた場合には、その変えた部分を GC に連絡するため、root insertion phase 時点で root から到達できたセルにはポインタの書き換えが起こっても到達可能である。この結果、LP セルの書き換えの有無に関わらず、GC は全く同じセルに対して marking を行うことになる。

また、free list は marking されない。free list につながるセルには、off-white というタグが付けられている。off-white のセルは、collecting phase では回収されない。また、LP が cons で獲得したセルは、直後の collecting phase では決して回収されない。直ちにゴミになる場合も次回以降の collecting phase で回収される。

3 並列GCの実装

並列GCはSunOS上のlight weight process、mach上のC-Thread libraryを用いて実装した。本実験では、LP、GCはそれぞれ1個ずつとした。

3.1 Lisp処理系

本実験で使用したLispは、慶應義塾大学中西研究室で開発されたklispをベースにした、Lisp 1.5の処理系である。今回並列GC化を行ったLispは、並列GC実装を前提にSunOSへ新たに移植したものである。C言語で記述されている。並列GC化を容易にするため、データ構造を簡単化し、klispで行われていたstackやa-listの抑制技法は取り除いた。多倍長演算が可能である。macroはFEXPRでサポートしている。実験用コンパイラを持つ。

並列GC用に追加されたデータ領域は、marking table、ps_stackである。marking tableは、GCタグを格納するtableである。セルと同じ数だけ、用意されている。タグは、`white`, `off-white`, `black`の3種類である。GC threadがmarking phaseである場合、ps_stackを用いてLP threadのセルの書き換え情報をGC threadへ送る。このps_stackの書き換えは、非可分処理化が行われている。

3.2 並列GC実装の問題点

root insertion

アルゴリズム上は、GCがroot insertion phaseになった場合には直ちにroot insertionを行わなければならない。Synapseの場合、マスタGCプロセッサがすべてのLPプロセッサに割り込みをかけ、一斉にroot insertion動作を行った。

SunOS lwp、mach C-Threadともに、これと同等の動作は記述できない。SunOS lwpでは、1CPUの疑似並列動作で実験を行ったため、高いpriorityを持つsignal処理用のthreadでこれを行った。mach C-Threadでは、LPのcons処理中のroot insertion要求フラグのチェックでこれを行った。

free cell empty

free listが空の場合、LPはwaiting状態になる。SunOS lwpの場合、message受け付け状態(message_recv)に入り、GCからのwake up messageを待つ。no free cell errorの検出はGCが行い、LPへmessageを送る。mach C-Threadの場合は、condition waitの状態になる。no free cell errorの検出はLPが行う。

3.3 SunOS上への並列GCの実装

次の4つのthreadに分かれている。

message handler thread

最も高いpriorityを持つmessage処理用のthread。起動後直ちにmessage受け付け状態(message_recv)に入り、waitingする。GC threadから送られるroot insertion messageを受け取るとLP threadのrootとなるすべてのポインタをgc_stackにpushする。このthreadは他のthreadより高いpriorityを持つため各threadの動作はブロックされる。これは結果的に割り込みと同じ効果をもたらす。

scheduler thread

2番めに高いpriorityを持つ。これより低いpriorityを持つLP threadとGC threadに対してスケジュリングを行う。

このthreadは、定められたtick時間ごとに、先の2つのthreadを交互にactiveにする。

LP thread

priorityはGC threadと同じである。もっとも低いpriorityを持つ。list処理を行う。通常のLispの処理系からGCの動作を取り除いたもの。
GC threadがmarking phaseである場合のセルの書き換え時に、GC threadに対してps_stackを用いてポインタの値を報告する。

free listがemptyの場合に、message受け付け状態(msg_recv)となりwaitingする。GC threadからのmessageにより、wake upする。GC threadからのmessageがno free cell (empty)である場合には、エラーとなる。

GCthread

priorityは、LP threadと同じである。root insertion phase, marking phase, collection phaseを繰り返す。root insertion phaseでは、message handler threadに対して、root insertion messageを送り、LP threadのrootポインタの転送を要求する。また、collecting phaseでは、LP thread

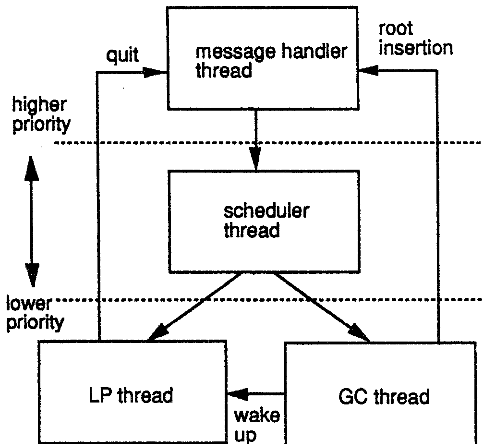

図 3: Light weight process への実装

がセル 待ちの waiting をしている場合には、 empty または filled message を送り wake up する。

図 3にこの関係を示す。

3.4 mach OS 上への並列 GC の実装

LUNA88k(machOS) へは、 C-Thread library を用いて実装した。次の 2 つの C thread に分かれている。

LP thread

list 処理を行う。通常の Lisp の処理系から、 GC の動作を取り除いたもの。 cons を行う直前に root insertion 要求フラグをチェックする。要求があれば、 root insertion を行う。 GC thread が marking phase である時にセルの書き換えが起こった場合、 GC thread に対して ps_stack を用いてポインタの値を報告する。

free list が空の場合、 LP は waiting 状態になる。 GC phase が marking の場合には、 LP は condition wait 状態となる。 GC は、 marking phase が終了した時点で、 LP が condition wait 状態であれば wake up を行う。 GC は waiting 状態に戻る。同じ waiting 状態で、 root insertion が 2 回行われた場合、セルをすべて消費したとみなし、 error となる。

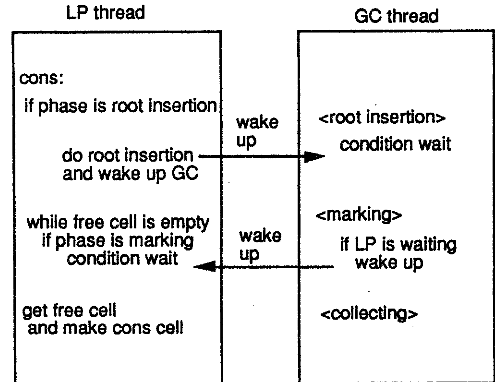

図 4: C-Thread への実装

GCthread

root insertion phase, marking phase, collection phase を繰り返す。 root insertion phase では、 LPthread に対して、 root insertion 要求フラグセットし、 condition wait 状態となる。また、 collecting phase では、 LP thread がセル 待ちの waiting をしている場合には、 LP を wake up する。

図 4にこの関係を示す。

4 並列 GC の評価

4.1 GC の効率

1 つの Lisp プログラムを、停止型 GC を持つ Lisp (以下 seq-lisp) と並列型 GC を持つ Lisp (以下 para-lisp) において実行した場合の処理時間を次のように定める。

$T_{seq.gc}$ seq-lisp の GC 時間の合計。

$T_{seq.lp}$ seq-lisp の LP 時間の合計。

$T_{seq.total}$ seq-lisp の全処理時間。

$$T_{seq.total} = T_{seq.gc} + T_{seq.lp}$$

$T_{para.gc}$ para-lisp の GC 時間の合計。

$T_{para.lp}$ para-lisp の LP 時間の合計。

$T_{para.total}$ para-lisp の全処理時間。

GC ratio G、Improvement ratio I を次のように定める。

$$G = \frac{T_{seq.gc}}{T_{seq.total}},$$

$$I = \frac{T_{seq.total} - T_{para.total}}{T_{seq.total}}.$$

この I は、seq-lisp の処理時間に対する para-lisp の処理時間の短縮の割合をあらわす。

次の仮定のもとで、G と I の関係を求める。

モデル

動作させる Lisp プログラムが安定であり、定常的にゴミを出す。

この場合、$T_{para.total}$ は次のようになる。

$$T_{para.total} = max(T_{para.lp}, T_{para.gc}).$$

したがって、I は、

$$I = min(\frac{T_{seq.total} - T_{para.lp}}{T_{seq.total}}, \frac{T_{seq.total} - T_{para.gc}}{T_{seq.total}}).$$

para-lisp の LP のオーバヘッド時間を $T_{para.oh}$、para-lisp と seq-lisp の GC 動作効率の比を n とすると、

$$T_{para.lp} = T_{seq.lp} + T_{para.oh},$$

$$T_{para.gc} = nT_{seq.gc} \ (n > 0).$$

overhead ratio を $O = T_{para.oh}/T_{seq.lp}$ とすると、I は、次のようにまとめられる。

$$I = min(G - O, 1 - nG).$$

4.2 parallel GC の評価

C-thread 上の 並列 GC Lisp で実験した G と I のグラフを図 5 に示す。

理想的な 並列 GC ($n = 1$, $O = 0$) も付記した。

実験に使用した関数は、結果を残さない cons を一定回繰り返す関数である。また、G を変えるために、数種の長さの固定リストを作り、同じ関数を実行した。

このグラフより、O は約 10%、n は約 2 であることが推定できる。この n の値は、Hickey らの解析 [6] と一致する。グラフの単調増加部分（G が

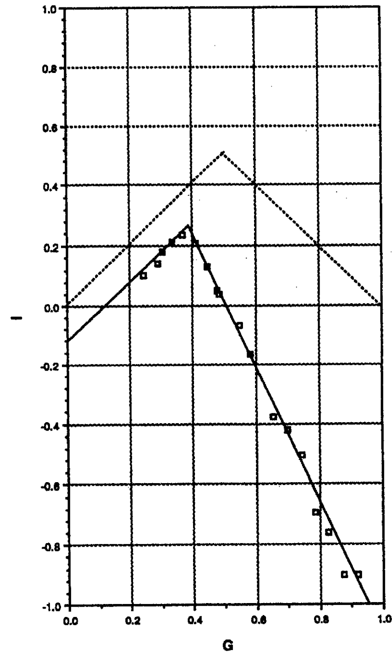

図 5: GC 率と改善率

0 から極大部分まで）は、Hickey らの論文の stable の状態（ LP が waiting を行わない状態）、単調減少部分のうち I が正の部分は alternating の状態（ 2GC サイクルに 1 度 LP が waiting を起こす状態）、単調減少部分のうちの負の部分は critical の状態（GC サイクル毎に LP が waiting を行う状態）である。G, I と Hickey らの解析の関連および Synapse の I の評価については、寺村 [10] が解析している。

n が 2 となる要因は、cons 後ただちにゴミとなるようなセルには off-white のタグが付加されているため、直後の collecting phase では回収されず、実際に回収されるまでに 2GC サイクル必要であることである。

4.3 効率の改善法

改善率を上昇させる方法として、次のようなものが考えられる。

1. GC 動作の抑制

   O は、LP と GC の共有資源アクセス競合が原因である。したがって、GC 動作を抑制す

ることで、O は小さくなる。実際には、free cell の個数により GC の起動を制御することが考えられる。

2. LP cons 時の GC タグの書き換え

   使用済みの off-white のセルの数を少なくするためには、off-white のセルの生成自体を抑制することが必要である。root insertion を行う前の cons では、セルは off-white である必要はない。これには cons 時の phase のチェックと GC タグの操作が必要である。Synapse GC アルゴリズムでは、本来 LP の cons 時には GC タグの操作は不要である。上記操作は、LP のオーバヘッドとなり、O が上昇する。この場合でも、n が十分小さい場合には、実時間性の向上につながる。

3. LP waiting 時の GC 動作の切り替え

   LP が waiting の状態では、GC は、停止型 mark & sweep 法と全く同じ動作を行ってもよい。これを行うことで、上で述べた回収に 2GC サイクル必要であった使用済みの off-white のセルも 1 回の GC で回収できる。これを行うためには、LP が waiting 状態になった時点で、GC サイクルを最初からやり直す必要がある。

4. marking 時間の短縮

   このアルゴリズムでは、ほとんどのゴミは 2 回目の collecting phase で回収される。したがって、回収効率は、停止型 mark & sweep 法の 1/2 である。しかし、marking には、毎回、停止型 mark & sweep 法と同じ時間かかる。この making のコストが、効率の悪さにつながっている。逆に考えると、回収効率が 1/2 程度であるならば、marking も毎回 full に行う必要はない。2 回に一度は簡単な partial marking で代用しても、連続した 2 つの GC サイクルをみれば、効率は変わらないことが予想される。partial marking は、直前に行われる full marking で付加された black の タグ を消去しないことで実現できる。これにより、平均的な marking 時間は減少し、n は減少する。これはミクロな generation scavenging GC ともいえる。

3 の方法を施した場合のグラフを図 6の improve #1 に示す。実装は、LP が waiting 状態になった時点で、root insertion を再度行う方法を取った。この場合、LP のセルの書き換え時に、切断されたポインタの他に、新たに上書きするポインタも stack に積む必要がある。グラフでは、seq-lisp より遅くなる場合はなくなる。明らかに改善されている。ただし、I のピークは移動しておらず、実時間性の向上には貢献していない。

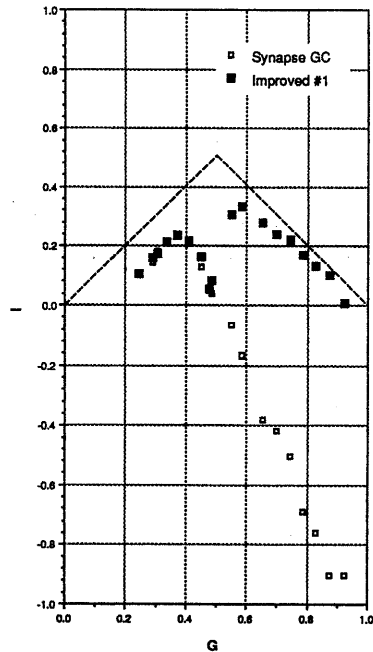

図 6: GC 率と改善率（改良法 3）

4 の方法を施した場合のグラフを図 6の improved #2 に示す。この場合も、full marking の marking phase から partial marking の marking phase 終了までの間に LP のセルの書き換えが起こった場合には、切断されたポインタの他に、新たに上書きするポインタも stack に積む必要がある。

グラフでは、明らかに改善されている。GC 効率は、停止型 mark & sweep 法と同等まで改善され、理想的な曲線に近付いている。また、3 の改良では改善されなかった実時間性を示す I のピークの位置も右に移動しており、GC 率 G が約 0.5 の付近まで実時間処理が可能であることを示している。

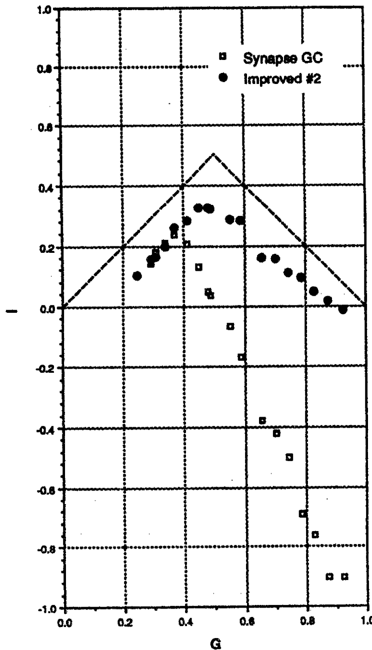

図 7: GC 率と改善率（改良法 4）

5 まとめ

本稿では、Lisp machine 上で動作する Synapse GC を汎用 OS 上に移植し、実装法、効率、アルゴリズムの改善法について検討した。

SunOS light weigth process, LUNA OS C-Thread ともに並列処理の基本機構であるセマフォおよびコンディションウェイトを用いて実装可能である。停止型 mark & sweep 法 と Synapse GC (LP 1 台、GC 1 台の動作) を比較することで、Synapse 型 GC の効率が停止型 mark & sweep 型の 1/2 であることを確認した。GC 率が上昇すると、並列 GC をもつ Lisp は従来の停止型 GC を持つ Lisp より処理速度が低下することを確認した。アルゴリズムの改良法の 1 つである partial marking 法は、GC 効率を停止型 GC と同程度まで改善する。並列 GC は、目的とする高速性および実時間性について十分配慮の上実装する必要がある。

今後の課題として、partial marking 実行比率を上げた場合および実行プログラムのタイプ別の有効性の検討が残されている。

謝辞

本研究は、慶應義塾大学と神奈川大学の両校において、wide 研究ネットワークを用いた、mail, ftp, talk により進められた。ここに、wide 研究ネットワーク関係者各位に感謝の意を表する。

参考文献

[1] Dijkstra, E. W. et al. On-the-fly garbage collection, An exercise in cooperation, in Lecture Note in Computer Science, Vol.46, Springer-Verlag, New York (1976) 43-56

[2] Kung, H. T. and Song, S. W. An Efficient Parallel Garbage Collection System and its Correctness Proof. Tech. Note, Dept. of Computer Science, Carnegie-Mellon University (Pittsburgh, Pennsylvania, 1977).

[3] 日比野靖 ガーベジコレクションとそのハードウェア化、情報処理、vol.23, No.8 (1982) 730-741

[4] 中西正和 他 マルチ CPU・LISP 処理系 SYNAPSE について、情報処理学会研究会報告 記号処理 22-3、(1983) 1-7

[5] 加藤良信 他 SYNAPSE の機能と動作について、情報処理学会研究会報告 記号処理 30-3、(1984) 1-7

[6] Hickey, T. et al. Perfromance Analysis of On-the-fly garbage collection, Comm. ACM, Vol.27, No.11 (1984) 1143-1154

[7] Gabriel, Richard P. Performance and Evaluation of Lisp systems. The MIT Press (Cambridge, Massachusetts, 1985)

[8] 松井祥悟 他 マルチプロセッサ Lisp マシン SYNAPSE のアーキテクチャ、電子情報通信学会研究会報告 CPSY86-53、(1987) 13-22

[9] Matsui, S. et al. SYNAPSE: A Multi-microprocessor Lisp Machine with Parallel Garbage Collector, Proceedings of the International Workshop on Parallel Algorithms and Architectures (1987) 131-137

[10] Teramura, S. Analysis of Parallel Garbage Collection With Multiple List Processors and Garbage Collectors, Journal of Information Processing, Vol.12, No.3, (1989) 229-238