Vol. 35 No. 12 情報処理学会論文誌 Dec. 1994

SIMD 型超並列計算機上の拡張 Common Lisp 処理系におけるごみ集めとその評価

安本太一[†] 湯淺太一[††] 貴島寿郎[††]

SIMD (Single Instruction, Multiple Data) 型超並列計算機上で動作する拡張 Common Lisp 言語処理系 TUPLE のごみ集めについて報告する．TUPLE は SIMD 型超並列計算の機能を Common Lisp に付加したもので，膨大な数のサブセット Common Lisp 処理系の並列動作により並列リスト処理を可能にしている．このために，各 PE (Processing Element) は，その局所メモリに独自のヒープを備えており，並列プログラムの実行効率の向上には，ごみ集めの並列化が不可欠である．本稿では，この TUPLE 処理系の並列ごみ集めについて，UNIX ワークステーションを FE（フロントエンド）とし，1024 台以上の PE を備えた SIMD 型超並列計算機 MasPar MP-1 における実装に焦点をあてて，議論を行う．一般的な SIMD 型並列計算機の特徴も十分考慮しており，他の計算機に TUPLE を実装した場合にも応用が可能である．

Garbage Collection of an Extended Common Lisp System for Massively Parallel SIMD Architecture and Its Evaluation

TAICHI YASUMOTO,[†] TAIICHI YUASA[††] and TOSHIRO KIJIMA[††]

We report the garbage collection of an extended Common Lisp system for massively parallel SIMD (Single Instruction, Multiple Data) architectures. We call this system and its language TUPLE. The TUPLE language is an extension of Common Lisp with features for SIMD massively parallel computation. By providing a huge number of subset Comnon Lisp systems running in parallel, TUPLE supports parallel list processing. For this purpose, each processing element (PE) of the target machine has its own heap in its local memory, and therefore we need parallelize the garabage collection in order to improve the performance of parallel programs. This paper describes the current implementation of the parallel garbage collection on the MasPar MP-1, a SIMD massively parallel computer with at least 1024 PEs. The presented techniques can be applied to other SIMD machines, since the discussions in the paper assume common features of SIMD architectures.

1. はじめに

TUPLE (Toyohashi University Parallel Lisp Environment) は，Common Lisp[3]に超並列計算の機能を付加したものである．TUPLE の処理系は，フルセットの Common Lisp 処理系である KCL (Kyoto Common Lisp)[4]をベースに実現されており，1024 台以上の PE を備えた SIMD 型超並列計算機 MasPar MP-1 上で動作している．KCL がC言語と Common Lisp で記述されているように，TUPLE 処理系は，MP-1 用の拡張C言語である MPL[9]と TUPLE 自身で記述されている．

TUPLE では，膨大な数の PE サブシステムとよばれるサブセットの Common Lisp 処理系が並列に動作し，これに加えて，フロントエンドシステムとよばれるフルセットの Common Lisp 処理系があり，ユーザはこれと対話することによって並列プログラムの開発と実行を行う．

各 PE サブシステムは固有のヒープをその局所メモリ上に持つので，TUPLE を構成する処理系のヒープの総計は膨大なものとなる．また，これらのヒープ内のいずれのオブジェクトも相互に参照が可能であるので，各処理系が単独でごみ集めを行うことはできない．ごみ集めも並列処理によって効率良く行うことが期待されるが，困難な問題もある．SIMD アーキテクチャにおける，PE 間や PE–FE 間の通信は比較的遅い．また，各 PE のアクティビティ (activity, 命令実

† 愛知教育大学総合科学課程
Faculty of Integrated Arts and Sciences, Aichi University of Education
†† 豊橋技術科学大学情報工学系
Department of Information and Computer Sciences, Toyohashi University of Technology

行権）が不揃いになると実行効率が低下するという短所は，ごみ集めにとって不利な条件である．しかしながら，このような困難を克服して高速なごみ集めを実装しなければ，並列リスト処理の恩恵を享受することはできない．

本稿では，この TUPLE 処理系のごみ集めについて報告する．次章で MP-1 における TUPLE 処理系のメモリ管理について述べ，3章で SIMD アーキテクチャの特徴を考慮したごみ集めのアルゴリズムを提案する．4章で評価実験の結果とあわせてその有効性を議論する．TUPLE の言語の詳細については文献1),6),8)にゆずることにするが，Lisp の一般的な用語に“PE”が前置される場合は，特に断りがない限り“PE サブシステムに配置された”と解釈していただきたい（例えば，PE 変数というのは，各 PE サブシステム上の変数という意味である）．また，MP-1 における TUPLE 処理系全体の性能評価の詳細については文献5),8)を参照されたい．

2. MP-1 における TUPLE 処理系のメモリ管理

MP-1 は，1024 台以上の PE をもつ SIMD 型の並列計算機であり，FE としての UNIX ワークステーションとバックエンド（back-end）の2つの部分から構成されている．さらに，バックエンドは，各 PE へ命令をブロードキャストする ACU（array control unit）と，この命令を実行する PE が二次元格子上に配置された PE アレイに分けられる．各 PE は独自の命令フローをもたず，ブロードキャストされた命令を，ある条件を満たしている時に実行する．バックエンドに搭載されているメモリの容量は比較的少なく，ACU には 128k バイト，PE には 16k バイトであり，仮想記憶はいずれにもサポートされていない．

FE とバックエンド，ACU と PE，PE と PE の間のデータのやりとりは通信によって行われ，通常のメモリアクセスと比べて遅い．FE 内で1ワード（4バイト）の代入に要する時間を1とすると，FE と ACU の間は約600，FE と PE の間は約15000，ACU と PE の間は約20の時間を要する．前者2つは著しく遅いが，バッファリングによって実効速度を向上することが可能である．また，PE 間は，近傍通信のためのメッシュと大域通信のためのルータによって結ばれている．これらのネットワークの特徴を生かした様々な通信ライブラリが提供されているが，メッシュ通信を用いた隣接 PE との通信でさえも約60の時間を要するので，通信回数を抑制するようにつとめなければならない．

2.1 ヒープ

TUPLE は**図1**に示すように，2種類の Lisp 処理系から構成されている．フロントエンドにはフロントエンドシステムが，バックエンドには 1024 個以上の PE サブシステムが配置されている．並列計算は，PE サブシステムに PE 式を渡すことによって，開始される．PE サブシステムに搭載されているメモリは少ないが，記号処理・リスト処理が可能な必要最低限の機能は備えている．命令を供給する ACU には，並列に実行される Lisp 関数である PE 関数の本体があり，各 PE はそれらの PE 関数が操作するデータを持っている．

以上のことから，TUPLE のヒープを次の3つに分けることができる．

- 通常の Common Lisp のデータを配置する FE ヒープ
- PE の cons セルを配置する PE ヒープ
- すべての PE サブシステムが共有するデータ（組込みおよびユーザが定義した PE 関数や PE ベクタのヘッダなど）を配置する ACU ヒープ

いずれのヒープ上のオブジェクトも，MP-1 のすべての構成要素（FE，ACU，PE）から相互に参照される．例えば，FE ヒープ上のオブジェクトは，ユーザが定義した PE 関数の一部として ACU から参照されたり，ブロードキャストの結果として各 PE から参照される．また，PE ヒープ上の cons セルは，リダクション（reduction，各 PE がもつデータを集め，

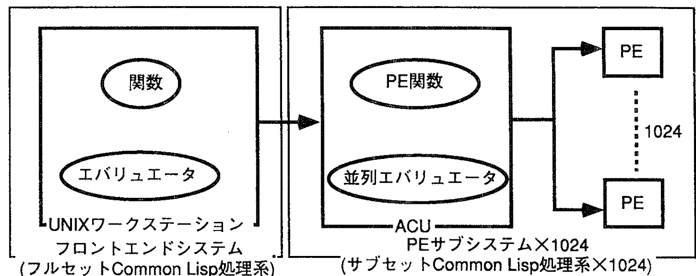


図1 MasPar MP-1 における TUPLE の構成
Fig. 1 Structure of the TUPLE system on MasPar MP-1.

処理をほどこして1つのデータにする操作）の結果として FE から参照されたり，PE 間通信によって他の PE から参照されたり，あるいは ACU からも参照される．

2.2 データの内部表現とその配置

PE サブシステムは，並列リスト処理のための cons に加えて，固定長短形式の整数（fixnum），単精度浮動小数点数（short-float），文字（character），ベクタ（vector，1次元の配列）をサポートしている．これらのうち，整数，単精度浮動小数点数，文字については，KCL と同様に即値データとしてポインタコーディングされているので，実質的にはメモリを消費しない[7]．

各 PE のデータ領域のうち，ごみ集めに関連する部分を**図2**に示す．MPL の使用する実行時スタックは記載していない．メモリ領域の先頭には，組込みの PE 定数である **nil**，**t**，**penumber**（プロセッサ番号）が割り当てられている．続いて，PE の大域領域があり，ユーザが定義した大域的 PE 変数と PE 定数，それに PE ベクタの本体が配置される．この大域領域は，ユーザが大域的 PE 変数などを定義するたびに，高位アドレスの方へ動的に拡張される．次に PE スタックの領域があり，局所的 PE 変数や PE 関数への引数などの一時的なデータが配置される．

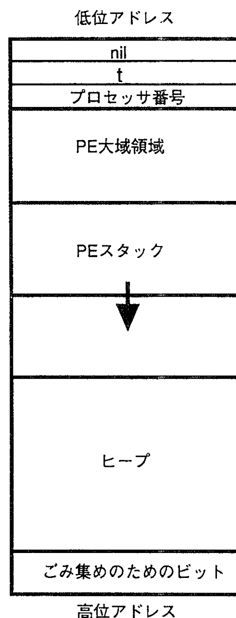


図2　PE のデータ領域
Fig. 2　The PE data area.

さらにその後には，PE cons セルが配置されるヒープ領域がある．PE あたり 16 k バイトの局所メモリを持つモデルでは，データ領域の大きさは 8 k バイトで，その半分の 4 k バイトをヒープ領域として割り当てた．1個の PE セルは car 部と cdr 部を合わせて2ワード（8バイト）を占めるので，各 PE ごとに 512 個のセルが使用可能である．ごみ集めのためのマークビットは，cons セルに隣接するのが普通であるが，MPL コンパイラのデータアライメントによって生じる未使用領域を無視できないほどデータ領域が小さいので，セル本体とは分離してデータ領域の最後にまとめて配置した．このようなマークビットの配置は実行効率の面から好ましくないように思われるが，1回の整数演算で 32 個のセルを調べることができるなどの利点もあり，TUPLE の並列ごみ集めはこれを積極的に利用している．

一方，ACU ヒープ内のセルは1種類であり，各 ACU セルは4ワードを占める．最初の1ワードはデータ識別タグとごみ集め用マークビットとして使用される．残りの3ワードの用途は，データ識別タグの内容によって異なるが，ごみ集めの観点からは，単に，ACU セルや他のヒープ上のセルへのポインタを3つ格納するセルと考えてよい．

3. ごみ集め

TUPLE のベースである KCL のごみ集めは，いわゆるマーク・スイープ方式を用いている．再利用可能なセルは大きさごとに区分され，その大きさごとに再利用可能なセルを連結したフリーリスト（free list）を構成する．また，配列のような可変長オブジェクトは，本体へのポインタを持つ固定長のヘッダセルとその本体から構成される．配列の場合を例にとると，要素は本体に格納され，ヘッダセルには次元数などのさまざまな情報が格納される．本体は，再配置可能領域とよばれる特別な領域に配置され，スイープフェイズにおいて，大きな空き領域が得られるように再配置される．この手法は，KCL がC言語で記述されていることに起因している．配列の本体は必ずヘッダを経由して参照されるので，本体の再配置が行われても，KCL のオブジェクトを指しているCの変数の内容を変更する必要はない．これによって，Cコンパイラによる変数配置の相違を考慮する必要がなくなり，KCL の移植性を高めている．

TUPLE はC言語を並列データが扱えるように拡

張したMPLで記述されていることもあり，基本的にはバックエンドのごみ集めもマーク・スイープ方式を採用することにした．バックエンドのヒープ内のセルは，ACU, PEともにすべて同じ大きさであるので，ACU，各 PE のヒープにそれぞれ1つのフリーリストを用意すればよく，空き領域の断片化も生じない．近年の逐次型 Lisp 処理系はコピー方式のごみ集めを採用することが多い．コピー方式は，(1)空き領域に断片化が生じない，(2)処理時間がメモリ空間全体ではなく，ごみでないメモリ領域の量のみに依存する，という長所がある．しかし，前者は，上記のことからTUPLE においては有効でない．また，一般的にSIMD 方式超並列計算機の各 PE のメモリは大きくないので，後者もほとんどプラスにならない．むしろ，コピー方式が使用可能セルの2倍のメモリを必要とすることは，通常の逐次計算機などより深刻な問題となる．さらに，アドレス書換えのオーバーヘッドを考慮すると，マーク・スイープ方式の方がこの場合明らかに適している．

TUPLE におけるごみ集めは，フリーリストの1つが空になるか，あるいは FE の再配置可能領域の空き領域がなくなった時に起動される．したがって，KCL と同じように，ごみ集めは以下の2つのフェイズからなる．

1. マークフェイズ：使用中のすべてのセルがマークされる．
2. スイープフェイズ：マークされていないセル（ごみとなったセル）がフリーリストに連結され，可変長オブジェクトについては，マークされたすべてのヘッダセルが指す本体が再配置される．

このスイープフェイズは，以下のような理由により，MP-1 の各構成要素ごとに独立して実行できる．

- 各構成要素中の未使用セルは，同一構成要素内のフリーリストに連結される．
- バックエンドから FE 内のオブジェクトを指すことはあっても，その本体を直接指すのではなく，必ずヘッダを経由して間接的に指すだけである．したがって，FE 内の本体が再配置された場合でも，バックエンド内のポインタを変更する必要はない．

FE 側のスイープフェイズルーチンは，KCL のものを変更せずに同じものを使用した．ACU 側のスイープフェイズルーチンは簡単であり，ACU ヒープを走査し，マークされていないセルを ACU のフリーリストに連結するだけである．個々の PE においても，ACU のスイープルーチンと同様なものが，すべての PE において並列に実行される．PE 側のセルは再配置されることはないので，個々の PE 間の同期は不要であり，同期にともなうオーバーヘッドもない．

一方，マークフェイズは，次節で示すような数多くの問題点があり，その実装は複雑なものとなった．そこで，今回特に工夫をした PE サブシステムのためのマークアルゴリズムについて詳細に述べた後，マークフェイズの全体のながれを追うことにする．

3.1 マークフェイズ実装上の問題点

これまでの議論から，マークフェイズの実装にあたって，留意しなければならない TUPLE および MP-1 の特徴は以下のとおりである．なお，これらの大部分は，他の SIMD 型並列計算機に TUPLE を実装した場合にも，あてはまるものと思われる．

1. FE，ACU，各 PE にヒープが配置されている．各 PE のヒープの内容はそれぞれ異なる．
2. 各ヒープ内のセルは，他のヒープ内のセルからも参照される．
3. PE 間通信は比較的遅く，FE とバックエンドの間の通信はきわめて遅い．
4. 1つの PE に搭載されているメモリは比較的少ない．
5. PE は同時に1つの命令のみを並列に実行できる．

1と2から，個々のヒープのマークを独立に行うことができないうえ，マークを行う際に MP-1 の構成要素間の通信が必要であることがわかる．さらに，3からは，これらの通信量の抑制の成否が，ごみ集めによる実行中断時間に大きな影響を与えることが予想される．また，4から，システムスタックの多用は期待できず，再帰的なアルゴリズムを採用することは難しい．そして，1と5からは，PE ごとにヒープの内容が異なる状況下で，アクティビティの不揃いを抑制しながら，効率良くマークを行わなければならない．

3.2 PE セルのマーク

PE セルのマークを効率よく並列に行うために，従来のマークビットのほかに，要求ビット（request bit）という特別なビットを各 PE セルに対応させ，マークビット群の後に配置した．この要求ビットは，マークを予約するためのビットである．PE セルへのポインタを再帰的にたどってマークを行うのではなく，PE ヒープ領域の走査を以下のように繰り返す．

走査中に，要求ビットがオンになっている PE セルに出あったら，そのマークビットをオンにし，その PE セルの car 部と cdr 部が指す PE セルの要求ビットをオンにする．そして，次の PE セルの要求ビットを調べるということを，要求ビットがオンにされなくなるまでくり返す．

```
/*PE マークルーチン*/
more:=true;
while more do
  more:=false;
  for i from 0 to M-1 do
    if i.request then
      if not i.mark then
        mark_object(i.car);
        mark_object(i.cdr);
        i.mark:=true;
        more:=true;
      endif
      i.request:=false;
    endif
  endfor
endwhile
```

ここで，M は各 PE ヒープにおけるセルの数であり，*i.mark* と *i.request* はマークビットと要求ビットを，*i.car* と *i.cdr* は i 番目のセルの car 部と cdr 部をそれぞれ表わす．要求ビットの初期化については後述することにする．

このアルゴリズムは，SIMD 型並列計算機で，すべての PE によって並列に実行するように考案したものであり，MPL のような SIMD 型並列言語を用いて実装するのに適している．また，セルを再帰的にたどらないためスタックを多量に使用することがなく，メモリ容量が少ない場合に適している．再帰にともなうスタックの消費がないかわりに，PE セルが1個あたり1ビットの要求ビットを必要とするが，32 ビットアーキテクチャの場合，PE ごとに高々 $M/32$ ワード確保できればよいので問題にはならない．次に，このアルゴリズムの効率であるが，最悪の場合 M^2 時間を必要とする．すなわち，すべての i 番目のセル ($0<i<M$) が $i-1$ 番目のセルを指していた場合は，**while** の中では1回の繰り返しにつき1つのセルしかマークできないからである．一方，典型的な再帰マークアルゴリズムでは，最悪の場合でも時間 M で終了するように思われる．しかし，対象としているのは SIMD アーキテクチャであり，たとえ各 PE が時間 M でマークを終ったとしても，最悪の場合，N を PE の総数とすると全体として時間 MN を要する可能性がある．なぜならば，TUPLE では，PE ごとに独立なセル配置が可能であるので，各 PE のヒープの内容が異なる場合は処理の分岐が起こり効率が著しく低下する．現在の実装では，MP-1 は 1024 台の PE をもっており，$M=512$, $N=1024$ となる．したがって，最悪の場合は，典型的な再帰アルゴリズムを並列化したものより，本アルゴリズムの方が効率がよく，PE 数が多いほどこの傾向は強くなる．

3.3 PE からのマーク

前節のマークルーチンにおける並列サブルーチン *mark_object* は，各 PE のオブジェクト x を受け取り，以下のように処理を行う．

1. x が PE セルへのポインタであれば，その PE セルの要求ビットをオンにする．
2. もし，x が FE セルへのポインタであれば，**FE バッファ**の中に格納する．詳細については，後述する．
3. もし，x が ACU セルへのポインタであれば，ACU のマークルーチンを呼び出す．

1と2については各 PE により並列に実行されるが，ACU は単一プロセッサで構成されており ACU のマークルーチンが同時に複数のポインタを扱うことが不可能であるため，3は逐次的に実行される．1では，他の PE のセルへのポインタであっても，その PE の要求ビットをオンにするだけであり，その要求ビットをオンにされた PE セルを所有している PE 自身が，最終的なマークを行うので，PE 間通信は高々1回でよい．また2では，FE とバックエンドの間の通信が必要であるが，この通信はバックエンドの構成要素間の通信に比べるとはるかに時間がかかる（2章参照）．そこで，通信回数を減らすために，FE へのポインタをバッファリングしている．このためのバッファは，各 PE に1ワードずつ割り当てられており，全体としては PE と同数のポインタを格納することができる．これらのバッファに空きがなくなった時に，バッファに格納されていたポインタが FE にブロック転送され，FE のマークルーチンが呼ばれる．

PE が FE セルへのポインタ p を *mark_object* への引数として受け取ると，まずはじめに，p をその PE 自身のバッファ（以後この1ワードのバッファを w と記す）に格納しようと試みるが，過去の *mark_*

object の呼び出しによって，既にそのバッファ w が FE へのポインタで満たされているかもしれない．しかし，そのような場合でも他の PE のバッファ w が空であるかもしれない．また，各 PE のバッファ w がすべて満たされていても，これらのバッファの内容に重複したものがあれば，これらを取り除いて空き領域を得ることができる．これらの考察によって，FE へのブロック転送の回数を減らすことができる場合がある．

以下に示す並列アルゴリズムは，バッファ w が既に満たされている PE が1つ以上ある場合に実行されるものである．N を PE の数とし，すべての PE に1から N まで番号づけを行った時の i 番目の PE を P_i と表す．また，*occupied* な PE は *mark_object* への引数として FE セルへのポインタを受けとっていてかつバッファ w が既に満たされている PE，*free* な PE は FE セル以外のオブジェクトへのポインタを受け取っていてかつバッファ w が空である PE を表す．

1. occupied な PE の各々について，序数 *ord* をふっていく．これらの数の中で最大のものを N_s とする．すなわち，occupied な PE の総数を N_s とする．そして，序数をふったときの i 番目の PE のプロセッサ番号を s_i とする．
2. free な PE の各々について，序数 *ord* をふっていく．これらの数の中で最大のものを N_d とする．すなわち，free な PE の総数を N_d とする．そして，序数をふったときの i 番目の PE のプロセッサ番号を d_i とする．
3. $1 \leq i \leq min(N_s, N_d)$ なる occupied な PE P_{s_i} の各々について，各 PE が持っているポインタ p を i 番目の free な PE P_{d_i} に格納する．この手順は，以下のように，さらに細かく分けることができる．
   (a) $1 \leq i \leq min(N_s, N_d)$ である各 occupied な PE P_{s_i} について
   $tmp \odot ord \Leftarrow p$
   (b) $1 \leq i \leq min(N_s, N_d)$ である各 free な PE P_{d_i} について
   $dest \odot ord \Leftarrow d_i$
   (c) $1 \leq i \leq min(N_s, N_d)$ である P_i について
   $w \odot dest \Leftarrow tmp$

   ここで，"$x \odot y \Leftarrow z$" は，P_y の x に z の値を代入することを表す．この操作は，P_y への PE 間通信をともなう．
4. もし，$N_d < N_s$ ならばバッファが満杯になっているので，重複しているポインタを以下のようにして取り除く．
   (a) $1 < i < N$ なるすべての P_i について，P_{i-1} の w の値が P_i の w の値より小さいか等しくなるように，バッファ内のポインタのソーティングを行う．
   (b) $1 < i \leq N$ なる各々の P_i について，P_i の w が P_{i-1} の w と等しい場合は，P_i のバッファ w を空にする．

   そして，1から3までを，もう一度繰り返す．それでもなお，$N_d < N_s$ ならば，このバッファのすべてを FE に転送し，FE のマークルーチンを呼び出す．マークルーチンから戻った後，以下を行う．
   (a) すべてのバッファを空にする．
   (b) $N_d < i \leq N_s$ である各々の occupied P_{s_i} について
   $w := p$

このアルゴリズムにともなう操作も，MP-1 のような SIMD アーキテクチャ上で効率良く実装することができる．例えば，1と2における序数割り付けや4におけるバッファ全体のソーティングは，$\log N$ 時間の並列アルゴリズムを用いることが可能である[2]．MP-1 ではこのようなアルゴリズムはライブラリとして提供されている．3では PE 間通信を3回行っているが，これらの PE 間通信は，PE 間ネットワークの特徴を生かしたアルゴリズムを用いることにより，通信時間を抑えることができる．MP-1 を含めた最近の多くの SIMD 型並列計算機ではそのようなライブラリが用意されている．また，PE 間の物理的な通信速度は FE とバックエンドの間に比べると高速であることから，3の PE 間通信は高速に実行される．

以上で述べたバッファリングの効果を概算する．まず，バッファリングを行わずに M 個のポインタを転送するときに要する時間は，PE から FE へ1ワードの転送に要する時間を T_{PF} とすると，

$$MT_{PF} \quad (1)$$

である．一方，バッファリングを行って，PE から FE へ，M 個のポインタを転送するときは，

$$B(M'') + KT_{PP}M' \quad (2)$$

の時間を要する．ここで，

T_{PP}：PE 間における1ワードの転送時間

$B(x)$：PE と FE 間の間で，x ワードのデータをブロック転送するのに要する時間

M'：空きバッファを探す回数（バッファが空いていたり，複数の PE が同時に空きバッファを探すことがあるので $M'<M$）

M''：M 個のポインタのうち異なるものの個数（$M''<M$）

K：序数割り付けやソーティングの性能によってきまる定数

である．第1項（$B(M'')$）は FE へのポインタ（重複するものは除く）をブロック転送するのに要する時間である．第2項（$KT_{PP}M'$）は PE 側で空きバッファの探索を M' 回行った時に要する時間である．式(1)と比較するために，式(2)を

$$MT_{PF}\left(\frac{B(M'')}{MT_{PF}}+K\frac{T_{PP}}{T_{PF}}\frac{M'}{M}\right) \qquad (3)$$

と変形する．括弧の部分が通信時間短縮率であり，1より小さくなれば，バッファリングの効果はない．括弧の中の第1項はブロック転送によるスピードアップであり，$B(M'')\ll M''T_{PF}<MT_{PF}$ である．第2項は PE 側で空きバッファを探すためのオーバーヘッドであり，この値が小さくなければブロック転送の効果が損なわれてしまう．序数割り付けやソーティングのアルゴリズムがネットワークの特徴を生かした効率のよいもので，かつ PE 間通信が PE-FE 間通信より十分速いことが要求されることがわかる．MP-1 では，おおよそ $T_{PP}/T_{PF}=0.004$ であり，M' が現実的には M よりはるかに小さいことから，通信時間短縮の可能性がきわめて高い．

3.4 マークフェイズ

アルゴリズムの簡略化のため，TUPLE は，常にごみ集めを FE のマークルーチンのトップレベルから開始し，ACU セルのマーク，PE セルのマークの順に行う．このトップレベルでは，FE 内の使用中セルのすべての起点からポインタをたどり，その軌跡に現れたすべての FE セルをマークする．また，バックエンドヒープにあるセルが現れた場合は，一時的にそのポインタをバッファに格納し，FE におけるマークを続行する．バックエンドへのポインタを格納するために，ACU へのポインタを格納する **ACU バッファ**と PE へのポインタを格納する **PE バッファ**の2つを設けた．これらのバッファは，FE とバックエンドの間の通信に伴うオーバーヘッドを抑制するために用いられる．

FE におけるマーキング中に ACU バッファが満たされると，バッファ内のポインタは ACU にブロック転送され，ACU のマークルーチンが起動される．一方，PE バッファが満たされると，PE のマーク要求ルーチンが起動され，PE バッファ内に格納されているポインタから指されている PE セルに対応する要求ビットをオンにする．この要求ビットは PE における使用中セルの起点となる．

先述の PE マークルーチンは，マークルーチンのトップレベル実行中には起動されず，FE メモリ内のすべての起点からの走査が終了した後にのみ起動される．よって，PE マークルーチンの開始時には，その時点で FE から指されていることが判明しているすべての PE セルの要求ビットがオンにされ，PE マークルーチンの走査回数を抑えることができる．

ACU マークルーチンはポインタをたどり，見つかったすべての ACU セルをすべてマークし，PE セルへのポインタがみつかった時は，PE セルの要求ビットをオンにするための通信を1回行うだけである．PE マークルーチンの起動を後回しにすることによって，PE マークルーチンを引き続いて起動した時に，要求ビットがオンにされている PE セルを，可能なかぎり多くすることができる．ACU マークルーチンが，FE セルへのポインタをみつけた時は，ただちに FE マークルーチンを呼び出すことはせず，PE から FE へのポインタを格納する FE バッファに格納する．ACU 内に別のバッファを設けず，このバッファを用いることによって，前述のように並列的に効率よく，重複するポインタを取り除くことができる．

なお，ACU セルおよび PE セルのマーク中に，FE へのポインタがみつかった場合は，再び FE からのマークを行う．FE，ACU，PE の使用中セルのすべてのマークが完了した時に，マークフェイズは終了する．

前節と同様に，ACU から FE へのバッファリングを行った時のおおよその通信時間を求める．今回は M は ACU から PE へのポインタ数である．

$$B(M'')+KT_{PP}M'+MT_{AP} \qquad (4)$$

$$=MT_{AF}\left(\frac{B(M'')}{MT_{AF}}+K\frac{T_{PP}}{T_{AF}}\frac{M'}{M}+\frac{T_{AP}}{T_{AF}}\right) \qquad (5)$$

となる．T_{AP} は ACU から PE への1ワードの転送時間，T_{AF} は ACU から FE への1ワードの転送時間であり，その他の記号は前節と同じである．式(4)の第1項と第2項は式(2)とほぼ同じ形をしており，

さらに FE セルへのポインタを ACU から PE に転送する時間が加算されている．第3項（に対応する処理）は必ず単独で実行されるが，第1，2項については式(2)と兼ねて行われる場合があり，その時には実質的な通信時間は短くなる．式(5)も，式(3)と同様に，バッファリングを行わない場合の通信時間と通信時間短縮率の積となる．この通信時間短縮率が1より小さいためには，ブロック転送の効果があり，かつ PE 間と ACU-PE 間の通信が ACU-FE 間より十分速くなければならない．MP-1 では，おおよそ，$T_{PP}/T_{AF}=0.1$，$T_{AP}/T_{AF}=0.03$ であり，通信時間短縮の可能性がある．

3.5 PE 大域領域の圧縮

先述のとおり，PE の大域領域には PE 大域変数や PE ベクタが配置されているが，これらもごみになることがある（本節の議論では，PE 変数と PE 定数の違いは本質的ではないので，PE 定数を PE 変数とみなして差し支えない）．そこで，未使用領域がある場合は，大域領域を圧縮し，PE 大域変数や PE ベクタの新たな要求にこたえるための領域を確保することにした．図2に示したように，PE 大域変数の後に PE スタック領域が続いているので，PE 大域領域の圧縮を行うと，PE スタック領域も拡大される．

この圧縮は，TUPLE が以下のように設計されているので，ごく簡単に行うことができる．

- PE 局所メモリの同一番地のワードは，すべて同一の用途に使用されており，1つの PE 変数や同じインデックスの PE ベクタの要素を構成している．
- 各 PE 変数は ACU 記号セル以外から参照されることはない．ACU 記号セルがごみになると，PE 変数もごみになる．PE 変数の再配置を行った場合は，ACU 記号セル内のポインタの置き換えをするだけでよい．
- 各 PE ベクタも ACU 内のヘッダセル以外からは参照されない．したがって，ヘッダセルがごみになると，PE ベクタもごみになる．PE ベクタの再配置を行った場合も，ACU ヘッダセル内のポインタの置き換えをするだけでよい．

圧縮を行うためには，PE 大域領域内の各ワードが，どの ACU セルから指されているかの記録が必要であるので，これらについての反転ポインタ表を ACU 内に用意した．この表の大きさとして，各 PE の大域領域が最大限に広がった時の大きさが必要である．現在の実装の対象となっている各 PE ごとに 16k バイトの局所メモリをもった MP-1 では，PE 大域領域の大きさの最大値は高々 512 ワードであり，ACU のデータ領域の大きさと比べても全く問題とならない．この表の更新は，スイープフェイズ中と PE 変数や PE ベクタが定義されたときのみに行えばよいので，実行効率の低下をまねくことはない．圧縮を行うと，PE 大域領域は低位アドレスの方へ縮み，PE スタック領域との間に空白が生じる．この空白領域は，TUPLE のトップレベルに制御が戻った後に，スタック領域として使われる．すなわち，現在のトップレベル式の実行が終了した時に，スタックの底を指していたポインタが，空白領域の先頭を指すようにリセットされる．

4. 評価

これまでの議論に基づいて実装したごみ集めの性能に対する関心は以下の3点であろう．

1. PE セルのマーク並列化の効果
2. FE とバックエンドの間のバッファリングの効果
3. 再帰をともなわずに PE セルのマークを行う走査法の効果

これらの効果を調べるために，今回提案したごみ集めを実装した TUPLE のほかに，比較用のごみ集め（例えば，PE セルのマークを逐次的に行うごみ集め）を実装した TUPLE を用意した．これら2つの TUPLE 上で同一のプログラムを実行して同一内容のヒープを作成した後で，ごみ集めを強制的に起動し，その実行時間を比較することによって評価を行った．この実行時間は，FE と ACU と PE におけるマークとスイープに要した時間の合計である．

PE セルを消費させるために，並列二進検索を行い，各 PE に木を構築することにする[6),8)]．二進木の節は，**図3**に示すように3つの PE セルで表現される．要素を追加する PE 関数 bs-add は，いずれの PE 木にもその要素が存在しない場合のみ，いずれか1つの PE 木にその要素を追加する．そして，ごみ集めの時は，PE 木を構成しているセルは使用中として処理

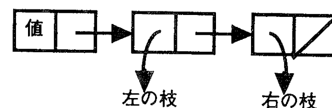


図 3 二進木の節
Fig. 3 A node of a binary search tree.

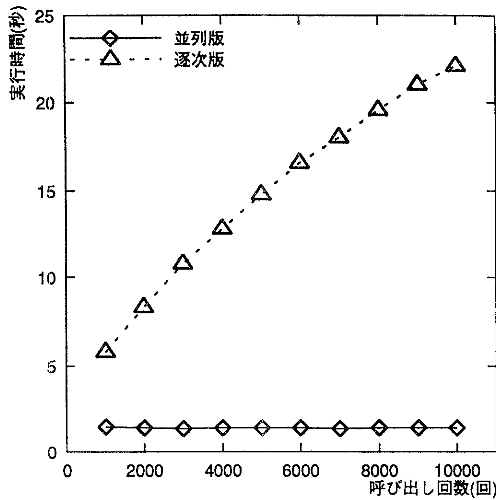


図 4 PE セルマーク並列化の効果
Fig. 4 Effects of parallel PE cell marking.

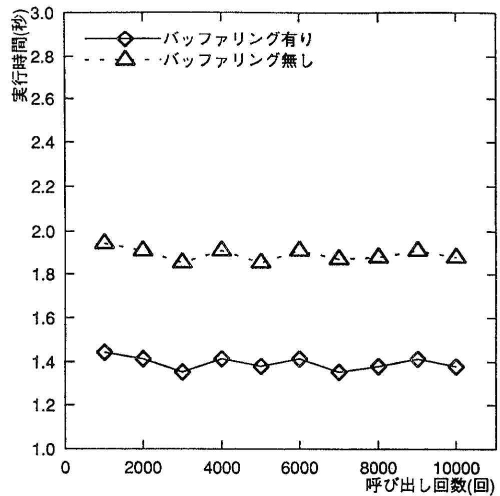


図 5 バッファリングの効果 その1 (PE 側に FE オブジェクト無し)
Fig. 5 Effects of buffering (no FE cell pointer in PEs).

される．なお，この実験は，FE として VAXstation 3520 をもち，バックエンドには 1024 台の PE を搭載した MP-1 上で行った．

4.1 PE セルのマーク並列化の効果

PE セルのマーク並列化の効果を調べるために，PE セルのマークを逐次的に行うごみ集め（逐次版）を試験的に実装し，今回実装した PE セルのマークを並列に行うごみ集め（並列版）と比較した．**図 4** に，それぞれの方式におけるごみ集めの時間を示す．横軸は PE 関数 bs-add を呼び出した回数，縦軸はごみ集めの実行に要した時間である．

並列版では，呼び出し回数が増えても実行時間がほとんど変化しないのに対し，逐次版では呼び出し回数が増えるとともに実行時間が大幅に増加している．しかも，その差は非常に大きく，呼び出し回数が 10,000 回の場合は，逐次版は並列版の約 16 倍の実行時間を要している．並列版では節の数が増加しても，1024 台の PE による並列処理によって，節の増加による実行時間の増加が 1/1024 に抑えられているからである．

4.2 FE とバックエンドの間のバッファリングの効果

FE-バッファエンド間のバッファリングの効果を調べるために，バッファリングを行わないごみ集め（バッファリング無し）を試験的に実装して，今回実装したバッファリングを行うごみ集め（バッファリング有り）と比較した時のごみ集めの時間を**図 5** に示す．

各 PE 木には FE のオブジェクトは含まれていないが，組み込みの PE 関数や，並列二進検索プログラムで定義された PE 関数による FE-ACU 間のリンクが存在している．いずれにも，要素追加回数による実行時間の変化はほとんどないが，バッファリング無しの場合は，バッファリング有りの場合の約 1.36 倍の実行時間を要している．PE-FE 間のリンクは存在しないうえ，4.1 節に述べたように PE セル数の増加は実行時間にほとんど影響を与えないことから，実行時間の差は ACU-FE 間通信の PE を介したバッファリングの効果である．そして，この差は PE 関数の増加とともにさらに拡がるものと考えられる．この時の FE-ACU 間のリンクは約 300 であり，PE 上のバッファは 1024 (>300) であるから，3.3 節の空きバッファ探しは行わないので，式(5)の第 2 項に相当する処理は行われない．実行時間の短縮は第 1 項と第 3 項によって達成されたもので，ACU-PE 間通信によるオーバーヘッドは，PE-FE 間のブロック転送の効果を損なっていないことがわかる．なお，第 2 項については，式(3)の通信時間短縮率の第 2 項と密接な関係があるので，この後の議論を参照されたい．

PE-FE 間の通信のバッファリングの効果を調べるために，並列二進検索のプログラムを変更し，節を表現しているリストの長さを 1 つ増やして，このリストの最後に FE の記号を意図的に追加した．二進検索そのものの処理には，変更はない．同様にして，ごみ集めの実行時間の比較を行った結果を**図 6** に示す．こ

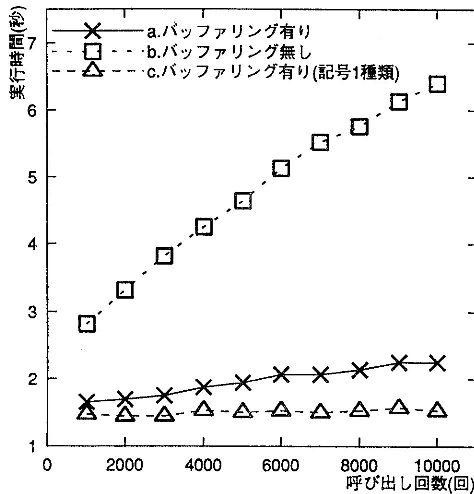


図 6　バッファリングの効果 その2 (PE 側に FE オブジェクト有り)
Fig. 6　Effects of buffering (with FE cell pointers in PEs).

の実行時間には，先ほどの ACU-FE 間の通信に要した時間も含まれている．a はバッファリング有り，b はバッファリング無しである．また，c はバッファリング有りであるが，a と b では節によって異なる記号を付加していたのに対して，すべての節に同じ記号を付加している．a, b, c ともに，生成される節の数は同じである．

a, b とも呼び出し回数の増加にともなって実行時間が増加しているが，b のバッファリング無しの方がはるかに多くの時間を要し，その増加の割合も大きい．呼び出し回数が 10,000 回の場合には，約 2.6 倍の開きがあり，バッファリングの効果が大きいことがわかる．FE と PE の間の通信パケットには，データのほかに，通信制御のための情報なども含まれており，1 通信あたりのコストは比較的高い．積極的にバッファリングを行い，1 回の通信でより多くのデータを含めるとともに，通信回数の抑制につとめるべきである．また，b と c を比較すると，b は呼び出し回数とともに実行時間がわずかながら増加しているが，c はほとんど変化していない．c においては，並列ソーティングを用いた重複ポインタの削除が功を奏しているものと思われるが，PE 間通信と並列ソーティングが効率よく行われなければこのような効果は現れなかったであろう．もちろん，並列ソーティングの前に行われる序数割り付けによる空きバッファ探しも，効率よく行われていることはいうまでもない（そうでなければ，他の PE のバッファを利用をせずに，FE へ直接マーキングする方が効率がよい）．したがって，式 (3) の通信時間短縮率の第 2 項に対応する空きバッファ探しのオーバーヘッドは，第 1 項の PE-FE 間のブロック転送の効果を損なっていないと考えられる．

4.3　再帰を行わない走査の効果

走査法の効果を調べるために，再帰をともなう典型的なマークを行うごみ集め（再帰版）を PE に実装して，今回実装したごみ集め（走査版）と比較した．その結果を**図 7** に示す．a が再帰版，b は走査版である．c も再帰版，d も走査版であるが，プロセッサ番号が 0 である PE のみをアクティブにした．a は呼び出し回数の増加とともに，実行時間が増加しているのに対して，b はほとんど変化していない．PE 木が成長していく過程で各 PE 木の形に差が生じるので，a の再帰版では木の根からたどっていって木の形が一致しないところから処理の分岐が生じ，アクティビティが不揃いになり効率の低下を招いているようである．

この考察は，a と c，b と d を比較すればより明確となる．PE 関数 bs-add は各 PE 木の高さのばらつきを抑えるように設計されており，各 PE 木に含まれている節の数に大きな差はないはずであるから，アクティビティに不揃いがなければ，1つの PE 木のマークに要する時間と，1024 個すべての PE 木のマークに要する時間は，ほぼ同じになるはずである．c は再帰版であるが，a のような実行時間の増加はみられない．おそらく，c は再帰版の最高性能であって，アク

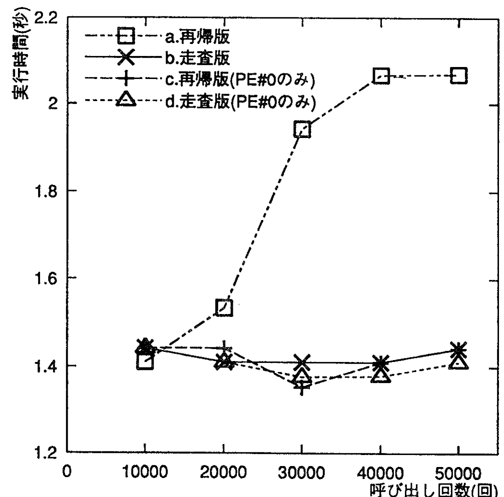


図 7　再帰を行わない走査法の効果
Fig. 7　Effects of parallel scan.

ティビティのばらつきにより効率が落ちて a となったのであろう．一方，b は d とほぼ同じであり，アクティビティのばらつきによる効率の低下はごくわずかであり，走査版が優れていることを示している．セルを木の枝に沿って連続的にたどらないことが，良い結果につながっているのであろう．

なお，呼び出し回数が 40,000 回あたりで，実行時間が頭打ちになっているのは，数値を追加しようとしても，すでにその数値が木に登録されていることが多くなり，飽和しているからである．乱数の発生範囲として，0 から 10,000 を指定したので，飽和している状態では，1024 個の PE を実装した MP-1 では 1 つの PE あたり約 10 個の数値が登録される．1 つの PE に高々 10 個の数値を登録した状態で，このような差がみられたのであるから，より多くの数値を登録して木を成長させた場合は，再帰版と走査版の性能の差がさらに拡大することは間違いない．ここには示さないが，意図的に各 PE 木の構造にばらつきを生じさせて実験した時には，この差がさらに大きくなる結果が実際に得られている．

5. まとめ

SIMD 型超並列計算機を対象とした拡張 Common Lisp 言語処理系 TUPLE のごみ集めについて報告した．並列ソーティングを併用したバッファリングにより，FE とバックエンドの間の通信のオーバーヘッドを抑制した．さらに，PE ヒープの走査により，スタックの消費とアクティビティのばらつきを抑えたうえで並列に PE 側のマークを行う手法を提案し，十分な成果を得ることができた．

PE セルのマークをヒープ領域を繰り返し走査することによって行うという手法は，走査方向と逆向きのポインタが多く存在する場合は，繰り返し回数が増大するという欠点をもつ．対案として，反転ポインタ法や tail-recursive 法（典型的な再帰法を，明示的なスタックを用いて，ループに展開したもの）が考えられる．しかし，反転ポインタ法は，前進・後退・切替の 3 つのフェイズから構成されるので複雑になり，再帰法ほどではないがアクティビティのばらつきが本質的に生じやすく，性能は期待できない．一方，tail-recursive 法はスタックの消費量の見積りが難しい．さらに，他の PE セルへのポインタ処理に伴う同期は，今回の実装では，そのセルが配置されている PE 内の要求ビットをオンにする 1 回の通信でよいが，両者ともこう簡単にはできないであろう．また，副次的な効果であるが，要求ビットやマークビットの重複処理について，特に考慮する必要もない．

しかしながら，繰り返し数増加に対して，全く対策を講じていないわけではない．マークビットと要求ビットを PE セルと分離して，連続した領域に配置しているので，1 ワードが 32 ビットの MP-1 では，32 個の PE セルの処理を 1 回の整数演算で行うことができる．例えば，要求ビットの有無や有効な要求ビットを得る操作（要求ビットとマークビットの否定の論理積）を，32 個分まとめて一度に行う．繰り返し数が増加するというのは，1 回の繰り返しあたりオンになる要求ビットの数が少ない場合であるので，32 個単位で要求ビットがオンになっていないことを確認して通過してしまうことが多く，1 回の繰り返しに要する実行時間を短縮することができる．さらに，要求ビットやマークビットをセルと分離して配置しているので，要求ビットがオンにならないセルについては，セル本体へのアクセスは不要であることから，繰り返しのコストが著しく高いとは考えられない．

今回のごみ集めの実装で，逐次実行で数多く採用されているアルゴリズムであっても，SIMD アーキテクチャのもとでは，必ずしもよい結果が得られないことがわかった．一見効率の悪いアルゴリズムであっても，単純な操作の繰り返しによってアクティビティの不揃いを抑え，高い性能をもたらす例を示した．

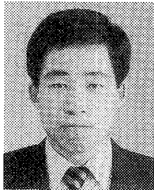
謝辞　筆者らとともに，TUPLE のごみ集めの実装を行った岡澤隆志氏に感謝の意を表す．また，有益なコメントを下さった査読者の方々に深謝する．本研究は，住友金属工業株式会社と Digital Equipment Corporation の援助をうけている．

参 考 文 献

1) 岡澤隆志：SIMD 型超並列計算機における拡張 Common Lisp の設計と実現，豊橋技術科学大学大学院修士学位論文 (1992).
2) Quinn, M.: *Designing Efficient Algorithms for Parallel Computers*, McGraw-Hill (1987).
3) Steele, G.: *Common Lisp the Language*, Digital Press (1984).
4) Yuasa, T.: Design and Implementation of Kyoto Common Lisp, *J. Inf. Process.*, Vol. 13, No. 3, pp. 284-295 (1990).
5) Yuasa, T.: *TUPLE—An Extension of KCL for Massively Parallel SIMD Architecture—*, Draft for the Second Edition, available from the author (1992).

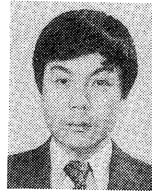
6) Yuasa, T.: TUPLE: An Extended Common Lisp for Massively Parallel SIMD Architecture, *Proceedings of the DPRI Symposium*, Boston (1992).
7) 湯淺太一，安本太一：KCL における即値データの実装とその評価，電子情報通信学会春季全国大会講論集，D-357 (1987).
8) 湯淺太一，安本太一，永野佳孝，畑中勝実：TUPLE：SIMD 型超並列計算のための拡張 Common Lisp，情報処理学会論文誌，Vol. 35, No. 11, pp. 2392-2402 (1994).
9) *MasPar Parallel Application Language (MPL) User Guide*, MasPar Computer Corporation (1991).

(平成 6 年 2 月 16 日受付)
(平成 6 年 9 月 6 日採録)



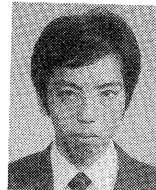
安本 太一（正会員）

1966 年生．1986 年詫間電波工業高等専門学校電子工学科卒業．1988 年豊橋技術科学大学情報工学課程卒業．1990 年同大学大学院修士課程情報工学専攻修了．同年愛知教育大学数理科学選修助手となり現在に至る．プログラミング言語処理系，並列計算機に興味を持つ．日本ソフトウェア科学会，電子情報通信学会各会員．



湯淺 太一（正会員）

1952 年神戸生．1977 年京都大学理学部卒業．1982 年同大学理学研究科博士課程修了．同年京都大学数理解析研究所助手．1987 年豊橋技術科学大学講師．現在，同大学助教授．理学博士．記号処理と超並列計算に興味を持っている．著書「Common Lisp 入門（共著）」ほか．ソフトウェア科学会，電子情報通信学会，IEEE, ACM 各会員．



貴島 寿郎（正会員）

1966 年生．1987 年佐世保工業高等専門学校電気工学科卒業．1989 年豊橋技術科学大学情報工学課程卒業．1991 年同大学大学院修士課程情報工学専攻修了．現在，同博士後期課程在学中．超並列計算機，プログラミング言語，言語処理系などに興味をもっている．