Vol. 35 No. 11 情報処理学会論文誌 Nov. 1994

TUPLE: SIMD 型超並列計算のための拡張 Common Lisp

湯淺太一[†] 安本太一[††]
永野佳孝[†] 畑中勝実[†]

SIMD (Single Instruction, Multiple Data) 型超並列計算機上で動作する拡張 Common Lisp 言語および処理系である TUPLE について報告する．TUPLE は SIMD 型超並列計算の機能を Common Lisp に付加したものである．従来の SIMD 型計算機用の Lisp 言語とは異なり，膨大な数の Common Lisp サブセットの処理系が並列に動作するという計算モデルを TUPLE は採用している．この目的のために，ターゲット・マシンの各 PE (Processing Element) は，固有のヒープをその局所メモリに保有する．これらのサブセット処理系を，フルセットの Common Lisp 処理系が制御し，ユーザは，フルセット処理系を使って並列プログラムの開発と実行を行う．本稿は，TUPLE の言語と処理系の概要を紹介するとともに，1024台以上の PE を有する SIMD 型超並列計算機 MasPar MP-1 における TUPLE の実現について触れ，その性能評価結果を報告する．

TUPLE: An Extended Common Lisp for Massively Parallel SIMD Architectures

Taiichi Yuasa,[†] Taichi Yasumoto,[††] Yoshitaka Nagano[†]
and Katsumi Hatanaka[†]

An extended Common Lisp language and system, called TUPLE, for massively parallel SIMD (Single Instruction, Multiple Data) architectures is presented. Unlike other Lisp languages on SIMD architectures, TUPLE supports the programming model that there are a huge number of subset Common Lisp systems running in parallel. For this purpose, each PE (Processing Element) of the target machine has its own heap in its local memory. In addition, there is a full-set Common Lisp system with which the user interacts to develop and execute parallel programs. This paper briefly introduces the TUPLE language and system, and then reports the current implementation and its performance measurements of TUPLE on the SIMD machine MasPar MP-1 with at least 1024 PEs.

1. はじめに

計算機の処理能力を飛躍的に向上させるために，数千から数十万といった膨大な台数の PE (Processing Element) を並列動作させる，いわゆる超並列計算機の開発と実用化の研究が進められている．いくつかの超並列計算方式が提案されているが，SIMD (Single Instruction, Multiple Data) 方式は，その応用範囲に制約があるものの，もっとも実用に近いものの１つであろう．実際，いくつかの商用 SIMD 型超並列計算機[3]がすでに利用され始めている．

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

従来，SIMD 型超並列計算機は数値計算や数値解析の分野で主に利用されており，C 言語や Fortran を拡張したプログラミング言語を使用することがほとんどであった．一方で，Lisp などを使用することの多いリスト処理，記号処理分野においても，計算処理能力向上のために並列化が強く望まれている．

本稿で紹介する TUPLE (Toyohashi University Parallel Lisp Environment) は，SIMD 型超並列計算機における記号処理分野の応用プログラム開発のための Lisp 言語およびその処理系である．高性能の処理系を提供することによって，記号処理分野における SIMD 型超並列計算の有効性を検証するとともに，超並列記号処理アルゴリズムの開発の手段となることを目的とする．

これまでに，いくつかの SIMD 型超並列計算機用 Lisp 言語および処理系が開発されている[4),6),7),9),13)~

[16]. これらの Lisp 言語は，既存の逐次型 Lisp 言語に，並列処理のためのデータ型（例えば，Connection Machine Lisp における **xapping**）を追加することによって並列処理を可能にしている．すなわち，フロント・エンド計算機（FE）が全体の制御を行い，PE は拡張されたデータ型の上の演算を並列実行する，という計算モデルを採用している．これに対して TUPLE は，全く異なった計算モデルを採用することにより，柔軟でしかも効率のよいプログラムの記述を可能にしている．

SIMD 型並列計算機では，PE は独自の命令ストリームを持たず，FE がブロードキャストする命令を（なんらかの条件を満たす）すべての PE が実行する．この点を除けば，個々の PE は，通常の単体プロセッサと同様のものと考えてよい．そこで TUPLE では，個々の PE に Lisp 処理系の機能を持たせることを試みた．ただし，現在の SIMD 型超並列計算機では，個々の PE の局所メモリのサイズは，PE あたり数キロバイトから数十キロバイト程度と小さく，各 PE 上の処理系は，記号処理・リスト処理が可能な必要最低限のものに限られる．一方，FE は UNIX ワークステーションといった通常の汎用マシンが普通であり，近代的な Lisp 処理系を実現するのに十分な能力を備えている．

以上のことから，TUPLE の計算モデルは次のようなものとした．物理的な PE ごとに **PE サブシステム**とよばれる Lisp 処理系が1つずつ配置され，それらが SIMD 的に動作する．さらに，これらの PE サブシステムを制御する **FE システム**というフルセットの Common Lisp[8] 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．このモデルを採用することにより，

- 通常の逐次型 Lisp 言語と同様の記法を使って並列プログラムの大部分を記述することが可能となる．
- SIMD アーキテクチャを直接反映したものであるために，応用プログラムの性能が容易に推測でき，プログラムのチューンアップが容易である．
- FE システムの提供する高機能のプログラミングツールを利用でき，効率の良い対話型超並列プログラミング環境が可能となる．

などのメリットが得られる．

本稿では，まず次章で TUPLE の言語と処理系の概要を紹介し，第3章で TUPLE による並列リスト処理の例をあげる．第4章では TUPLE の全体像を明確にし，第5章で MP-1 における実現を報告し，最後に第6章で TUPLE 処理系の性能評価を行う．

2. TUPLE の概要

本章では，TUPLE の言語と処理系の概要を，簡単なプログラム例をあげて紹介する．例として，絶対値を求める関数 **abs** をあげる．Common Lisp ではこの関数は次のように定義できるであろう．

```
(defun abs (x)
  (if (>= x 0) x (- x)))
```

すなわち，引数が0以上であれば引数をそのまま返し，そうでなければ引数の符号を反転したものを返す．**defun** を **defpefun** に置き換えることによって，PE サブシステムにおいて同様な動作をする関数が定義される．

```
(defpefun abs (x)
  (if (>= x 0) x (- x)))
```

この **PE 関数**が呼び出されると，すべての PE サブシステムは，独立した引数を受け取り，0以上の引数を受け取った PE サブシステムは引数をそのまま返し，その他の PE サブシステムは引数の符号を反転したものを返す．

TUPLE は SIMD アーキテクチャ上で動作するので，すべての PE は常に同じ命令を実行し，PE によって異なる命令を実行することはない．PE 関数 **abs** が起動されると，条件を満たさない PE はインアクティブになり，条件を満たす PE のみが *then* 節を評価する．その後，PE のアクティビティが反転し，今度はインアクティブであった PE がアクティブとなって *else* 節を評価し，はじめにアクティブであった PE はその間インアクティブになる．

ユーザと TUPLE 処理系の実際の対話例を次に示す．TUPLE のトップレベルは通常の Common Lisp 処理系と同様であり，ユーザは任意の Common Lisp の式を入力することができる．入力された式は逐次的に評価され，その結果が表示される．

```
% tuple
TUPLE (Massively Parallel KCL)

>(defun abs (x)
    (if (>= x 0) x (- x)))
ABS
>(abs -3)
3
```

並列計算を開始するには，TUPLE言語において拡張された式を用いる．

```
〉(defpefun abs (x)
    (if (〉= x 0) x (- x)))
ABS
〉(ppe penumber)
#P(0 1 2 3…)
〉(ppe (abs (- penumber 2)))
#P(2 1 0 1…)
```

この例では，ユーザは，**ppe** 式を用いて，PE サブシステムに **PE式**を送って評価させ，その返り値を表示させている．**ppe** 式は主に TUPLE のトップレベルにおいて，トップレベルフォーム (top-level form) を PE サブシステムに渡すために用いられる．

上の例における **penumber** は，PE サブシステムの組込み定数であり，各 PE のプロセッサ番号（最初の PE は **0**，次の PE は **1**，等々）を保持している．2番目の **ppe** 式は，PE 関数 **abs** を用いて **penumber** −2の絶対値を求めており，例えば最初の PE は **2** を返す．

TUPLE においては，PE 関数の名前空間は，通常の逐次関数の名前空間とは異なるものである．上の例では，**abs** という名前が，通常の逐次関数と PE 関数の双方で使用されている．これは，PE 関数は PE サブシステムに，逐次関数はユーザが対話を行うフルセット Common Lisp 処理系上に定義されるためである．同様な動作をする関数を，同じ名前で双方のシステムに定義できるために，逐次プログラムの並列化作業の負担を大幅に軽減できる．

上の例では，**ppe** 式は値を返しているかのようにみえるが，実は PE サブシステムが返した値を表示しているだけであり何も返していない．TUPLE では，いわゆる“並列データ”というものは存在せず，**ppe** 式は値をまったく返さない式である (Common Lisp では，値をまったく返さない関数を定義できる)．PE サブシステム側の値を，FE 側で得るためには，**リダクション** (reduction，各 PE サブシステムが持っているデータを集め，処理をほどこして1つのデータにまとめる操作) を行う．**表1**に，TUPLE の提供するリダクション操作の一覧をあげる．例えば，**reduce-max** はアクティブな PE が保持する値の最大値を求める．**reduce** は汎用のリダクションであり，リダクションに使用する関数を指定できる．例えば，

```
(reduce #'max x)
```

は，

```
(reduce-max x)
```

と等価である．

3. 並列リスト処理

本章では，TUPLE における並列リスト処理の典型的な例として，二進検索 (binary search) の並列化をあげる．Lisp では，二進検索木の各節は3つの要素をもつリストとして，**図1**のように表現することができる．最初の要素はその節の値，2番目と3番目の要素はそれぞれ左と右の枝を表す．ここでは，各節には数値が格納されるものとする．ある節の値を n とすると，その節の左の枝から到達できる節の値は n より小さく，右の枝から到達できる節の値は n より大きい．通常，二進検索関数は，目標の数値と節が保持する数値との比較によって探索方向を決定しながら，与えられた木を再帰的にたどるように定義され，節の数 N に対して $\log N$ の時間で検索することができる．Common Lisp における二進検索関数の定義例を以下に示す．

```
(defun binary-search (node item)
  (if (null node)
      nil
      (if (= (car node) item)
          t
          (binary-search
            (if (< item (car node))
                (cadr node)
                (caddr node))
            item))))
```

二進検索を並列に行うため，**図2**のように，二進検索木全体を各 PE に1個ずつ配置した **PE 部分木**に

表1 リダクション
Table 1 Reductions.

some-pe	some-penumber	some-pevalue
every-pe	not-any-pe	not-every-pe
reduce-max	reduce-min	reduce-char-max
reduce-char-min	reduce-logand	reduce-logior
reduce-logxor	reduce-+	reduce-*
reduce		

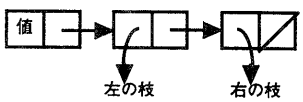


図1 二進検索木の節
Fig. 1 A node of a binary search tree.

分割することにする．各 PE の PE 部分木も二進検索木であり，その PE サブシステムの cons セル (**PE cons**) を使って構成されるものとする．これらの PE 部分木に重複する値を持った節がなければ，PE 部分木の集まり全体を1つの大きな二進検索木とみなすことができる．PE 部分木を構築するには，追加したいデータごとに適切な PE を選択し，その PE の持つ PE 部分木の適切な位置にそのデータを値とする節を追加する．その詳細については後述する．

二進検索関数の並列版は次のように定義できる．

```
(defpefun binary-search (node item)
  (if (null node)
    nil
    (exif (= (car node) item)
          t
          (binary-search
            (if (< item (car node))
                (cadr node)
                (caddr node))
            item))))
```

これは Common Lisp で記述した逐次版とほとんど同じである．しかし並列版では，ある PE がその PE 部分木の中に目標の数値をみつけたら，他の PE はそれ以上検索を続行する必要がない．目標の数値をみつけた PE があれば，全体の計算をただちに終了すべきである．このような“プロセッサ間同期”は，多くの SIMD 型並列アルゴリズムに要求されるものであり，このための新しい構文として **exif** (**exclusive if**) を導入した．**exif** 式

(exif 《条件》《*then* 節》《*else* 節》**)**

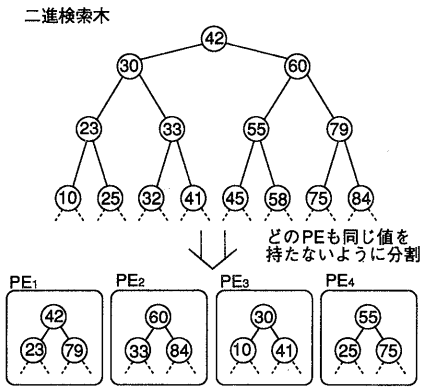


図 2 PE 部分木 (PE 数が4の時)
Fig. 2 PE subtrees (for 4 PEs).

は，**if** 式と同様の構文であるが，ある PE が《条件》を満たした場合，他の PE は《*else* 節》を評価しないで単に **nil** を返すところが異なる．上記の並列二進検索関数では，ある PE の現在注目している節の値が目標の数値と一致すれば，その PE は真を表す **t** を返し，その他の PE は偽を表す **nil** を返してただちに終了する．

次に PE 部分木の構築について述べる．二進検索を効率よく行うためには，木の左右のバランスを保つ必要がある．並列二進検索においては，個々の PE 部分木のバランスを保つと同時に，すべての PE 部分木の深さのバランスを保つことが要求される．新たな数値を追加する時には，検索時に最初に葉に達する PE を選ぶのが最善の方法であろう．この方法にもとづいて，ある数値をいずれかの PE 部分木に追加する PE 関数 **bs-add** を定義する．

```
(defpefun bs-add (place item)
  (cond ((some-pe (null (car place)))
         (exif (binary-search (car place)
                              item)
               nil
               (when (= (some-penumber
                          (null
                            (car place)))
                        penumber)
                 (rplaca place
                   (list item nil nil)))))
        ((some-pe (= (caar place) item))
         nil)
        (t (bs-add
             (if (> (caar place) item)
                 (cdar place)
                 (cddar place))
             item))))
```

この関数は，次のように動作する．

1. 葉に達した PE があれば，残りの PE は単純に検索を続ける．
   1.1. もし，いずれかの PE が目標の数値をみつければ，すべての PE が偽を返して終了する．
   1.2. さもなければ，最初に葉に到達した PE のうちの任意の1つを選んで，そこに数値を追加する．
2. もし，いずれかの PE が目標の数値をみつけれ

ば，すべての PE が偽を返して終了する．

3. 上記以外の場合は，すべての PE が PE 部分木を再帰的にたどる．

bs-add には，現在注目している節へのポインタを保持する“場所”を引数として与える．それぞれの“場所”は PE cons であり，その car 部が現在の節へのポインタを保持している．**some-pe** と **some-penumber** はリダクション関数である．**some-pe** は，1台でも PE が引数として真を受け取れば，すべての PE が真を返す．**some-penumber** は，真を受け取った PE のプロセッサ番号をすべての PE が返す．真を受け取った PE が複数の時は，適当なプロセッサ番号を選択して，返す値として用いる．2行目の

(some-pe (null (car place)))

は，葉に達した PE が存在するかどうかを調べている．また，**exif** の *else* 節では，**some-penumber** を使って，葉に到達した PE のうちの1つを選択している．

初期状態においては，各 PE の持つ PE 部分木には節が1つもない．この初期化は，トップレベルで次の式を評価することによって行う．

```
〉(defpevar pe-tree (list nil))
```

pe-tree という名前の変数（**PE 変数**）を各 PE サブシステムに配置し，初期値として空木（つまり **nil**）を保持する PE cons を与える．PE 関数 **bs-add** を呼び出すことによって，指定された数値がいずれかの PE 部分木に登録される．その例を次に示す．

```
〉(ppe pe-tree)
#P((NIL) (NIL) (NIL) (NIL) …)
〉(ppe (bs-add pe-tree 503))
#P(T NIL NIL NIL…)
〉(ppe pe-tree)
#P(((503 NIL NIL)) (NIL) (NIL) (NIL) …)
…
```

PE 関数 **bs-add** を繰り返し呼び出すことによって，PE の局所メモリに PE 部分木を構築してゆくことができる．

4. TUPLE 言語の仕様

本章では，TUPLE 言語の仕様について簡単に述べる．前述のように TUPLE 言語は Common Lisp 言語を拡張したものであるから，Common Lisp のすべての機能をサポートしている．したがって，TUPLE 処理系は，逐次的ではあるが，Common Lisp のプログラムをすべて実行できる．

4.1 データオブジェクト

Common Lisp 言語において定義されているデータオブジェクトは，TUPLE 言語においてもすべて定義されている．さらに，TUPLE 言語は，並列計算のために，以下のデータオブジェクトをサポートしている．

- PE cons
  PE サブシステムに配置される cons オブジェクトである．前章の並列二進検索の例では，これを用いて二進検索木を構築した．PE cons は，cons と同様に，その car 部と cdr 部に，TUPLE の任意のオブジェクトを格納することができる．
- PE ベクタ
  PE サブシステムに配置される一次元の配列である．Common Lisp のベクタと同様に，PE ベクタは，すべてのデータを要素として格納できる．
- PE 関数オブジェクト
  通常の関数と同様に，第一級のオブジェクト (first-class object) である．したがって，TUPLE のプログラムでは，呼び出される関数を動的に指定することが可能である．ただし，SIMD アーキテクチャ上で動作するので，同時に1つの関数しか呼び出すことができない．

FE の変数と PE 変数はすべての TUPLE オブジェクトを格納することができる．すなわち，FE の変数は PE cons, PE ベクタ，PE 関数オブジェクトを格納可能であり，PE 変数も Common Lisp のすべてのデータを格納可能である．

多くの Common Lisp システムにみられるように，TUPLE では，固定長短形式の整数 (fixnum)，単精度浮動小数点数 (short-float)，文字 (character) を，即値データ (immediate data) として実装している．これらのデータ型についての演算は，並列に行われる．一方，記号など他の Common Lisp のデータ型については，FE ヒープ内のデータセルへのポインタとして表現される．SIMD アーキテクチャでは，複数の PE が FE メモリの異なる番地を同時に参照することはできない．したがって，PE サブシステムから FE のヒープへの参照は，必然的に逐次実行にならざるをえない．FE のデータに対して並列実行が可能なのは，ポインタの比較を行う **eq** のみである．

4.2 PE 式

TUPLE のトップレベルに入力される式は，**FE 式**とよばれる．すべての Common Lisp の式は FE 式

である．FE 式は，並列計算のための式がみつかるまで，FE で評価される．もし，FE 式が並列計算のための式を含んでいなければ，TUPLE 処理系は，Common Lisp 処理系と全く同じように動作する．

2章で，並列計算を行うための

(ppe《式》**)**

を紹介したが，この《式》は通常の式（FE 式）ではなく，PE サブシステムで並列に実行される式であり，**PE 式**とよぶ．式は通常 TUPLE のトップレベルで入力されるので，PE 式は FE ヒープ上のオブジェクトによって表現される．

PE 式には次のものがある．

- リスト式
  - —スペシャルフォーム
  - —マクロ式
  - —関数呼び出しの式
- 変数
- 式自身が式の値であるもの

この PE 式の分類は，FE 式（Common Lisp の式）と全く同じである．

TUPLE は，各 PE システムが，FE の Common Lisp システムに可能な限り近いものになるように設計されている．PE サブシステムで使用可能なスペシャルフォームのうち，Common Lisp に対応するものを**表 2** にあげる．各スペシャルフォームの構文は，同名の Common Lisp のスペシャルフォームのそれと同じである．そして，PE サブシステムで並列に実行されることを除いて，その動作も同じである．

また，並列計算のために特別に導入したスペシャルフォームもいくつかあり，既に紹介した**exif**(exclusive if) もその1つである．そのようなスペシャルフォームはごくわずかであり，おそらく，実際の TUPLE のプログラムでは，**exif** ぐらいしか使われないであろう．いいかえると，大抵の TUPLE のプログラムは，**exif** と上に挙げた PE スペシャルフォームを用いて記述でき，逐次動作する Common Lisp のプログラムとほぼ同様に記述される．

表 2 PE 式のスペシャルフォーム
Table 2 Special PE forms.

and	function	prog1
case	if	prog2
cond	labels	progn
declare	let	psetq
do	let*	quote
do*	locally	setq
dolist	loop	unless
dotimes	macrolet	when
flet	or	

PE サブシステムのためのマクロ式は，Common Lisp と同様に，マクロ展開されてから評価される．PE 式は FE オブジェクトによって表現されているので，マクロ展開は FE システムが行う．

PE サブシステムにおける関数呼び出しの式は，最初の要素が関数名またはラムダリストであるリストである．関数への引数が PE サブシステムによって並列に評価された後，その関数は PE サブシステムによって同時に呼び出される．

Common Lisp と同様に，PE 変数は記号によって表現される．記号を評価することによって，その記号を名前とする PE 変数の値が各 PE サブシステムに同時に返される．PE サブシステムは PE 変数を共有しない．各 PE サブシステムは記号によって名前がつけられた固有の変数をもち，その変数の値が PE サブシステムに返される．

リストや記号以外のすべての TUPLE オブジェクトは，PE 式として評価されると，そのオブジェクト自身が値となり，PE サブシステムにブロードキャストされる．もし，そのオブジェクトがデータセルへのポインタとして表現されている場合は，ポインタだけがブロードキャストされ，データセル自身は PE サブシステムにはコピーされない．

5. 実現の概要

TUPLE の処理系は，フルセットの Common Lisp 処理系である KCL (Kyoto Common Lisp)[11] をベースに実現した．現在，1024 台以上の PE を備えた MasPar 社の SIMD 型超並列計算機 MP-1 上で動作している．ベースとなる KCL は C 言語と Common Lisp で記述されており，TUPLE の処理系は，MP-1 用の拡張 C 言語である MPL[17] と TUPLE 自身で記述した．このために，KCL と同様に高い移植性を有しており，他の SIMD 型，あるいは SPMD (Single Program, Multiple Data) 型超並列計算機へ容易に移植できるものと思われる．

MP-1 は，1024 台以上の PE をもつ SIMD 型並列計算機であり，FE としての UNIX ワークステーションとバックエンドの2つの部分から構成されている．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，PE が2次元

格子状に配置された PE アレイからなる．MP-1 のプログラムは，FE 関数と ACU 関数からなり，並列計算は FE 関数から ACU 関数を呼び出すことによって開始される．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 のデータメモリは 128K バイト，各 PE の局所メモリは 16K バイトである．バックエンドには仮想記憶はサポートされていない．

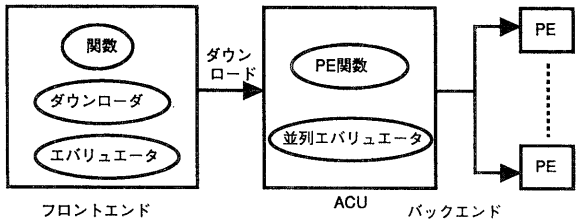


図 3 MP-1 における TUPLE 実現の概要
Fig. 3 An overview of TUPLE implementation on MP-1.

5.1 ヒープ

MP-1 では，FE とバックエンドの間の通信が非常に低速である．そこで，TUPLE は通信をできるだけ避けて高い実行効率を得るために，すべての PE 関数を ACU に配置するとともに，並列エバリュエータ (parallel evaluator) も ACU 上で動作させている (**図 3** 参照)．ユーザが新たに PE 関数を定義すると，FE の**ダウンローダ** (downloader) が，その関数定義を ACU に送信する．また，**ppe** をはじめとするいくつかの FE 式は，並列実行を行う式をまず ACU にダウンロードしておいてから並列エバリュエータに制御を渡す．この方法により，いったん FE から ACU に制御が移ると，並列計算は完全に ACU のみによって行われ，FE とバックエンドの間の通信は全く起こらない．

このことから，MP-1 における TUPLE の実現では，次の 3 種類のヒープを使用している．

- 通常の Common Lisp のデータを配置する FE ヒープ
- PE cons 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

これらのヒープ内のいずれのオブジェクトも，MP-1 のすべての構成要素 (FE, ACU, PE) から参照可能である．例えば，FE ヒープ内のデータ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 から参照することができる．また，PE cons は，リダクションによって 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．

5.2 データ表現とその配置

TUPLE におけるデータ表現を**図 4** に示す．この表現形式は，MP-1 のすべての構成要素で共通である．共通のデータ表現を採用したことにより，構成要素間の通信の際に，表現形式の変換を行う必要がない．はじめの 4 つの形式は，オリジナルの KCL でも使用しているものである．図から明らかなように，下位 2 ビットがこれら 4 つの形式を識別するために用いられている．文字データの下位から 3 ビット目は常に **0** であることに着目し，下位 3 ビットが **110** のパターンのときに，ACU と PE のヒープへのポインタを表現するようにデータ表現の拡張を行った．

各 PE のデータ領域のうち，TUPLE が直接取り扱う部分を**図 5** に示す．MPL の使用する実行時スタックは図には記載されていない．メモリ領域の先頭には，組込みの PE 定数領域に続いて，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

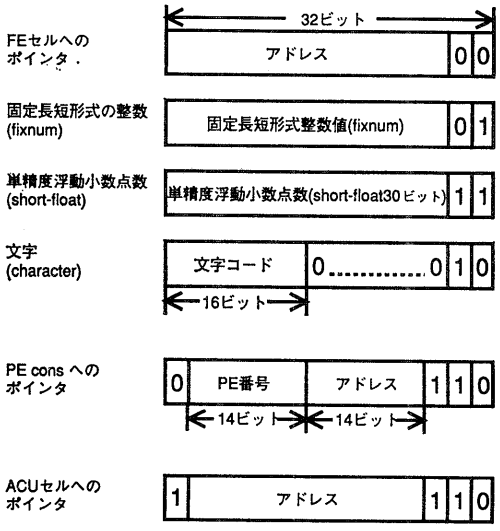


図 4 TUPLE におけるデータ表現
Fig. 4 Object representation of TUPLE.

の引数などの一時的なデータが配置される．最後に，PE cons が配置されるヒープ領域がある．TUPLE には，並列ごみ集めルーチン[10),12)]が実装されているが，これについては別の機会に報告する．

PE あたり 16 K バイトの局所メモリを持つ MP-1 では，データ領域の大きさは 8 K バイトであり，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 cons は 8 バイト（2 ワード）を占めるので，PE ごとに 512 個のセルが使用可能である．

5.3 記号セル

大域的 PE 変数と大域的 PE 関数の束縛は，**ACU 記号セル**と呼ばれる ACU セルによって表現される．ACU 記号セルは，FE の記号セルに 1 対 1 に対応しており，複数の ACU セルが同一の FE 記号セルに対応することはない．**defpevar** によって大域的な PE 変数が定義されたり，**defpefun** によって大域的な PE 関数が定義された場合に，その変数名や関数名と同一の名前を持った ACU 記号セルが存在しなければ，新しい ACU 記号セルが生成される．ダウンローダは，PE 式に現れるすべての大域的 PE 変数や大域的 PE 関数への参照を，対応する ACU 記号セルへのポインタに変換する．

各 ACU 記号セルは，大域的 PE 変数の大域領域内のアドレス（全 PE で共通），PE 関数へのポインタ（ACU セルへのポインタ），対応する FE の記号セルへのポインタを格納している．FE の記号セルへのポインタは，実行時に未定義の PE 変数や PE 関数を検出した際に，エラーメッセージを表示するためなどに用いられる．一方，FE の記号セルは，対応する ACU 記号セルがあれば，その ACU セルへのポインタを格納しており，ダウンローダが FE の記号から ACU の記号への変換を行う際などに使用される．

FE の記号セルには，対応する ACU 記号セルへのポインタのほかに，PE サブシステムのスペシャルフォームを扱う ACU ルーチンなどの情報が付加されている．しかし，記号以外の FE セルの表現形式は，ベースとなった KCL のものと全く同じである．このために，TUPLE 実現の際の KCL の変更はごくわずかであった．

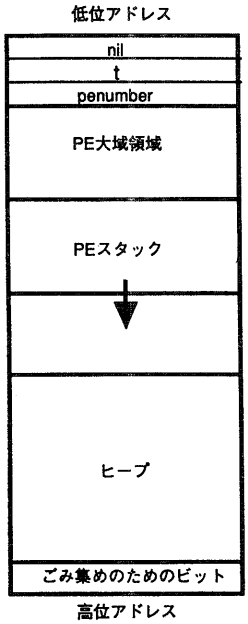


図 5 PE のデータ領域
Fig. 5 The PE data area.

6. 性能評価

ランダムな数値を引数として 3 章の PE 関数 **bs-add** を繰り返し呼び出し，その実行時間を測定した．実行時間の測定には，VAXstation 3520 を FE とし，PE を 1024 台持つ MP-1 を用いた．結果を**図 6** に示す．図からわかるように，FE 上で実行した逐次版に比べて 5 倍ないし 6 倍の速度向上しか得られていない．**bs-add** の呼び出し回数を n，PE の総数を 2^k とするとき，逐次版の実行には約 $\log_2 n$ の時間がかかるのに対し，並列版では約 $\log_2(n/2^k)$ の時間を要する（我々の経験から，FE と個々の PE の実行性能はほぼ同

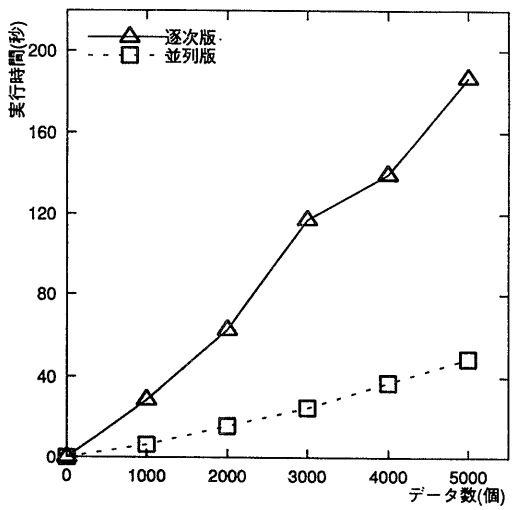


図 6 1024 台の PE を使ったデータの挿入
Fig. 6 Adding items with 1024 PEs.

等と考えてよい)．この実験では，1024 個の PE を用いたので $k=10$ であり，例えば $n=2^{12}=4096$ とすると，逐次版の実行時間は 12，並列版では 2 となり，最大で 6 倍の速度向上が得られることになる．実測値と比較すると，TUPLE の処理系は理論的限界にきわめて近い実行効率を有していることがわかる．速度向上が小さいもう 1 つの理由として，PE 部分木の拡張の際に，1 台の PE しか動作していないことがあげられる．PE 部分木の拡張操作は，PE ヒープに PE cons を割り当てるために，このプログラム中でもっとも時間を消費する部分の 1 つである．SIMD アーキテクチャでは，選択された PE がその PE 部分木の拡張を行っている間，他のすべての PE は待ち状態になる．プログラムを修正して，多くの(あるいはすべての) PE が自分の PE 部分木を拡張できる状態になるまで，PE 部分木の拡張を延期することによって，速度向上が可能である[2]．

同じプログラムを使ったもう 1 つの実験結果を**図 7** に示す．この実験では，並列計算を行う PE の台数を変化させた．プログラムが PE の個数とは無関係であることに注意されたい．1 台の PE のみを使って "並列" プログラムを実行したとき，逐次版より実行時間が余計にかかっているが，あまり大きな差ではない．これは TUPLE が個々の PE に対して効率よく実現されていることを意味している．図中，実行時間は PE の数の増加に対して着実に減少しているが，その実行時間は収束するように見える．収束点はプログラムの本質的に逐次的な部分の処理時間を指し示している．

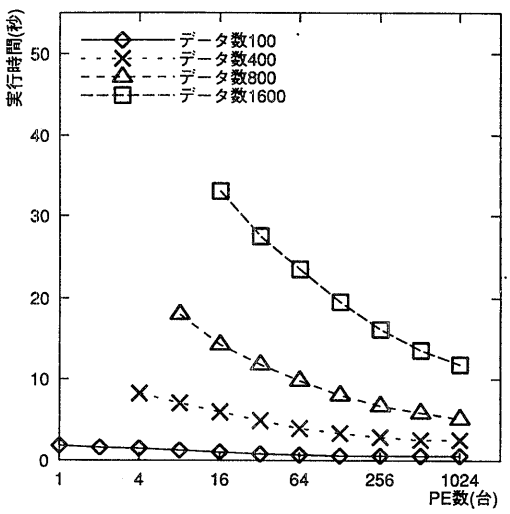


図 7 PE 数を変化させたときのデータの挿入
Fig. 7 Adding items with various numbers of PEs.

表 3 流体解析プログラムの実行結果
(200 回の繰り返し)
Table 3 Time for 200 iterations for the fluid field program.

C 版	
VAXstation 3520	108.8 秒
SPARCstation 1+	32.9 秒
MPL 版	3.0 秒
KCL 版 (インタープリタ)	
VAXstation 3520	1700.4 秒 (GC 49 回)
SPARCstation 1+	390.3 秒 (GC 49 回)
KCL 版 (コンパイルコード)	
SPARCstation 1+	44.4 秒 (GC なし)
TUPLE 版	8.0 秒 (GC なし)

より一般的な SIMD 向きの応用における TUPLE システムの性能を知るために，Navier-Stokes 方程式を解くことによって，流体運動の解析を行うプログラムを作成した．流路全体を格子状の小領域に分割し，各小領域の値を更新するループを繰り返すことによって計算を行う典型的な手法 (例えば文献 1) を参照されたい) を採用した．その測定結果を**表 3** に示す．

現時点では TUPLE にはコンパイラがなく，プログラムはインタープリタで実行されている．インタープリタによる逐次版の実行結果と比較すれば，並列版がはるかに速いことがわかる．SPARCstation 1+ 上で動作するコンパイルされた逐次版と比較しても並列版は 5 倍程度速い．TUPLE で記述したプログラムが，MPL で記述した等価なプログラムより遅くなるのは明らかであるが，この TUPLE プログラムの実行時間は MPL 版よりそれほど遅いわけではない．

7. ま と め

本稿では，SIMD アーキテクチャのための TUPLE 言語と，SIMD 型並列計算機 Maspar MP-1 における現在の TUPLE 処理系の実装について述べた．そして，TUPLE は，典型的な SIMD 用アプリケーションにおいて，高い性能を持つことを示した．

現在，TUPLE において数式処理や項書換え系のような記号処理分野における並列アルゴリズムの開発が行われている．また，TUPLE は教育にも使用されており，その対話的な並列実行環境は，初めて並列処理を学ぶ学生の演習に最適である．MP-1 は 1 台しか設置されていないが，UNIX ワークステーション上で動作する TUPLE シミュレータが開発されている．この

シミュレータ上でプログラムの開発を行い，速度向上を評価する時にのみ実機を利用することによって，限られた計算機資源を有効利用している．

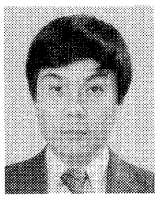
TUPLE から得た我々の経験によると，PE あたり 8K バイトのヒープは，実際のアプリケーションを動作させるにはあまりに小さい．これは，TUPLE における並列プログラムの開発の大きな障害となっている．1024 個の PE を備えた MP-1 では総メモリ量は 8M バイトとなり，大抵の場合十分であるように思われるが，データを各 PE に分散させることは現実には非常にむずかしい．データの適正配置という点では，並列二進木検索の例は成功した数少ないものの 1 つである．大容量の局所メモリを備えた SIMD 型並列計算機の出現によって，このような本質的でない問題が解決されることを期待する．

謝辞 TUPLE のプロトタイプバージョンのバックエンド部分を実現した岡澤隆志氏，SIMD 用 C 言語の設計・実現の経験を基に，ごみ集め設計にあたって有益な提案をしてくれた貴島寿郎氏，これらの方々に感謝の意を表す．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 の援助を受けている．

参 考 文 献

1) Grosch, C.: Adapting a Navier-Stokes Code to the ICL-DAP, *SIAM J. Sci. Stat. Comput.*, Vol. 8, No. 1, pp. 96-117 (1987).
2) Halstead, R.: private communication with the author (1992).
3) 喜連川優，湯淺太一：SIMD 型商用超並列コンピュータとその応用，情報処理，Vol. 32, No. 4, pp. 52-64 (1991).
4) Merrall, S. C. and Padget, J.: Collections and Garbage Collection, Proceedings of the International Workshop IWMM 92, *Lecture Notes in Computer Science 637*, pp. 473-489, Springer-Verlag (1992).
5) Quinn, M.: *Designing Efficient Algorithms for Parallel Computers*, McGraw-Hill (1987).
6) Sabot, G.: Introduction to Paralation Lisp, *Technical Report*, PL 87-1, Thinking Machines Corporation (1987).
7) Sabot, G.: *The Paralation Model: Architecture Independent Parallel Programming*, MIT Press (1988).
8) Steele, G.: *Common Lisp the Language*, Digital Press (1984).
9) Steele, G. and Hillis, D.: Connection Machine Lisp: Fine-Grained Parallel Symbolic Processing, *Proc. 1986 ACM Conf. on Lisp and Functional Programming*, pp. 279-297 (1986).
10) 安本太一，湯淺太一，貴島寿郎：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，情報処理学会論文誌，Vol. 35, No. 12 (1994) 掲載予定.
11) Yuasa, T.: Design and Implementation of Kyoto Common Lisp, *J. Inf. Process.*, Vol. 13, No. 3, pp. 284-295 (1990).
12) Yuasa, T.: Memory Management and Garbage Collection of an Extended Common Lisp System for Massively Parallel SIMD Architecture, Proceedings of the International Workshop IWMM 92, *Lecture Notes in Computer Science 637*, pp. 490-506, Springer-Verlag (1992).
13) Wholey, S. and Steele, G.: Connection Machine Lisp: A Dialect of Common Lisp for Data Parallel Programming, *Proc. Second International Conf. on Supercomputing*, pp. 45-54 (1987).
14) *Connection Machine Lisp Reference Manual*, Thinking Machines Corporation (1987).
15) Introduction to Data Level Parallelism, *Technical Report*, PR 86-14, Thinking Machines Corporation (1986).
16) **Lisp Reference Manual*, Thinking Machines Corporation (1988).
17) *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 7 月 14 日採録)

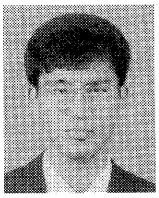


湯淺 太一（正会員）

1952 年神戸生．1977 年京都大学理学部卒業．1979 年同大学理学部修士課程修了．1982 年同大学博士課程修了．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．1988 年同大学助教授となり現在に至る．理学博士．記号処理システムと並列処理に興味を持っている．著者「Common Lisp 入門（共著）」ほか．訳書「プログラミング言語 Turing(共訳)」ほか．

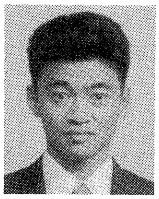
安本 太一（正会員）

1966 年生．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．



永野　佳孝

1967年生．1991年豊橋技術科学大学情報工学課程卒業．1993年同大学大学院修士課程情報工学専攻修了．同年NTN(株)入社．現在，画像処理装置の開発に従事．



畑中　勝実

1969年生．1992年豊橋技術科学大学情報工学課程卒業．1994年同大学大学院修士課程情報工学専攻修了．同年沖電気工業(株)入社．プログラミング言語処理系，並列処理に興味を持つ．現在，データベースシステムの開発に従事．