新しい計算モデル キューマシンとその並列関数型言語への応用

前田敦司† 中西正和†

本論文では，幅優先で式の評価を行う新たな計算機アーキテクチャであるキューマシンを提案し，その実行モデルを用いて関数型言語を特殊なハードウェアのサポートのない密結合並列計算機上で効率良く自動並列実行する言語処理系の構築法を述べる．関数型言語においては複数の関数呼び出しを並列に処理することが可能であるが，すべての関数呼び出しの実行が終了するまで待って実行を再開するための同期オーバヘッドが問題となる．また，通常スタックに保持する局所的な実行の文脈をヒープ上に保持する必要があるため，メモリ管理のオーバヘッドも増大する．本論文では，キューマシンの実行モデルを模倣してスタックをキューに置き換えることにより上記のオーバヘッドを大幅に削減することができ，既存の計算機上で並列関数呼び出しが効率良く実現できることを示す．この手法を用いたプロトタイプ言語処理系の実行時間を密結合並列計算機で測定した結果，逐次実行ではCなどの他の（逐次）言語処理系に劣るものの，2CPU以上では他の処理系を上回り，高い台数効果が得られている．

New Computation Model Queue Machine and Its Application to Parallel Functional Programming Languages

ATUSI MAEDA† and MASAKAZU NAKANISHI†

We present a new evaluation scheme for expressions called *queue machine model of execution* which enables automatic (implicit) parallel execution of functional programming languages with very small synchronization overhead without special hardware support.

In purely functional languages, multiple function call can be evaluated parallely without changing semantics of the program. But when implemented naively, synchronization overhead to wait for termination of all subcomputations becomes prohibitive. Moreover, local context information usually stored in stack must be maintained in garbage-collected heap. So overhead of memory management also increases when compared to sequential implementations.

In this paper, we show that by emulating execution model of queue machines and by replacing stacks with queues, the overhead can be drastically reduced and parallel function invocation can be implemented efficiently on stock hardware.

Preliminary measurement of prototype implementation based on this technique is presented. The measurement shows that, although programs compiled with our prototype compiler run slower than other implementations on sequential machines, they show good scalability and run faster than sequential implementations when executed with two or more processors.

1. はじめに

並列計算機が普及するにともない，その性能を引き出すための並列プログラミングが重要な研究課題となりつつある．純関数型言語のプログラムは副作用のない関数呼び出しの組合せからなる．したがってプログラムの各部分に依存性がきわめて少なく，並列実行に適しているとされてきた．しかしながら，特殊なハードウェアを持たない汎用の計算機上で個々の関数呼び出しを並列化の単位とした場合，並列化の粒度は非常に小さいものとなり，同期のオーバヘッドが並列化によって得られる利益を上回るため，速度の向上は期待できない．また，逐次型言語では関数呼び出しの文脈情報をスタックを用いて効率良く管理するが，並列関数呼び出しを実現する環境ではこの目的のためにスタックを用いることができず，より複雑なメモリ管理（ガーベッジコレクション）が必要となる．

本論文では，キューマシンと呼ぶ仮想の計算機アーキテクチャの実行を提案する．キューマシンはそれ自体で興味深い研究対象であるが，本論文では特にその実行を模倣することにより，特殊なハードウェアを持たない汎用の密結合並列計算機において同期やメモリ管理のオーバヘッドがきわめて小さい，効率の良い並

† 慶應義塾大学大学院理工学研究科
Graduate School of Science and Technology, Keio University

列関数呼び出しを実現できることを示す．実行対象とする言語としては，副作用のない関数型言語一般を対象にする．この手法により効率良い並列化が行われることを検証するため，制限つきながら副作用を許す関数型言語であるLinear Lispの処理系を試作し，並列化による速度の向上を確認した．

次章では，キューマシンの基本的なアーキテクチャを，スタックマシンと対比させて解説する．以降3章では試作処理系で用いたLinear Lispの簡単な解説と，キューマシンの設計上重要な概念について述べる．残りの章では，通常の計算機上でのキューマシンモデルの実装の詳細と，その並列処理への拡張，および現状と将来の展望について順に述べる．最後に本論文の結論と，他の研究との関連について述べる．

2. キューマシンのアーキテクチャ

2.1 スタックマシン

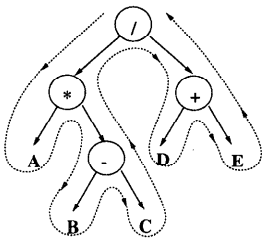
式評価の中間結果をLIFOメモリに貯える計算機をスタックマシン[14]と呼ぶ．スタックマシンは，計算機の歴史の初期から一般的な計算機アーキテクチャの1つである．このアーキテクチャは実際のプロセッサのアーキテクチャとしても多く採用され，またプログラミング言語処理系の中間言語としても広く用いられてきた．

スタックマシンはオペランドをスタックトップより取り出し，計算結果をスタックにプッシュする．この実行方式から，式をスタックマシンで評価するためのプログラムは逆ポーランド記法（RPN）として知られる独特の順序の機械語命令の列となる．式の構文木よりスタックマシンのために機械語命令を生成するには，式を深さ優先でたどり，節の子をすべて処理した後で節のラベルに対応する命令を生成すればよい（**図1**）．

レジスタマシンなど命令のオペランドとしてスタックを用いない計算機アーキテクチャも多いが，このような計算機においても（再帰的）関数呼び出しを処理するため何らかの形のスタック機構を備えているのが普通である．

たとえば，レジスタマシンでは小さい静的な式の評価にはスタックを用いる必要がない．しかし関数呼び出しが構成する動的な計算の木の全体の大きさには上限がなく，レジスタマシンに限らずいかなるアーキテクチャでも，この計算の木を巡回して制御を移していくうえで，実行の文脈を保存する何らかの記憶域が必要となる．この目的のために，最近の計算機ではほとんど例外なくスタックを用いている．

このようにスタックが広く使われている理由は前述



ABC-*DE+/

図1 スタックマシンの機械語の生成
Fig. 1 Code generation for stack machine.

したとおり，スタックを用いて木のすべての節を調べる効率の良いアルゴリズムが存在するからにほかならない．しかしながら，スタックを用いたツリーウオークは逐次的実行を強いることになる．なぜなら，すべての操作がスタックの片端（スタックトップ）に集中するため，前の計算がすべて終了してからでないと次の操作を実行できないからである．このため，スタックマシンは命令のパイプライン実行やオーバラップした実行，あるいは不整順序の実行（out-of-order execution）に向かず，高性能を目指すプロセッサで近年スタックマシンアーキテクチャを採用したものは少ない．

2.2 キューマシン

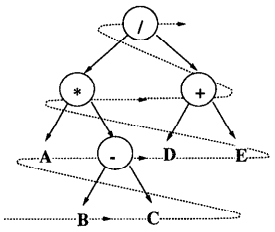
木の巡回に適するデータ構造はスタックのほかにも存在する．すなわち**キュー**がそれである．

キューの先頭より命令のオペランドを取り出し，結果をキューの末尾に入れるような計算機を想定する．このような計算機を**キューマシン**と呼ぶことにする．キューマシンで実行できるように式をコンパイルするには，式の構文木を（右から左へ）幅優先で探索し，訪れた順序と逆の順序に節を並べて対応する命令を出力すればよい（**図2**）．キューマシンの実行の様子をトレースした結果を**図3**に示す．

2.3 式の並列実行

キューマシンでは，キュー中のデータに対する演算を任意の順序で行ってよい．すべての演算結果をキューへ格納し終えた結果が正しい順序に並んでさえいればよい．たとえば図3において，乗算と加算はどちらを先に実行してもよいし，同時に（並列に）実行してもかまわない．キューマシンのキューは，非常に長い演算パイプラインと見なすことができる．キュー内につねに十分な数のデータがあれば，CPUの各機能ユニットは待ち状態に陥ることなく，効率の良い並列処理を行うことが期待できる．

単一の式を評価する際，キュー内のデータ数は構文



BCA-DE*+/

図2 キューマシンの機械語の生成
Fig. 2 Code generation for queue machine.

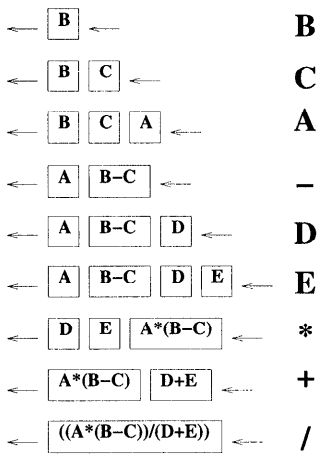
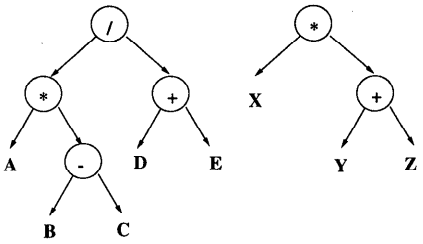


図3 キューマシンの実行トレース
Fig. 3 Execution example of queue machine.

木の幅によって決まる．構文木のルートノード近くのノードではキュー中のデータ数が少なくなり，直前の命令の実行結果がキューに入るまで演算を実行できない待ち状態が生ずる．

独立した2つの式の構文木を横に並べ，評価をオーバラップして疑似並列実行を行うことにより，式の見かけの幅を増やし，キュー中のデータ数を増加させることができる（図4）．さらに，任意の数の式に対してこの技法を拡張することができる（図5）．高さをずらして式を並べ，オーバラップした実行を行うことにより，キュー中のデータ数がつねにほぼ一定となるように制御することが可能となる

長いパイプラインレイテンシを持つ最近のRISCプロセッサでは，データの依存性によるハザードを取り除く最適化（code scheduling）をコンパイラが行っている[7),17)]．キューマシンにおけるオーバラップした式の評価は，code schedulingと同様にパイプラインレイテンシを隠蔽し，命令レベルでの高い並列性をもたらすことが期待できる．



BCA-DEYZ*+X+/*

図4 2つの式の並列実行
Fig. 4 Interleaved evaluation of two expressions.

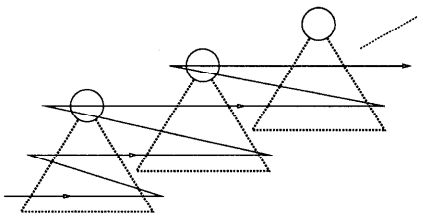


図5 複数の式のインタリーブ実行
Fig. 5 Interleaved evaluation of multiple expressions.

3. 線形論理とLinear Lisp

線形論理[10)]はGirardにより提唱された新しい論理体系で，資源や状態といった概念を古典論理より良く記述できるという特徴を持っている[5),11)]．このためプログラミング言語理論の分野で線形論理を用いた研究がさかんに行われている[1),15)]．

Lisp言語に線形論理を応用した例としてBakerのLinear Lisp[3)~5)]がある．

通常のLispとの差異として，Linear Lispにおけるすべての変数はちょうど1回だけ参照しなければならない．複数回使いたい場合は内容を陽にコピーしなければならない．1度も使わない場合は陽に消去しなければならない．

変数の使用に関する上記の規則はコンパイラにより容易にチェックでき，違反した場合はエラーとなる．

データの使用にこの制限を設けた結果，すべてのデータへの参照はただ1つとなる．したがって，以下の利点が得られる．

- 不要となったデータをただちに検出し，回収することができる．
- セルや配列のデータの変更を破壊的に行っても，参照の透明性を損なうことがない．
- データの参照や変更を並列に行っても他のプロセスと干渉することがない（排他制御の必要がない）．

これらの性質により，Linear Lispは効率の良い破

壊的な代入操作を許しながら，事実上は副作用のない関数型言語となっている．

純粋に線形のデータのみを扱うとすると，大きな定数などを何度もコピーする必要が生じ，またグラフ構造が扱えないなどの欠点があるため現実的でない．そのため Linear Lisp では複数回参照できるデータ（非線形のデータ）も許している[5]．非線形のデータは読み出し専用である．

4. キューマシンの実現

現在の計算機アーキテクチャの主流はレジスタマシンである．レジスタマシンにおいて小さな（静的な）式を評価する際に，キューマシンと同様の FIFO 順の評価を行うのは効率的でないと思われる．しかしながら，関数呼び出しの実現にキューを用いるのは比較的容易であり，スタックを用いた場合と同程度の手間ですむ．

この章では，既存の計算機上でキューマシンと同様の幅優先実行を実現する手法およびその利点について述べる．

4.1 フレームと関数呼び出しの木

スタックを用いた実行モデルでは，関数呼び出しのたびに実行の文脈を保持するメモリ領域をスタック内に割り付ける．このメモリ領域を通常はスタックフレームと呼ぶが，ここでは単に**フレーム**と呼ぶことにする．フレーム内には通常，その局所変数の値・呼び出し元のフレームへのポインタ・プログラムカウンタの値などが保持される．

プログラムの実行が進むとともに生成されるフレームをすべて図示すれば**図 6** に示すような木構造となる．図中の矢印は呼び出し元のフレームへのポインタを示している（以下，矢印が指し示している先のフレームを**親**，矢印が出ているフレームを**子**のフレームとそれぞれ呼ぶことにする）．スタックを用いた実行モデルでは，プログラム実行の 1 つの時点ではこの木構造のうち 1 本の経路上にあるフレームのみを 1 次元のスタックメモリ内に保持しているのが普通である（たとえば，図 6 中の点線で囲んだ部分）．フレームは関数呼び出しの際に 1 つずつ割り付けられ，関数呼び出しから戻る際に消去される．

前章で述べた Linear Lisp のような副作用のない関数型言語では，複数の関数呼び出しを並列に行ってもプログラムの意味は変わらない．この場合，あるフレームを指すフレームが同時に複数存在することになり，実行時のフレーム間の接続は 1 次元でなく，実際に木構造をなす．木構造の葉のフレームはその時点

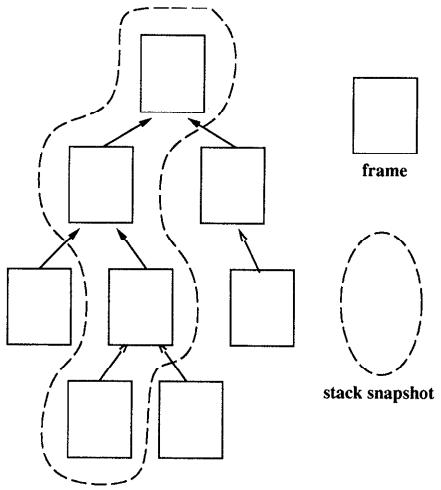


図 6 関数呼び出しの動的な木
Fig. 6 Dynamic tree of function invocation.

で実行可能であり，葉でないフレームは子のフレームがすべて実行を終えて値を返すまで実行することができない．以下では，実行可能な葉のフレームを**正のフレーム**，子のフレームの実行終了を待っているフレームを**負のフレーム**と呼ぶことにする．

関数呼び出しを単純に並列実行するには，各フレームに子のフレームの数を示す参照カウントを設け，任意の順序で正のフレームを実行し，実行を終えた際に呼び出し元のフレームの参照カウントを減じて，0 になれば実行を再開するようにすればよい．しかしながら，この単純な実行方式ではオーバヘッドが大きく，高速化は期待できない．関数呼び出しごとに 2 度ずつ，排他的な操作を要する参照カウントの操作が必要となるためである．また，フレーム間の接続が木構造となるため，フレームを割り付ける領域としてスタックを用いることはできず，ガーベッジコレクションを行う必要がある．このため，メモリ管理のオーバヘッドも大きくなる．

4.2 セグメントを用いた効率化

上で述べた並列関数呼び出しの問題点を解決するため，キューマシンの実行モデルを導入して効率化をはかる．各フレームを完全に非同期に実行するのではなく，木の同じ深さのフレームをキューマシンと同様の順序でなるべく 1 次元的に並べ，順に実行することにする．

木の全体の形は動的な計算の過程によって定まるので，完全に順序づけて単一のキューを用いて実行することは困難である．そこで，フレームはキューセグメント，または単に**セグメント**と呼ぶ固定長の記憶域中に割り付けることとし，同じセグメント中のフレーム

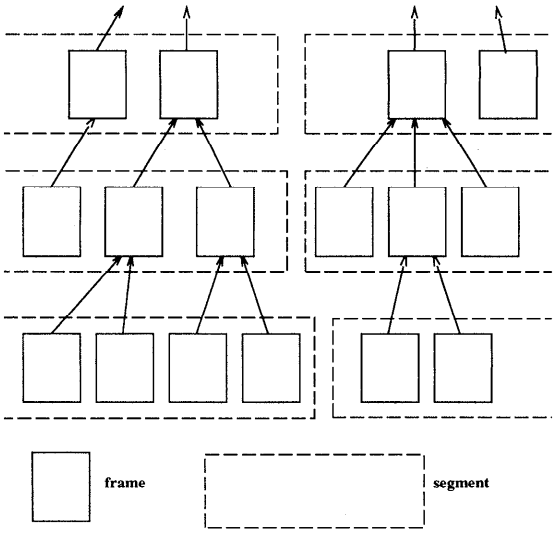


図7 セグメントによるフレームのグループ化
Fig. 7 Clustering of frames into segments.

が指す親のフレームは，すべて同一のセグメント内にあるようにする（**図7**）．セグメント中のフレームがすべて正のフレームであるとき，そのセグメントを**正のセグメント**，1つでも負のフレームがあるとき，**負のセグメント**とそれぞれ呼ぶことにする．

1つのセグメントAの中のフレームはすべて同じセグメントBの中のフレームを指すようにした．このときフレームと同様にAを子，Bを親と呼ぶことにすれば親子関係がセグメントについても定義でき，セグメントが正となるのは子セグメントの数が0であるときかつそのときに限ることが分かる．したがって各フレームに参照カウントを持つ必要はなく，セグメントごとに1つ参照カウントを設ければよいので，結果として実行時の排他操作の回数が大幅に削減できる．

またセグメント内にフレームを割り付ける際には，スタック中にフレームを割り付けるのと同様にオーバフローをチェックしつつ1次元的にメモリを割り付けていけばよく，メモリ管理のオーバヘッドも小さくなる．

以下ではこの実行方式を用いるためのフレームとセグメントの構造について詳細に述べる．

フレーム内には以下の情報を保持する（**図8**）．

- プログラムコードの入口番地を指し示すポインタ．
- プログラムの入力となる変数領域（入力フィールド）．
- 関数の値を返す親のフレーム内の，入力フィールドを指すポインタ（**出力ポインタ**）．

また，各セグメントには以下の情報を保持する

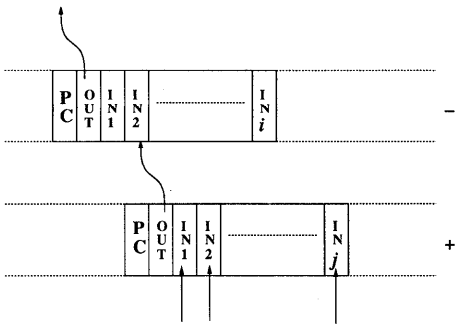


図8 フレームの構造
Fig. 8 Frame structure.

（**図9**）．

- 排他制御のためのロック．
- 親のセグメントを指すポインタ．
- 参照カウント．
- 正のセグメントを双方向リストに接続しておくためのポインタ．
- このセグメントの終りのアドレス．
- 1個以上のフレーム．

正のセグメントはすべてセグメントプールと呼ぶセグメントの集まりに入れておき，プロセッサはプールから1つずつセグメントを取り出して実行する．セグメント内の各フレームの実行は，フレーム内のプログラムカウンタが指すプログラムコードブロックの先頭にジャンプすることによって行う．コードブロックの末尾には，セグメントから次のフレームを取り出して実行するコードを入れておく．

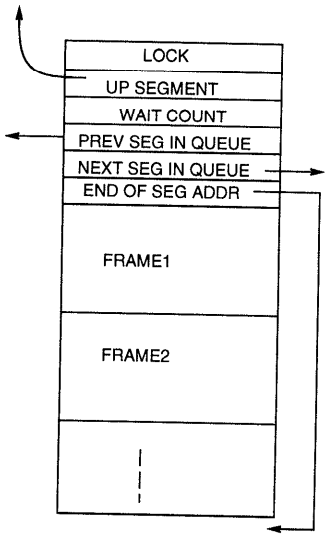


図9 セグメントの構造
Fig. 9 Segment structure.

現在処理しているセグメント（**入力セグメント**）の中の1つのフレームを実行した結果は以下の3通りのいずれかとなる．

- 出力ポインタが指すフレームに演算結果を返す（変数，定数，基本演算）．
- 関数呼び出しの深さが同じ正のセグメントに新たなフレームを割り付ける（末尾再帰的な関数呼び出し）．現在実行中のフレームの出力ポインタを，新しいフレームの出力ポインタとして書き込む．
- 同じ深さの負のセグメントと，1つ深いレベルの正のセグメントにフレームを割り付ける（末尾再帰的でない関数呼び出し）．正のセグメントに割り付けたフレームの出力ポインタは，負のセグメントに割り付けたフレーム内のあるフィールドを指す．負のセグメントに割り付けたフレームの出力ポインタには，現在実行中のフレームの出力ポインタを書き込む．

入力セグメントを最後まで処理し終えると，そのセグメントの親の参照カウントを1だけ減じ，もし0になれば親セグメントをセグメントプールに入れる．

上で述べたフレームの実行結果3通りに応じて，入力セグメント以外に，フレームを割り付けるためのセグメント（**出力セグメント**）を3つ保持しておく必要がある（現在のレベルの正および負，1つ深いレベルの正）．実行時に保持するセグメントの状態を**図10**に示す．

出力セグメント内に新たなフレームを割り付ける際に，セグメントに十分な余地がない場合は新たなセグメントを割り付ける．したがって，1つの出力セグメントにつき，最後にフレームを割り付けた位置を指すポインタ（フレームポインタ）とセグメントの末尾を示すポインタの2つのポインタが必要となる．フレームを割り付けるセグメントは3種類あるので，合計で6つの出力用ポインタが必要となる．さらに，入力セグメント中のフレームを指すポインタが1つ必要である．ここで，入力セグメントの末尾をチェックするためのポインタは不要である．セグメントの最後のフレームに，セグメントが空になった際の処理ルーチンのアドレスを入れておくことにより，セグメント終了を検出することができる．

実行中に正のセグメントがいっぱいになった場合，そのセグメントはただちにセグメントプールに移される．また，入力セグメントの処理を終えた際，正の出力セグメントが空でなければそのセグメントもセグメントプールに入れる．

4.3 複数のプロセッサによる実行

正のセグメント内のすべてのフレームは，互いにデータの依存関係がない．したがってセグメントをキューに入れてFIFO順で処理する必要はなく，プール内のセグメントを任意の順序で処理すればよい．またLinear Lispの線形性から，正のセグメント内のデータが複数のフレームで共有されていることはなく，データ参照時に排他制御はまったく不要である．このため複数のセグメントを複数のプロセッサを用いて並列に処理することができる．

この場合，排他制御が必要となるのはセグメントの参照カウントを増減する場合，およびセグメントプールに対してセグメントの挿入・取出しの操作を行う場合のみである．すなわち，排他制御は個々のフレーム単位ではまったく不要で，セグメント単位で行うだけでよい．通常はセグメント中の各フレームを逐次的に実行し続けることができるので，セグメントが十分大きければ排他制御の頻度を大幅に削減することができ，効率を高めることができる．

4.4 並列度の制御

プールからセグメントを取り出す順序が，本実行モデルのスケジューリング方式を決定することになる．また，正のセグメントの総数がその時点での並列度を示す．正のセグメント全体をFIFOで処理した場合，計算の木のサイズに比例する記憶域が必要となり，空間計算量はスタックを用いた逐次的な実行方式と比較して指数関数的に増大することとなる．そこで，十分な並列度が得られた後には正のセグメントをLIFO順で処理することにより並列度の爆発を避ける．これ

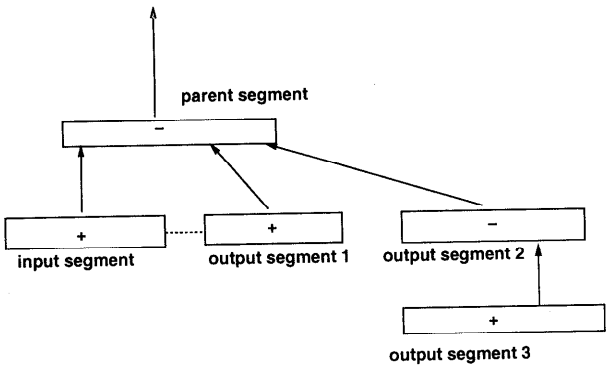


図10 実行時のセグメントの構成
Fig. 10 Organization of segments at runtime.

によりスタックに基づいた通常の処理方式とほぼ同じオーダの空間計算量で計算を行うことが可能となる．

また，計算の初期の時点では正のセグメントが1つしかなく並列処理が行えない．正のセグメントが少ないときにはセグメントのサイズを小さくすることにより意図的にセグメントの分裂を起こし，並列度をより速く増加させるようにする．

5. 他の研究との関連

Baker は，再帰的呼び出しを用いるプログラミング言語においてもスタックによる記憶域の割付けは必ずしも必須ではなく，単に処理系作成者が取り得る選択肢の1つにすぎないと指摘した[2]．ただし，Baker がスタック以外の選択肢としてあげたのはキューではなくガーベッジコレクションにより管理されるヒープである．

キューマシンの実行モデルは，データフローマシン[20]と多くの類似点を持っている．スタックに基づいた実行モデルにおいて，データの流れは制御の流れと一致している．すなわち，関数への引数は呼び出しと同時に関数へ渡され，関数からの戻り値は関数から制御が復帰する際に返される．キューマシンでは，制御の流れとデータの流れが切り離されている．実行はセグメント内のフレームの列に沿ってシーケンシャルに行われ，データはその実行の流れとは別にセグメント間で転送される．

並列度が飽和している状況でのセグメントの LIFO スケジューリングは，Multilisp の unfair scheduling policy[12]と類似している．ただし本稿の方式はそれに加えて並列度が不足している際にセグメントを分割して並列度を増加させる制御を可能としている．結果として，本稿で述べたスケジューリング方式は，理想的な動的スケジューリング方式である BUSD (Breadth-first Until Saturation, then Depth-first)[16]のより良い近似と見なすことができる．

間接ジャンプを用いたキューフレームの実行は，Forth インタプリタなどで用いられるスレッデッドコード[6]とほぼ同じである．ただし，本稿の方式では実行されたスレッデッドコードはキューから取り除かれ，消えてなくなる（use-once である）点が異なる．実際，ハードウェアで実現する場合には，プログラムのアドレスでなくプログラムコード自身をキューに入れた方が効率が良くなる可能性がある．プログラムコードを毎回キューに入れるのは非効率なようであるが，いずれにせよプログラムはすべてプロセッサに読み込まれるわけであるから，ポインタを経由する場合よりもバストラフィックが低減できる可能性がある．この場合，セグメントはプロセッサ上で巨大な命令パイプラインとそれと同期して進むデータキューの対を形成することになる．もしプロセッサ内に多数のセグメントをキャッシュできるならば，特に効率の良い実行が期待できる．スタックにプッシュしたプログラムを実行する提案に文献 19) がある．これは，自己修飾コード（self-modifying code）を制限した形で導入することにより柔軟性と安全性を両立させるという提案である．

線形論理に基づいた言語の実行に適したスタックアーキテクチャが提案されている[4]．このアーキテクチャは，スタックトップに近い要素を並べ替える（rotate する）ことができるスタックを持つ．また，同じ論文の中で，スタック内に構築して実行すると消える（execute once）プログラムについても述べられている．実際には Linear Lisp を PostScript にコンパイルし，スタックの並べ替えには PostScript の rotate オペレータを用いている．

表1 thread 数と実行時間（単位：秒）
Table 1 Number of threads vs. elapsed time (in seconds).

処理系	(fib 35)	相対時間	(tak 27 18 9)	相対時間
LL (1 thread)	17.66	(1.00)	7.76	(1.00)
LL (2 threads)	8.69	(0.49)	3.92	(0.51)
LL (3 threads)	6.86	(0.38)	2.77	(0.36)
LL (4 threads)	4.92	(0.28)	2.11	(0.27)
GCC (参考)	9.35	(0.53)	3.98	(0.52)
CMU CL (参考)	14.33	(0.81)	4.84	(0.62)

6. おわりに

6.1 結　論

本論文では，細粒度の並列処理に適した新たなコンピュータアーキテクチャであるキューマシンを提案した．また，既存の計算機上でキューマシンの実行をエミュレートすることにより，関数型言語のプログラムを効率良く並列実行する実行方式を提示した．この実行方式においては，個々のフレーム単位でなくセグメントごとに同期操作を行うため，並列性の粒度を適切な大きさに保つことができ，同期オーバヘッドをきわめて小さく保つことができる．またメモリ管理に関しても，フレームの割付けを一次元のセグメント内で行い，参照カウントを用いたメモリ管理をセグメント単位にまとめて行うため，個々のフレームに関するメモリ管理のオーバヘッドはスタックを用いた通常の関数呼び出しと比較できる程度に小さなものとなる．

この実行方式において，並列度は容易に調節することが可能であり，メモリ消費量と処理速度の最適なトレードオフを得ることができる．さらに，並列性の抽出はコンパイラにより完全に自動的に行われる．

6.2 キューマシンモデルの効果

キューマシンモデルを用いた並列化により実際に速度が向上することを確かめるため，Linear Lisp を並列実行する試験的な処理系を作成した．この処理系により作成したプログラムを密結合並列計算機上で実行した際の実行時間を**表1**に示す．また，言語がまったく異なるため直接の比較は難しいが，参考として GNU C v2.7.2（表中では GCC と略記）および CMU Common Lisp 17f（CMU CL と略記）で同等のプログラムを実行した場合の実行時間を併記する．GCC のコンパイルオプションには`-O3`，CMU CL のコンパイラの設定には`(speed 3) (space 0) (safety 0)`をそれぞれ指定した．

テストプログラムとしては Fibonacci 関数および TAK 関数[9]を用い，50 MHz で動作している SuperSPARC 4 CPU の計算機（OS は Solaris 2.3）上で計測した経過時間を示した．並列化による速度向上は，Solaris の thread ライブラリを用い thread 数を変化させることにより測定した．括弧内の数値は Linear Lisp（表中では LL）を 1thread で動作させたときの経過時間を基準とした相対値である．現在の処理系では，ほとんど何の最適化も行っておらず，単一の thread で実行した場合，同等の C プログラムに比べて Linear Lisp は約 2 倍の実行時間となっているが，thread 数の増加に応じて速度が向上し，2 thread 以上では C をしのぐ処理速度を示している．

6.3 現時点における課題

現在未解決の問題として，アッカーマン関数のように枝分かれのない深い再帰呼び出しを行うプログラムではメモリ効率が非常に悪い点があげられる．これは，1 つしかフレームを持たない負のセグメントを大量に保持する必要があるためである．このような場合，キューマシン実行方式はスタックマシンの効率の悪いエミュレーションを行っていることになる．すべての関数呼び出しごとにセグメントの割付けが必要となり，また関数から復帰する際には参照カウントの処理とプールの操作が毎回必要となってしまう．

また，セグメント中に 1 つでも負のフレームがあると，そのセグメントは負になるため，問題によっては，真の並列度と実行できる並列度の間に大きな差が生じることがありうる．特に，プログラマが陽に同期操作を行うことのできる構文を導入した場合，待っているフレームと同一セグメント中にあるフレームがすべて待たされることになり，場合によってはデッドロックを生じる可能性もある．

現在の，陰の並列性を用いた実装ではデッドロックは生じない．並列度が低くなることによる効率の低下については，通常の並列度不足の場合と同様に，セグメントのサイズを縮小することによってセグメントの分裂を起こし，正のセグメントの増加をはかることができるが，この制御の有効性については個々の問題についてさらに検討が必要であると思われる．

6.4 将来の展望

将来の研究の方向性として，キューマシンのハードウェアによる実現と，Linear Lisp の分散並列処理へ

の応用が有望である．キューマシンは，依存性の少ない高度な並列性を内包しており，ハードウェアによる実現に適していると考えられる．メモリアクセスのパターンはきわめて規則的であり，セグメントの内容は1度しか読み出さないので，通常のRAMの代わりに1度読み出せば内容が破壊されるようなメモリ（read-once RAM）を用いることができ，メモリ転送の高速化やチップ面積の削減をはかることができる．また，通常の条件分岐命令に替えて条件PCストア命令を設けることにより，条件文の実行をキューの次の回転に延期することができ，条件分岐命令によって引き起こされるパイプラインストールをなくすことが期待できる．

本論文で述べた並列化手法は，Linear Lispにとどまらずstrictな引数評価を行う副作用を持たない関数型言語全般に適用できる．Haskell[13]やMiranda[18]など引数をlazyに評価する言語は，引数が必ずしも評価されないため，本論文の並列化手法を適用するにはコンパイル時にstrictness analysis[8]などの手法を併用して，プログラムの意味が並列化により変わってしまわないことを検証する必要がある．また，特にLinear Lispを対象とした場合はセグメント間にデータの共有がなく，またセグメントの実行に必要なデータはすべて直接セグメント内に保持されているという特徴がある．この局所性とデータの独立性から，本論文の手法をLinear Lispの分散並列処理に適用することが可能であると思われる．

謝辞 本研究の初期から，貴重な示唆に富んだコメントをいただいた國吉芳夫氏に感謝する．本稿の作成にあたって数々のアドバイスをいただいた田中良夫氏と田中詠子氏に感謝する．

参考文献

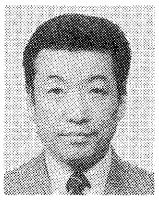
1) Abramsky, S.: Computational Interpretations of Linear Logic, *Theoretical Computer Science*, Vol.111, pp.3–57 (1993).
2) Baker, H.: CONS Should not CONS Its Arguments, or a Lazy Alloc is a Smart Alloc, *ACM SIGPLAN Notices*, Vol.27, No.3, pp.24–34 (1992).
3) Baker, H.: Lively Linear Lisp – 'Look Ma, No Garbage!', *ACM SIGPLAN Notices*, Vol.27, No.8, pp.89–98 (1992).
4) Baker, H.: Linear Logic and Permutation Stacks – The Forth Shall Be First, *ACM Computer Architecture News*, Vol.22, No.1, pp.34–43 (1994).
5) Baker, H.: 'Use-Once' Variables and Linear Objects – Storage Management, Reflection and Multi-Threading, *ACM SIGPLAN Notices*, Vol.30, No.1, pp.45–52 (1995).
6) Bell, J.: Threaded Code, *Comm. ACM*, Vol.16, No.6, pp.370–372 (1973).
7) Bernstein, D. and Rodeh, M.: Global Instruction Scheduling for Superscalar Machines, *Proc. SIGPLAN '91 Conference on Programming Language Design and Implementation*, pp.241–255, ACM Press (1991).
8) Clack, C.D. and Jones, S.L.P.: Strictness Analysis – A Practical Approach, *Proc. 1981 Conference on Functional Programming Languages and Computer Architecture*, LNCS, Vol.201, pp.35–49, Springer-Verlag (1985).
9) Gabriel, R.P.: *Performance and Evaluation of LISP Systems*, Stanford Univ., Stanford (1982).
10) Girard, J.-Y.: Linear Logic, *Theoretical Computer Science*, Vol.50, pp.1–102 (1987).
11) Girard, J.-Y.: Linear Logic: Its Syntax and Semantics, *Advances in Linear Logic* (Proc. Workshop on Linear Logic, Ithaca, New York, June 1993), Girard, J.-Y., Lafont, Y. and Regnier, L. (Eds.), pp.1–42, Cambridge University Press (1995).
12) Halstead, R.: Multilisp: A Language for Concurrent Symbolic Computation, *ACM Trans. Prog. Lang. Syst.*, Vol.7, No.4, pp.501–538 (1985).
13) Hudak, P. and Wadler, P. (Eds.): Report on the Functional Programming Language Haskell, Technical Report YALEU/DCS/RR656, Department of Computer Science, Yale University (1988).
14) Koopman, P.: *Stack Computers: The New Wave*, Ellis Horwood, West Sussex, England (1989). 藤井敬雄（訳）：スタックコンピュータ～CISC/RISCとスタックアーキテクチャ，共立出版 (1994).
15) Lafont, Y.: From Proof Nets to Interaction Nets, *Advances in Linear Logic* (Proc. Workshop on Linear Logic, Ithaca, New York, June 1993), Girard, J.-Y., Lafont, Y. and Regnier, L. (Eds.), pp.225–247, Cambridge University Press (1995).
16) Mohr, E., Kranz, D. and Halstead, R.: Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs, *Conf. Record of the 1990 ACM Conf. LISP & FP*, Nice, France, pp.185–197, ACM Press (1990).
17) Pinter, S.: Register Allocation with Instruction Scheduling: A New Approach, *Proc. ACM '93 Conference on Programming Language De-*

sign and Implementation, Albuquerque, New Mexico, pp.248–256, ACM Press (1993). Also in *SIGPLAN Notices*; Vol.28, No.6, (1993).
18) Turner, D.A.: Miranda: A Non-strict Functional Language with Polymorphic Types, *Proc. 1981 Conference on Functional Programming Languages and Computer Architecture*, LNCS, Vol.201, pp.1–16, Springer-Verlag (1985).
19) 塚本享治：プログラム・スタッキング技法，情報処理，Vol.17, No.3, pp.237–244 (1977).
20) 弓場敏嗣，山口喜教：データ駆動型並列計算機，オーム社 (1993).

(平成8年 3 月 21 日受付)
(平成8年 12 月 5 日採録)

前田 敦司（正会員）

昭和 61 年慶應義塾大学理工学部数理科学科卒業．平成 6 年同大学大学院後期博士課程単位取得退学．現在，同大学数理科学科中西研究室研究生．Lisp 処理系，コンパイラ，プログラミング言語理論，情報理論に関心を持つ．ソフトウェア科学会，ACM 各会員．



中西 正和（正会員）

昭和 41 年慶應義塾大学工学部卒業．昭和 44 年同大学工学部助手．平成元年同大学理工学部教授．工学博士．昭和 42 年，日本初の実用 Lisp 処理系を作成．以後，記号処理言語，人工知能用言語等の研究に従事．昭和 57 年，Lisp マシン SYNAPSE の開発など．情報処理学会プログラミングシンポジウム委員会幹事長．