プログラミング 16－2
（1997．11．21）

キューマシン方式並列実行の複数階層に渡る関数呼び出しフレームの併合による効率化

船戸 潤一† 前田 敦司†† 中西 正和†††

† 慶應義塾大学大学院 理工学研究科 計算機科学専攻
†† 電気通信大学大学院 情報システム学研究科
††† 慶應義塾大学 理工学部 情報工学科

本論文ではキューマシン方式並列実行におけるタスク併合の手法を拡張し，複数階層に渡る関数呼び出しもまとめて一つのタスクとする手法を提案する．従来のキューマシン方式は同一階層の関数呼び出しフレームをキューセグメント単位に併合してタスクとしたため，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実行効率が低下する．そこでキューセグメントをフレームを保持するスタックと見なすことで，複数階層に渡るフレームの併合も可能とする手法を示す．この手法を用いた処理系を実装した結果，従来のキューマシン方式の利点を生かしたまま，枝分かれの少ない深い再帰呼び出しを行うプログラムにおいても，効率の良い実行が可能であることが示された．

The New Task-Merging Method for Queue Machine Model of Execution

Junichi FUNATO† Atsushi MAEDA†† Masakazu NAKANISHI†††

† Department of Computer Science Graduate School of Science and Technology, Keio University
†† Graduate School of Information Systems, The University of Electo-Communications
††† Department of Information and Computer Science, Faculty of Science and Technology, Keio University

In our previous implementation of queue-machine–based parallel execution, child frames are always allocated in deeper segments. With this strategy, thin and deep recursive calls resulted in tall tree of sparse segments, yielded poor performance. In this paper, we present a new task-merging method which can merge multiple level frames. In this method, when no more than one function call is performed in a frame, current segment is treated as a stack and the child frame is allocated in LIFO manner. As a result, when programs contain thin and deep calls, our new implementation performs much better than the previous one.

1 はじめに

キューマシン方式並列実行 [2] は，関数型言語のプログラムを並列に実行するための実行モデルの一つである．キューマシン方式による並列実行では，同じ深さの関数呼び出しフレームをまとめたキューセグメントを単位として並列化を行う．この方法により，従来のスタックを用いた並列実行の手法では効率の悪かった，並列度が高く計算木の形が均等でないプログラムにおいて，効率の良い実行が可能となった．しかし一方で，同じ深さの関数呼び出しを多く生成しないプログラムでは，一つのタスクの粒度は大きくならず，並列化の効果は得られない．

本研究では，キューマシン方式で用いられているタスク併合の手法を拡張し，単純なキューマシン方式では効率の低下する場合に，キューセグメントをスタックと見なすことによりタスクの粒度を大きくする手法を提案する．具体的には，複数階層にわたる逐次的な関数呼び出しフレームをまとめて，一つのキューセグメントに割り付けて実行する．この方法により，粒度の小さなタスクの生成を抑え，より効率の良い並列実行が可能となる．

2 キューマシン方式並列関数呼出

本章ではキューマシン方式並列関数呼び出し [2, 3] の概要について述べる．

2.1 キューマシン

スタックマシンと対照的な動作をする計算機アーキテクチャとして，キューマシン [2] がある．キューマシンによる計算の進め方は，(1) キューの先頭から一組の演算子と被演算子を取り出し，(2) その演算を行い，(3) 演算結果をキューの末尾に入れる，という操作を繰り返す．

キューマシンの一番の特徴は，キューの中のデータの並び順が正しく保たれていれば，各データは任意の順序で演算を行えることである．

2.2 複数のキューによるキューマシンのエミュレーション

前節で述べたキューマシンアーキテクチャをそのまま用いて計算を行うには，一つの問題がある．それは，この方法では，計算木の大きさが動的に決まるような式に対応できないことである．再帰的な関数呼び出しのように，計算木の大きさがデータに依存して実行時まで確定できないものの場合，コンパイル時にすべての計算木のノードを，キューマシンの実行順序に従って並べることは困難である．したがって，前節で述べた単一のキューから成るキューマシンでは，関数型言語のプログラムを実行するには不十分であるといえる．

そこでこの問題の解決策として，単一のキューではなく，**関数呼び出しの深さに応じた複数のキュー**を用いることにする．ある深さのキューで呼び出された関数は，その呼び出しの深さに応じたキューに入れられる．この方法により，一つのキューは計算木全体を保持する必要はなく，ある関数呼び出しのレベルの要素だけを保持すればよい．

ここで，呼び出しの深さごとに生成されるキューを，キューセグメント (queue segment) あるいは単にセグメント (segment) と呼ぶことにする．

このように，本来は単一のキューであるべきものを，複数のレベルのキューに分けてエミュレートすることにより，キューマシンという実行モデルを実現できる．

2.3 セグメント単位の実行による効率化

2.3.1 関数呼び出しフレーム

一つの関数呼び出しの情報は，フレーム (frame) と呼ばれる単位で保持される．このフレームが一つの実行単位である．またキューセグメント内のフレームは，キューの中のデータは任意の順序で実行できるというキューマシンの性質より，それぞれ並列に実行可能である．

フレームには次のような情報を格納する．

- 関数のプログラムコードの先頭番地を指すポインタ (プログラムカウンタ：PC)
- 関数の値を返す先のポインタ (出力ポインタ)
- 関数の引数となる変数領域 (入力フィールド)

出力ポインタは値を返す先として，ほかのフレームの入力フィールドを指す．

関数呼び出しを実行して計算を進めると，フレームをノードとする計算木ができる．複数のフレームが並列に実行可能であるとすると，実行時のフレーム間の接続は 1 次元ではなく，実際に木構造をなす．計算木の節にあたるフレームは子のフレームからの実行結果を待っている状態であり，葉にあたるフレー

ムはすぐに実行できるフレームである．ここで，実行可能なフレームを**正のフレーム**，子のフレームの実行終了を待っていて実行不可能なフレームを**負のフレーム**と呼ぶことにする．

実行時のフレーム間の構成が木構造になるため，並列関数呼び出しを行う際には次の二つのオーバーヘッドがかかる．

- 同期をとるためのオーバーヘッド
- メモリ管理のオーバーヘッド

まず同期のオーバーヘッドだが，負のフレームが実行可能となるためには，すべての子のフレームが実行を終えるのを待ち合わせなければならない．そのためには負のフレームに参照カウントを持たせればよいのだが，参照カウントにアクセスするたびに排他制御を行わなければならず，大きなオーバーヘッドがかかる．次にメモリ管理についてだが，実行時のフレームの構成が木構造になることから，フレームをスタックに割り付けることはできない．したがってガーベッジコレクションなど，より複雑なメモリ管理を行う必要性が生じ，メモリ管理のオーバーヘッドが増大する[1]．

2.3.2 セグメント単位の実行

フレームは一つの実行単位となるものだが，フレーム一つの粒度は非常に小さい．したがってフレームを単位として並列実行を行うと，2.3.1節で述べたオーバーヘッドが大きくかかる．そこで，タスクの粒度を大きくするために，**キューセグメントを単位とした並列実行**を考える．セグメントは互いに依存関係のないフレームの集まりなので，このセグメントを並列実行の一つの単位として用いることができる．

セグメント単位の実行により，前述した二つのオーバーヘッドはいずれも大幅に削減される．まず同期のオーバーヘッドについては，参照カウントはフレームごとではなくセグメントごとに保持すればよく，参照カウンタ増減のための排他制御の回数を大幅に減らすことができる．またメモリ管理については，フレームはセグメントに1次元的に割り付けていけばよく，ガーベッジコレクションもセグメント単位で行えばよいため，オーバーヘッドは小さくなる．

[1] 言語によっては大差ないとする説もある [1].

2.3.3 実行時のセグメントの構成

キューマシン方式では，あるセグメント内の一つのフレームが実行された結果は，他のセグメントに渡される．実行時のセグメントは，図1のように構成される．ここで，一つのフレームが実行された際の，フレームの割り付けられ方・セグメントの構成のされ方をみる．

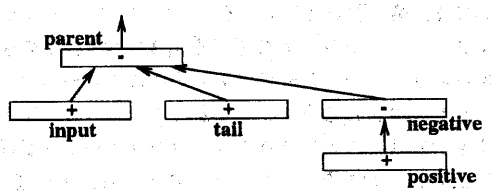

図 1: 実行時のセグメントの構成

現在処理しているセグメントを**入力セグメント (input segment)** と呼ぶことにすると，入力セグメント中の1つのフレームを実行した結果は，次のいずれかになる．

1. 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，プリミティブな演算の場合
2. 関数呼び出しの深さが同じ正のセグメント (tail segment) に，新たなフレームを割り付ける
   – 単純な tail recursive な関数呼び出しの場合
3. 関数呼び出しの深さが同じ負のセグメント (negative segment) と，一つ深いレベルの正のセグメント (positive segment) に，新たなフレームを割り付ける
   – tail recursive でない関数呼び出しの場合

出力先となる tail segment，negative segment，positive segment を総称して，**出力セグメント (output segment)** と呼ぶことにする．

3 キューマシン方式の問題点

キューマシン方式並列関数呼び出しは，枝分かれのない深い再帰呼び出しを行うプログラムの実行効率が低下する [3]．その原因は，キューマシン方式のフレーム割り付けの方法にある．キューマシン方式では，同じレベルの関数呼び出しをまとめたセグメントを，一つのタスクとする．そのため枝分かれのない深い再帰呼び出しを行うプログラム部分では，セグメント内のフレームの数が非常に少なくなる．し

たがって，一つのタスクの粒度が小さくなり，スケジューリングによるオーバーヘッドが大きくなる．

ここで，アッカーマン関数 (図 2，以降 ack と表記) を例にとり，フレーム割り付けが行われる様子を示す．Ack は並行して二つ以上の関数呼び出しを行わないため，すべての ack 呼び出しのフレームを新たなセグメントに割り付ける．Ack に第一引数として 3，第二引数として 8 を与えて実行したときの，セグメントとフレームの様子を図 3 に示す．この図 3 のように，一つのセグメントの中にフレームは一つしか割り付けらない．したがって，一つのフレームを実行するごとにスケジューリングルーチンが呼び出され，大きなオーバーヘッドとなる．また図 3 からも明らかであるが，一つのセグメント内に大量の空き領域ができてしまい，メモリ効率も悪い[2]．

```
(defun ack (x y)
  (cond ((zerop x) (1+ y))
        ((zerop y) (ack (1- x) 1))
        (t (ack (1- x) (ack x (1- y))))))
```

図 2: アッカーマン関数

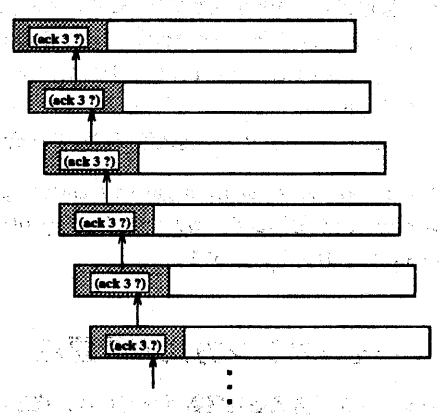

図 3: (ack 3 8) を実行したときのセグメントツリー

[2]現在の処理系では，一つのセグメントの大きさは通常 4Kbyte，引数が二つの関数呼び出しフレーム一つの大きさは 16byte である．

4 キューマシン方式へのスタック機構の導入

4.1 セグメントのスタック化

3 章で述べた問題点を解決する方法として，縦方向に連なったいくつかのフレームを，一つのセグメントにまとめて実行する方法を考える．2.3 節で述べたように，キューマシン方式並列実行におけるセグメントの構成法では，セグメント内のフレーム間に互いに依存関係はない．したがって，セグメント内のどのフレームから実行しても，得られる結果は変わらない．

ここで，セグメントの構成法に次の二つの条件を与えることにより，セグメントのスタック化を実現する．

- フレームは，セグメントの終端から，順次割り付けていく
- 実行は，必ずセグメントの先頭のフレームから行う

この 2 条件を満たすことで，セグメントを，フレームを保持するスタックとみなすことができる．

また，各プロセスごとにスタックを用意し，フレームを割り付ける際にセグメントから溢れてしまう場合は，そのスタックにフレームを割り付ける．

4.2 フレームのアロケート方法

4.2.1 キューアロケートとスタックアロケート

従来のキューマシン方式では，新しいフレームは必ず 3 つの出力セグメントのうちのいずれかに割り付けられた (2.3.3 節)．しかし，セグメントのスタック化により，新たに入力セグメントにフレームを割り付ける場合が生じる．ここで両者を区別するために，キューマシン方式に従って出力セグメントにフレームを割り付ける場合をフレームのキューアロケート，セグメントをスタックと見なして入力セグメントにフレームを割り付ける場合をフレームの スタックアロケートと，それぞれ呼ぶことにする．

4.2.2 フレームのアロケート方法の選択

セグメントのスタック化により，フレームのアロケート方法が二つ存在することになった．そこで各関数呼び出しごとに，その関数に適したアロケート

方法を選ぶことが重要である．ある関数呼び出しフレームを実行し引数を評価する際，定数引数などはそのまま値が求まる．しかし，関数呼び出しを行う引数は，新たにフレームを割り付ける必要がある．この新たにフレームを割り付けるときに，どちらのアロケート方法を選ぶかによって，フレームの実行順序が決まることになる．

ここでは，ある関数呼び出しフレームを実行したときのその引数フレームのアロケート方法を，次の条件により選択する．

- 関数呼び出しを行う引数が 2 つ以上ある
  → キューアロケート
- 関数呼び出しを行う引数が 1 つ以下
  → スタックアロケート

すなわち，フレーム割り付けを行う引数が 2 つ以上あるときは，その引数フレームはキューアロケートされる．一方フレーム割り付けを行う引数が 1 つ以下の場合，引数フレームはスタックアロケートされる．

4.2.3 スタックアロケートにおける問題点

ここで，スタックアロケートにおける問題点について述べる．

図 4 のようにフレームが割り付けられた入力セグメントを実行する場合を考える．このセグメントに入っているフレームはスタックアロケートされていて，フレーム A の値は同じセグメント内にあるフレーム X に返される．実行はセグメントの先端にあるフレームから順に行われるので，まずフレーム A が実行される．ここで，フレーム A の実行結果として次の場合が考えられる．

1. フレーム X にそのまま値を返す．
2. 新しいフレームをスタックアロケートする．
3. 新しいフレームをキューアロケートする．

1，2 の場合は正常に実行される．問題は 3 の場合である．新しいフレーム B をキューアロケートすると，フレーム B は値をフレーム X に返すことになる (図 5)．フレーム X を実行する段階では，フレーム B を実行した結果が必要である．しかし，キューマシン方式では入力セグメントにあるフレームを順次実行していくので，フレーム A のあとに実行されるのは，フレーム B ではなくフレーム X である．

結論として，実行すると引数フレームをキューアロケートするようなフレームは，スタックアロケートをしてはならない，といえる．実際にコンパイラがフレーム割り付けを行うコードを生成するときは，この条件を満たしていなければならない．

図 4: スタックアロケートされたフレーム

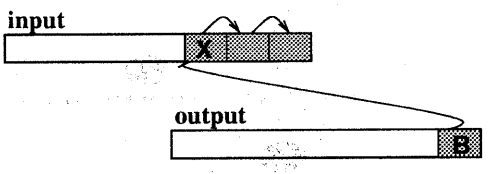

図 5: フレーム A が実行され，出力セグメントにフレーム B が割り付けられた場合

ここで，引数フレームをキューアロケートするフレームを キューアロケートフレーム，引数フレームをスタックアロケートするフレームを スタックアロケートフレームと呼ぶことにする．

4.2.4 コンパイラによるフレームアロケート方法の決定

各フレームは， 4.2.2 節で述べた条件に従ってフレームのアロケート方法を決定する．すなわち，「関数呼び出しを行う引数が 2 つ以上ある場合引数フレームはキューアロケート，1 つ以下の場合引数フレームはスタックアロケート」を行う，という条件に従いアロケート方法を決める．

さらにアロケート方法は，4.2.3 節で述べた問題が起きないように定めなければならない．すなわち，「キューアロケートフレームをスタックアロケートしない」ように，フレームのアロケート方法を決定しなければならない．これを実現するために，次の条件を加える．

- あるフレームがキューアロケートフレームと定まったならば，それを呼び出すフレームのアロケート方法もキューアロケートとする．

つまり，キューアロケートをする関数を呼び出す関数は，キューアロケートしなければならない，ということである．この条件により関数呼び出しフレームのツリーにおいて，キューアロケートを行うフレームの親にあたるフレームは，全てキューアロケート

されることになる．ここで図6を考える．図で黒まるはキューアロケートフレームを表す．図中のフレームAはキューアロケートフレームであるため，Aから上にたどっていける全てのフレーム(図で点線内にあるフレーム)をキューアロケートすることになる．

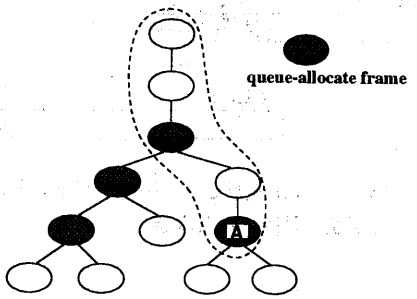

図 6: フレーム A から上にたどっていけるフレームは，全てキューアロケート

この後者の条件により，キューアロケートするフレームとその親フレームとの間で必ず同期がとられ，4.2.3 節で述べた問題は起こらなくなる．

5 実験

5.1 実験結果

セグメントのスタック化の効果を検証するために，本方式を用いた Lisp 処理系を作成し，セグメントのスタック化を行わない場合とで実行時間の比較を行った．測定に用いたプログラムは以下の通りである．

fib 35 番目のフィボナッチ数を求める．

ack Ackermann 関数 $Ack(3,8)$.

SPARCstation 20 (Solaris 2.5, 96MB, SuperSPARC 50MHz x 4CPU) 上での実行時間の測定結果を表1に示す．参考に gcc 2.6.2 および CMU CommonLisp 17f で同様のプログラムを実行した結果を共に示す。

5.2 考察

関数 `fib` は一つのタスクの粒度が小さく，並列度の上限が非常に高い．このようなプログラムは，従来のキューマシン方式による並列化に最も適したものであり，高い台数効果が得られる．本システムによる実行においても，従来のキューマシンとほぼ同

表 1: fib, ack の実行時間 (秒)

CPU 数	fib 従来版	fib stack	ack 従来版	ack stack
1 CPU	17.91	18.79	21.62	1.00
2 CPU	9.92	10.01	27.95	1.01
3 CPU	6.99	7.34	27.96	1.01
4 CPU	5.91	6.28	26.27	1.00
GCC	-	9.44	-	4.11
CMU CL	-	14.05	-	0.81

等の結果が得られており，キューマシン方式の利点をそのまま生かしたものとなっていることがわかる．

一方，関数 ack は枝分かれのない深い再帰呼び出しを行う．3 章でも述べたように，このようなプログラムは従来のキューマシン方式ではもっとも苦手とするものである．表1からわかるように，本システムでは，ack の実行効率が大幅に向上した．この結果より，セグメントのスタック化が有効に働いていることが確認できた．

6 結論および今後の展望

本稿ではキューマシン方式並列実行におけるタスク併合の手法として，複数階層に渡るフレームの併合も可能とする，セグメントのスタック化という方法を提案した．またその実現方法を示し，実際に処理系を実装し，その効果を検証した．セグメントのスタック化により，従来のキューマシン方式の利点を生かしながら，枝分かれのない深い再帰呼び出しを行うプログラムにおいても効率の良い実行が可能となった．

今後は，フレームのアロケート方法の決定について，さらに検討を加える．

参考文献

[1] Andrew W. Appel and Zhong Shao. Empirical and Analytic Study of Stack versus Heap Cost for Languages with Closures. *Journal of Functional Programming*, 6(1):47–74, 1996.

[2] 前田 敦司, 中西 正和. 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. 情報処理学会論文誌, Vol.38(3):574–583, March 1997.

[3] 前田 敦司, 中西 正和. キューマシン方式による並列 Lisp 処理系のスケジューリング手法. 電子通信情報学会論文誌, J80–D–I(7):624–634, July 1997.