マルチスレッド PaiLisp の実行方式

川本 真一　伊藤 貴康

東北大学情報科学研究科

PaiLisp/MT はマルチスレッドアーキテクチャをベースにした並列 Lisp 言語 PaiLisp の新しいインタプリタである。PaiLisp/MT は 2 つのスケジューリング方式 FCFS (first-come, first-served)、RR (round-robin) と 2 つのタスク生成方式 ETC (eager task creation)、LTC(lazy task creation) の組合せからなる 4 つの評価方式を持ち、プログラムの並列構造と粒度によってそれらを使い分ける多重評価方式を採用している。本稿では、PaiLisp/MT の 4 つの評価方式とその使い分けを説明し、共有メモリ型並列マシン DEC7000 上での評価結果について述べる。

Evaluation strategy of multi-threaded PaiLisp

Shin-ichi Kawamoto　Takayasu Ito

Department of Computer and Mathematical Sciences,
Graduate School of Information Sciences,
Tohoku University

PaiLisp/MT is a new PaiLisp interpreter based on the multi thread architecture. PaiLisp/MT equips with multiple evaluation strategy based on two scheduling policies, FCFS(first-come, first-served), RR(round-robin), and two task partitioning methods, ETC(eager task creation), LTC(lazy task creation). According to the parallel structure and the process granularity of target programs, PaiLisp/MT selects one of four evaluation strategies and executes them using it. PaiLisp/MT was implemented on a shared memory parallel machine DEC7000. In this paper we describe the multiple evaluation strategy of PaiLisp/MT and show some experimental results.

1 はじめに

PaiLisp は共有メモリアーキテクチャに基づき、豊富な並列構文を備えた並列 Scheme である[5]。PaiLisp の並列構文は **par**、**pcall**、**future**、排他的関数クロージャ、P-continuation と呼ばれる並列コンティニュエーションを捕える **call/cc**、並列 and などがあり、これらを用いて記述された PaiLisp プログラムは並列動作可能な PaiLisp プロセスを複数生成する。PaiLisp/FX と呼ばれる PaiLisp のインタプリタが共有メモリ並列マシン上で P-continuation を利用して実装されている[6]。PaiLisp/FX は FCFS スケジューリングと ETC からなる単一処理方式を採用しているが、プロセッサアイドリング問題とプロセス過剰生成問題を抱えている。

本稿はマルチスレッドアーキテクチャ上での PaiLisp の新しいインタプリタ PaiLisp/MT の実行方式と DEC 7000 上での実験結果について述べる。PaiLisp/MT は複数の評価方式を持ち、プログラムの構造と粒度に合せて適切な処理方式を選択し実行する多重評価方式を備えることで、プロセッサアイドリング問題とプロセス過剰生成問題に対処する。評価方式は 2 つのスケジューリング FCFS (first-come, first-served)、RR (round-robin) と 2 つのタスク生成方式 ETC (eager task creation)、LTC(lazy task creation) の組み合わせから構成される。処理方式の選択はプログラムの並列構造と粒度を下に行われる。木構造型並列プログラムの場合は LTC が、線形構造型並列プログラムの場合は ETC が選択され、粗粒度の場合は RR が、細粒度の場合は FCFS が選択される。プログラムの粒度はプロセスの実行時間の平均と標準偏差から見積もられる。予備的な実験では粗粒度線形構造型の場合は ETC+RR が、細粒度木構造型の場合は LTC+FCFS がそれぞれ他の方式よりも効率的であることが示されている。PaiLisp/MT の RR スケジューリングは OSF/1 によるスレッドの RR スケジューリング機能を利用し、LTC は Mul-T における実現法[7]を基に P-continuation に対応するよう修正している。

2 PaiLisp と PaiLisp/FX

本章では PaiLisp[5]について述べ、そのインタプリタ PaiLisp/FX[6]の FCFS と ETC に基づいた単一評価方式の問題点を指摘する。

2.1 PaiLisp と PaiLisp-Kernel

PaiLisp は豊富な並列構文を持つ並列 Scheme で共有メモリ型並列マシンを指向して設計された。PaiLisp-Kernel は PaiLisp の核となる小さな言語で、すべての PaiLisp の並列構文の意味は PaiLisp-Kernel によって記述することができる[5]。PaiLisp-Kernel は次のように定義される。

PaiLisp-Kernel = Scheme
+ { **spawn**, **suspend**, **call/cc**, **exlambda** }

4 つの並列構文はそれぞれ次のような意味を持つ。

(spawn e**)** e を計算するプロセスを生成する。

(suspend) それを実行したプロセスの実行を停止する。

(call/cc e**)** PaiLisp の **call/cc** は Scheme の call/cc の並列への拡張であり、残りの計算とそれが実行されていたプロセスの id を持つ一引数手続き P-continuation を生成し、それに一引数手続き e を適用する。P-continuation が起動されると、その P-continuation を捕えたプロセスが残りの計算を行う。

(exlambda $(x_1 \cdots x_n)$ $e_1 \cdots e_n$**)** 排他的に実行される関数クロージャを生成する。

PaiLisp は PaiLisp-Kernel に様々な並列構文を加えたもので、そのうちのいくつかを以下に示す。

(pcall f $e_1 \cdots e_n$**)** まず引数 $e_1,\cdots,e_n$ が並列評価され、次に式 f が評価されその評価値を引数 $e_1,\cdots,e_n$ の評価値に適用する。

(par $e_1 \cdots e_n$**)** 引数 $e_1,\cdots,e_n$ をそれぞれ実行するプロセスを生成し、それらすべての実行が終了した時点で **par** 自身の実行が終了する。

(future e**)** Halstead[2]によって Multilisp に導入された構文。**future** は式 e の future 値と呼ばれる仮の値を即返し、式 e を計算する新しいプロセスを生成する。親プロセスは e の評価値の仮の値を用いて計算を進め、その真の値が必要となった場合には、停止して e の評価が終了するのを待つ。

par-and などの他の構文に関しては[5]を参照されたい。

2.2 PaiLisp/FX

PaiLisp/FX は 8 台のプロセッサを持つ共有メモリ型並列マシン Alliant FX/80 の Concentrix OS 上に実装された PaiLisp インタプリタの実行方式である[6]。

PaiLisp/FX においては PaiLisp のプログラムは Abelson, Sussman[1] のレジスタマシン (以後 RM と略す) と呼ばれる仮想マシンによって解釈される。各 RM は PaiLisp の式を評価する度にシステム内部の関数 `eval_dispatch` を呼び出す。`eval_dispatch` は PaiLisp 式の種類を判別しその式に合った内部関数を呼び出す。

PaiLisp/FX は FCFS と ETC からなる単一処理方式を採用している。PaiLisp/FX のすべての RM は new キューと呼ばれる 1 つのキューを共有する。PaiLisp/FX

はETC(eager task creation) プロセス生成方式を採用しており、**par** や **future** などの並列構文が評価されると、引数を計算する PaiLisp プロセスが必ず生成され new キューに入れられる。また、FCFS(first-come, first-served) プロセススケジューリング方式を採用しており、新しく生成された PaiLisp プロセスは new キューに入れられた順に RM によって取り出され、それが終了するか停止するまで実行し続けられる。

2.2.1 FCFS におけるプロセッサアイドリング問題

FCFS スケジューリングの下では、RM はプロセスを実行し出すとそのプロセスがブロックされない限り終了するまで実行する。このようにプロセスを固定的に割り付ける FCFS では、プロセスのコンテクスト切り替えのオーバヘッドは小さい。しかし固定的なプロセス割り付けが RM をアイドリング化し、大きなオーバヘッドとなる場合がある。k個のプロセッサを用い FCFS の下で以下のプログラムを実行する場合を考える。

$$(\textbf{par}\ p_0 \cdots p_k)$$

par 構文が生成する式 $p_0, \cdots, p_k$ をそれぞれ実行する $k+1$ 個のプロセスは互いに独立であり、プロセス生成のコストを含む各プロセスの実行時間は同じで t であるとする。FCFS の下では、はじめ k個のプロセスが k 個のプロセッサに割当てられて実行され、それらは t 後に終了する。その後残りの 1 つのプロセスがあるプロセッサに割り当てられて実行される。FCFS スケジューリングと **par** 構文の意味から、残りの $k-1$ 個のプロセッサは実行中のプロセッサの終了を待ちアイドル状態となる。プロセスの実行時間 t が長くなればなるほど、プロセッサのアイドル時間の総和 $(k-1)t$ は大きくなる。これをプロセッサアイドリング問題と呼ぶ。

2.2.2 ETC におけるプロセス過剰生成問題

PaiLisp/FX の並列構文の実現は ETC 方式に基づいている。ETC は並列構文の評価のたびに新しいプロセスを生成する。ETC は並列構文の素直な実現法だが、プロセッサ台数に比較し多くのプロセスを生成しかつ各プロセスの実行時間が短い細粒度プログラムにおいては、プロセス生成のコストによるオーバヘッドが非常に大きい。これをプロセス過剰生成問題と呼ぶ。

プロセッサアイドルリング問題とプロセス過剰生成問題に対処するため、PaiLisp/MT はマルチスレッドアーキテクチャの基で複数の処理方式を使い分ける多重評価方式を備えている。

3 PaiLisp/MT の多重評価方式

PaiLisp/MT は共有メモリ型並列マシン上でマルチスレッド機構を用いて実現された新しい PaiLisp インタプリタである。2.2 節で述べたように、PaiLisp/FX は FCFS と ETC からなる単一評価方式を備えているが、この方式にはプロセッサアイドルリング問題やプロセス過剰生成問題がある。PaiLisp/MT は 2 つのスケジューリング方式 (FCFS,RR) と 2 つのタスク生成方式 (ETC,LTC) の組み合わせからなる多重評価方式によって、これらの問題に対処する。本章では PaiLisp/MT の 2 つのスケジューリング方式と 2 つのタスク生成方式について説明し、それらを組合わせることによって得られる PaiLisp/MT の 4 つの評価方式の効果について述べる。

3.1 PaiLisp/MT の構成

PaiLisp/FX と同様に PaiLisp/MT においても、PaiLisp プロセスは RM によって実行される。各 RM は 1 つのスレッド上に実装され、プロセッサ台数以上のスレッドが OS のスケジューラの下で FCFS 又は RR によってスケジュールされる。各 RM は LT キューと呼ばれる双方向キューを持ち、すべての RM は New キューを共有する。New キューは ETC によって生成されたプロセスを蓄えるために用いられ、LT キューは LTC の基で残りの計算や未評価の式を蓄えておくために使用される。図 1に PaiLisp/MT の構成を示す。

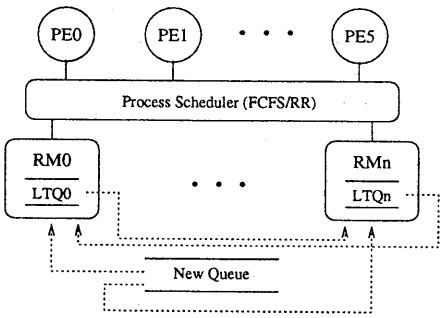

図 1: PaiLisp/MT の構成

3.2 FCFS と RR スケジューリング方式

2.2.1 で述べたように、FCFS スケジューリングは固定的にプロセスをプロセッサに割り当てるため、プロセッサ毎の処理時間がアンバランスになり、それによっ

てプロセッサアイドルリング問題を引き起す。FCFS スケジューリングのこの性質は、オペレーティングシステムの理論や実践において良く知られている。RR(round-robin) スケジューリングは、FCFS によるプロセッサのアイドリング時間を減少させる方法として良く用いられる。

RR スケジューリングは、すべてのプロセスをタイムクオンタムと呼ばれる短い時間間隔ずつ次々と実行していく方法で、プロセスは実行時間がタイムクオンタムであるような小さな複数のプロセスに分割されて実行される。この方式では、プロセッサのアイドル化によるオーバヘッドを低減することができる。しかし、RR スケジューリングはタイムクオンタム毎にプロセスのコンテクスト切り替えを行うので、そのコストがオーバヘッドとなり、また、コンテクスト切り替えに伴なうプロセスのマイグレーションコストもある。このオーバヘッドを、コンテクスト切り替えとマイグレーション問題と呼ぶ。

従って、プロセッサアイドルリング問題と、コンテクスト切り替えとマイグレーション問題とのトレードオフにより FCFS と RR のどちらを用いるべきかを決める必要がある。

3.2.1 マルチスレッド機構を用いた RR

PaiLisp プロセスの RR はオペレーティングシステムによるスレッドの RR 機構を利用して実現する。実現の詳細は 4.1 で述べる。各 PaiLisp プロセスをそれぞれ 1 つのスレッド上で実行する。スレッドの数がプロセッサ台数より多ければ、OS は各スレッドを RR の下で実行する。従って、各スレッドの上で実行されている PaiLisp プロセスも RR の下で実行される。この方法は RR を容易に実現できるが、次のような制約がある。

(1) 生成できるスレッド数には限りがある。(実装に使用したマシンでは 100 個程度)
(2) OS がユーザに提供するマルチスレッド機構では、タイムクオンタムは一定で変えることができない。(実装に使用したマシンでは 10[msec] 程度)

(1) の制約は同時に実行される PaiLisp プロセスの数を制限する。しかし、RR を導入するのはプロセッサアイドル時間を減少させるためであり、すべてのプロセスをまんべんなく実行するのが目的ではない。同時に RR によってスケジュールされるプロセスの数が増えれば、システム内部で行う共有資源へのアクセスが頻繁になり同期待ち時間が増えることを考えれば、すべてを一度に RR によって実行するのではなく、プロセスを幾つかのグループに分けそのグループ毎に RR スケジューリングを行っていく方法がより現実的で効率的であると考えられる。そこで、PaiLisp/MT の RR スケジューリングは、T個のスレッドの基で T 個のプロセス毎に RR スケジューリングを行う。最も効率的な Tは簡単な実験により 13~25 の範囲にあることが判っている。(2) の制約は、プロセスの実行時間が 10[msec] 以下であるときは、コンテクスト切り替えを行う前に実行が終了してしまうので、RR が適用できないことを意味する。

3.2.2 プロセス粒度とスケジューリング方式

FCFS と RR の使い分けは、プロセッサアイドル化問題と、コンテクスト切り替えとマイグレーション問題とのトレードオフによって決めなければならないが、さらに前節で述べた PaiLisp/MT の実装上の制約も考慮しなければならない。

プロセスの粒度が非常に細い場合は、コンテクスト切り替えやマイグレーションのコストがプロセスの実行時間に占める割合が大きく、効率が悪い。また、タイムクオンタムが 10[msec] であるので、これより短い実行時間のプロセスを RR によって実行することはできない。従って、プロセス一個の実行時間が 10[msec] 以下の細粒度プロセスの実行は FCFS によって行う。

一方、2.2.1 で述べたようにプロセスの粒度が粗ければ FCFS の基での実行はプロセッサの総アイドリング時間を長くする。またタイムクオンタムが一定であることから、コンテクスト切り替えとマイグレーションオーバヘッドも大きくなる。プロセス一個の実行時間が 10[msec] 程度であるとき、プロセッサのアイドル時間は 10[msec] のオーダであるが、これに比べて実装で用いたマシンのコンテクスト切り替えとマイグレーションオーバヘッドは無視できるくらい小さい。従って、実行時間が 10[msec] 以上の粗粒度プロセスからなるプログラムの場合は、プロセッサアイドリング問題に対処するため RR を用いる。

3.3 ETC と LTC タスク生成方式

ETC(eager task creation) は並列構文が評価される度にプロセスを生成する方式で、通常並列構文の実現方法として用いられるが、2.2.2 で述べたプロセス過剰生成問題を引き起こす。この問題に対処するために、Mul-T における **future** 構文の効率的な実現方法として提案された LTC(lazy task creation)[7] を PaiLisp/MT に導入する。LTC は木構造型並列プログラムにおいて大部分のプロセス生成を抑制するので、プロセス過剰生成問題を最小限に押さえることができる。PaiLisp に LTC を導入するには、1) PaiLisp の P-continuation と、2) **future** 以外の PaiLisp の並列構文における実現の 2 点

について考慮しなければならない。

3.3.1 PaiLisp 環境下における LTC

LTC を PaiLisp に導入するには、Mul-T(Multilisp) と PaiLisp におけるコンティニュエーションの概念の違いを考慮し、P-continuation 環境下で適切に動作するように Mul-T における LTC の実現方法[7]を修正する。PaiLisp の P-continuation は各プロセスに対して定義される。P-continuation はプロセス id と残りの計算からなり、その適用はその P-continuation が捕えられたプロセスの上で残りの計算が実行される[5]。残りの計算を実行すべきプロセスは P-continuation に含まれているプロセス id によって識別される[6]。P-continuation 適用を正しく行うには、P-continuation を捕らえた計算とプロセス id によって識別されるプロセスの関係が首尾一貫していなければならない。

LTC における (f (**future** e)) の実行は、式全体を評価したプロセッサ上で e が評価され、f は他のアイドルプロセッサに奪われて評価されるか (これをスチールと呼ぶ)、アイドルプロセッサが存在しない場合は e の実行後に e を実行したプロセッサが逐次的に評価する (これをインライン化と呼ぶ)。つまり、f がスチールされた場合には f を計算するプロセスが新たに作られ、インライン化される場合は e と f が同一プロセス上で実行されるように見える。しかし、正しい P-continuation 適用のためには、スチールやインライン化が起ってもあくまでも f は親プロセスであり、e は子プロセスとして識別する必要がある。そこで、LTC の下での **future** の実行は次のようになる。

- **future** のボディ部分 (即ち e) の実行は新しいプロセス id を生成してそのプロセス id の下で行い、いままでのプロセス id は **future** の残りの計算 (即ち f) とともに LT キューにセーブされる。
- **future** の残りの計算 (即ち f) は、インライン化される場合も別のプロセッサ上で実行される場合も、それと共に LT キューにセーブしておいたプロセス id の基で行われる。

この方法は **future** 構文が呼ばれる度に新しいプロセス id を生成するので、Mul-T よりコストがかかる。

3.3.2 future 以外の並列構文における LTC の実現

PaiLisp には **par**、**pcall**、**par-and** などの様々な並列構文があり、ユーザはプログラム構造によってそれらを使い分ける。ETC による実現では、これらすべての並列構文においてプロセス過剰生成問題が起る。これに対処するため、すべての並列構文に LTC を導入する。

LTC において最も重要な考え方は、スチールによるプロセスの分割である。プロセッサがアイドル状態になったときのみ他のプロセッサから計算を奪うことによって、プロセス生成が最小限に抑えられる。そこで、PaiLisp の並列構文における LTC は、プロセスの分割をスチールによって行う方式を基本とする。以後、**par** 構文を例として説明する。(**par** $a_1 \cdots a_k$) を評価すると、式 $a_1, \cdots, a_k$ の k 個の計算が実行可能となる。最後の式 a_k の計算を親プロセスの下で実行し、残りの式 $a_1 \cdots a_{k-1}$ は他のプロセッサがスチールできるようにキューに入れておく。アイドル状態のプロセッサは、キューから式を取り出し実行する (スチール操作)。a_k の計算が終了すると、親プロセスはキューから式を取り出して実行する (インライン化)。前節で述べた P-continuation 問題に対処するため、スチール操作やインライン化の度に新しくプロセス id を生成し、その下で実行を行う。

future 構文におけるスチールは親プロセスのコンティニュエーションつまりスタックの内容がコピーされるが、**par** の場合は未評価式だけでありコストは小さい。ただし、**par** の場合は親プロセスの再起動のための環境操作のコストがかかる。

他の並列構文における LTC も基本的な考え方は **par** と同様である。

3.3.3 プログラムの並列構造とタスク生成方式

PaiLisp/MT は ETC のプロセス過剰生成問題に対処するために LTC を導入する。LTC は木構造型並列プログラムにおいては大部分の計算がインライン化でき[7]、プロセス生成コストが大幅に抑制されるので、プロセス過剰生成問題を最小限に抑えることができる。つまり、木構造型並列プログラムの場合には LTC を用いる。

一方もう一つの基本的な並列構造である配列のような線形構造型並列プログラムは、LTC でもほとんどインライン化できず、大部分のプロセスが実際に分割される。この場合は、ETC と LTC でプロセス生成数に差がないので、並列構文の実行コストが効率の差となる。PaiLisp の LTC は、3.3.1 や 3.3.2 で述べたように新しいプロセスの実行の度に新しいプロセス id を生成し、またスチール操作に伴なうスタックのコピーや式の受け渡しなどを行うため、コストがかかる。実験から、LTC のコストは ETC のコストより大きいため、ETC と LTC の使い分けが必要となる。すなわち線形構造型並列プログラムの場合は ETC を用いる。

3.4 PaiLisp/MT の 4 つの評価方式

PaiLisp/MT の評価方式は、2 つのスケジューリングと 2 つのタスク生成方式の組合せからなる、FCFS+ETC、FCFS+LTC、RR+ETC、RR+LTC の 4 つである。ただし、FCFS+ETC はスケジューリングとして FCFS を、タスク生成方式として ETC を用いることを表しており、他も同様である。3.2.2 と 3.3.3 で述べたように、PaiLisp/MT では 4 つの方式を次のように使い分ける。

FCFS+ETC 細粒度線形構造型並列プログラム
FCFS+LTC 細粒度木構造型並列プログラム
RR+ETC 粗粒度線形構造型並列プログラム
RR+LTC 粗粒度木構造型並列プログラム

木構造型並列プログラムの場合、LTC は ETC のプロセスの過剰生成問題を最小限に抑えることができるので、FCFS+LTC や RR+LTC が他の方式に比べて効率が良い。粗粒度線形構造型並列プログラムの場合はプロセッサアイドリング問題に対処できる RR+ETC が他の方式に比べて効率が良いと考えられる。細粒度線形構造型並列プログラムの場合 FCFS+ETC が最も効率的であると考えられるが、この型のプログラムの実行時間は短いので他の方式との差はほとんどないものと思われる。

3.5 プログラムの並列構造と粒度に合せた処理方式の選択

PaiLisp/MT の 4 つの処理方式は、プログラムの並列構造と粒度に基づいて選択される。並列構造の判定は、プロセスの階層から求められる。線形の場合、一つの親プロセスがすべての子プロセスを生成するのでプロセスの関係は親と子の一階層となる。一方、木構造の場合新しく生成された子プロセスがまたプロセスを生成するため、木の深さに比例してプロセスの階層が深くなる。従って、プロセスの階層が 1 であれば線形、2 以上であれば木構造と判定する。

プログラムの粒度はプロセスの平均実行時間μと標準偏差σより見積られる。プロセスの実行時間の偏りを考慮し、$\mu+\sigma$が 10[msec] より短いとき細粒度と判定し、長いとき粗粒度と判定する。

PaiLisp/MT における処理方式の選択は、次のようなステップによって行われる。

1. プロセスの階層の深さと実行時間を計測しつつプログラムを実行する。(これを解析モードによる実行と呼ぶ。)
2. プロセス階層の深さからプログラムの並列構造が線形か木構造かを判定し、線形であれば ETC、木構造であれば LTC をタスク生成方式として設定する。
3. 実行時間の平均と標準偏差から粒度を求め、細粒度であれば FCFS、粗粒度であれば RR をスケジューリング方式として設定する。
4. 設定された評価方式の下でプログラムを再び実行。

4 PaiLisp/MT の実現

PaiLisp/MT の実装は 6 台の Alpha プロセッサを備えた共有メモリ型並列マシン DEC7000 上の OSF/1 オペレーティングシステムの基で、Pthread ライブラリを用いて行われた。PaiLisp/MT は PaiLisp/FX[6] のコードをベースに、1) スレッドによる RM の実行と RR スケジューリングへの対応、2)LTC の導入、3) プログラムの並列構造と粒度の判定機能と評価方式の選択機能の導入、をそれぞれ行うことによって実装された。

本章では 1) から 3) の各実現方法について説明する。

4.1 マルチスレッド機構を用いたスケジューリングの実現

PaiLisp/MT では、RM は OSF/1 のスレッド上で実行され (これを RM スレッドと呼ぶ)、RM スレッドはプロセッサ台数 6 より多い T 個生成され、OSF/1 によって RR スケジュールされる。各 RM はそれぞれ PaiLisp プロセスを実行するので、PaiLisp プロセスも RR スケジューリングの下で実行されることになる。アイドル状態の RM は、ETC の時は New キューを、LTC の時は他の RM の LT キューを検査し、残りの計算又は未評価式が存在すればそれを取って実行し、存在しない場合は Pthread のライブラリ関数 `pthread_condition_wait` を用いて停止する。従って、スレッド数がプロセス数より多くても、余分なスレッドが実行の邪魔をすることはない。FCFS スケジューリングは以上のように実現された RR スケジューリングの下でスレッド数 T をプロセッサ台数 6 に合せることによって実現されている。

4.2 LTC の実現

PaiLisp/MT の **future** 構文における LTC の実現は Mohr の実現方法[7] を P-continuation に対処できるように修正した。また、**future** 以外の並列構文に関しては 3.3.2 で述べた方針に従い、**future** のコードを拡張することによって実現した。各 RM はスチール可能な残りの計算 (**future** の場合) や未評価式 (その他の並列構文の場合) をストアしておくために使用される LT キュー (lazy task queue) と呼ばれる双方向キューを持つ。

4.2.1 future 構文における LTC の実現

future 構文の評価、インライン化、スチールのそれぞれは次のように実現されている。

- future が評価されると新しいプロセス id が生成され、その時のスタックポインタと古いプロセス id が組としてまとめられて LT キューにプッシュされ、future の引数の評価を行う。
- 引数の評価が終了すると、LT キューから残りの計算をポップし、取り出されたプロセス id をプロセスに付け、取り出されたスタックポインタが指し示す部分から実行を行う。
- アイドル状態になった RM は他の RM の LT キューを検査し、存在すればそれを LT キューの底から取り出し、取り出されたプロセス id をプロセスに付け、取り出されたスタックポインタが示す部分から下のすべてのスタックの内容を自分のスタックにコピーし、実行を行う。

4.2.2 par 構文における LTC の実現

par 構文の評価、インライン化、スチールのそれぞれは次のように実現されている。

- par が評価されると新しいプロセス id が生成され、最後の引数を除くすべての par の引数が環境と共に LT キューにプッシュされ、最後の引数の評価が開始される。
- 引数の評価が終了すると、新たにプロセス id が生成され、その下で LT キューから取り出した未評価式を評価する。
- アイドル状態になった RM は他の RM の LT キューを検査し、存在すればそれを LT キューの底から取り出し、新たに生成したプロセス id の下で実行する。

各未評価式の評価が終了する度に、すべての引数の評価が終了したかどうかのチェックが必ず行われ、終了していれば親プロセスの実行が再開される。

他の並列構文についてもほぼ同様に実現されている。

4.3 プログラムの並列構造と粒度の判定及び評価方式の選択の実現

プログラムの並列構造の判定は、プロセスの階層を調べることによって行われる。プロセスの階層を計るために、PaiLisp のプロセスに *depth* という変数を導入する。トップレベルプロセスの *depth* 値は 0 であり、並列構文によってプロセスが生成されると、その新しいプロセスの *depth* 値は親プロセスの *depth* 値に 1 を足した値が代入される。*depth* の最大値 Max_{depth} からプロセスの階層数が求められる。

プロセスの実行時間の計測は、システム内部における PaiLisp 式の評価関数である `eval_dispatch` の呼び出し回数から求める。`eval_dispatch` は PaiLisp の式を評価する度に必ず呼び出されるので、その呼び出し回数は実行時間にほぼ比例する。現在の PaiLisp/MT システムにおいては `eval_dispatch` 一回の呼び出しが 1/330[msec] に相当する。`eval_dispatch` の呼び出し回数を数える変数 *ecount* をプロセスに導入し、`eval_dispatch` が呼び出され度にそれをインクリメントする。各プロセスは実行終了時に *ecount* の値をファイルに書き出す。プログラムの実行終了後、書き出された *ecount* 値から実行時間の平均 μ と標準偏差 σ が計算される。

これらの計測結果からプロセスの階層値を表す Max_{depth} が 1 であれば ETC を、そうでなければ LTC を選び、プロセスの平均実行時間と標準偏差の和 $\mu+\sigma$ が 10[msec] 以下であれば FCFS を、そうでなければ RR を処理方式として選ぶ。

5 PaiLisp/MT の評価

DEC7000 の OSF/1 の Pthread を用いて実装された PaiLisp/MT の下で、4 つのベンチマークプログラムを用いて予備的な評価を行った。

5.1 ベンチマークプログラム

ベンチマークプログラムとして次の 4 つのプログラムを使用した。

upall リストの要素をすべて大文字に変換。リストの各要素毎の処理を par 構文によって並列に行う。細粒度線形構造型並列プログラム。

wcount リストの各単語の頭文字の頻度を求める。アルファベット 26 文字の各文字毎に頻度を求める処理を par 構文によって並列に行う。粗粒度線形構造型並列プログラム。

search1 2 進木の探索 (ノードの評価コストが小)。左右の枝の探索を par 構文によって並列に処理する。細粒度木構造型並列プログラム。

search2 2 進木の探索 (ノードの評価コストが大)。左右の枝の探索を par 構文によって並列に処理する。粗粒度木構造型並列プログラム。

5.2 実験結果

実験は次に示す環境の基で行った。

実行マシン	DEC7000AXP DECchip 21064 × 6
OS	DEC OSF/1 V3.0
スレッド	DEC OSF/1 Pthread ライブラリ
システム	PaiLisp/MT
スレッド数	FCFS の時 6 台、RR の時 13 台固定

実験はまず 4.3 で述べた PaiLisp/MT の解析モードの下で各プログラムを実行し、そのプログラムの並列構造と粒度を求めた。次に、4 つの評価方式の基で実行を行いその実行時間を計測した。結果を表 1に示す。

表 1: 4 つのベンチマークプログラムの実験結果

プログラム	upall	wcount	search1	search2
プロセス数	1160	26	37118	510
Max_{depth}	1	1	28	8
μ [msec]	0.98	716	0.06	75
σ [msec]	1.34	198	0.07	198
並列構造	線形	線形	木構造	木構造
粒度	細粒度	粗粒度	細粒度	粗粒度
FCFS+ETC	0.20	4.03	2.02	5.73
FCFS+LTC	0.21	4.14	0.53	5.64
RR+ETC	0.85	3.68	2.17	5.67
RR+LTC	0.21	3.79	0.59	5.60

表 1の 2 段目から 7 段目は解析モードの結果を示している。Max_{depth}はプロセスの階層数を表わし、μとσはそれぞれプロセスの実行時間の平均と標準偏差を表す。これらの結果から、6,7 段目のようにプログラムの並列構造と粒度が期待通り判定される。8 段目以降は PaiLisp/MT の 4 つの評価方式の下での実行時間を示している。ただし単位はすべて [sec] である。

これらの実験結果より次のことがいえる。

- 細粒度線形構造型並列プログラム upall の場合、差は小さいが FCFS+ETC が最も効率的となっている。upall は細粒度なので、RR でもプロセッサアイドリング問題に対処できず、線形なので LTC によるプロセス過剰生成問題の抑制効果も薄く、結局実行コストのより小さな FCFS+ETC 方式が効率的となる。
- 粗粒度線形構造型並列プログラム wcount の場合、期待通り RR+ETC が最も速い。粗粒度なので RR スケジューリングによってプロセッサアイドリング問題が抑えられ、また ETC のほうが LTC より実行コストが小さいため RR+ETC が最も速い。
- 細粒度木構造型並列プログラム search1 の場合、期待通り FCFS+LTC が最も効率的である。これは、細粒度なので FCFS が RR より効率的であり、木構造なので LTC が ETC のプロセス過剰生成問題を最小限に抑えることができるためである。FCFS+LTC と FCFS+ETC との差はプロセス生成数が増えるほど大きくなると考えられる。
- 粗粒度木構造型並列プログラム search2 の場合、期待通り RR+LTC が最も効率的である。これは、RR によってプロセッサアイドル問題に対処し、LTC がプロセス過剰生成問題を最小限に抑えるためである。

予備的な実験ではあるが、各評価方式に関しそれぞれ期待した通りの結果が得られているといえる。

6 まとめ

PaiLisp/MT の多重評価方式を構成する 4 つの評価方式について説明し、プログラムの並列構造と粒度から適切な処理方式を選択する方法を示した。予備的な実験から、各評価方式に関しそれぞれ期待した通りの結果が得られた。今回はプログラムの並列構造として線型と木構造の 2 種類を考えたが、より実際的なプログラムに適応するという観点からは、これらの 2 つを組合せた複合型並列プログラムを効率良く実行できる処理方式を考察しなければならない。

参考文献

1) H.Abelson, G.Sussman. Structure and Interpretation of Computer Programs. The MIT Press, 1985.
2) R.Halstead,Jr. Implementation of Multilisp: Lisp on a multiprocessor. ACM Symp on Lisp and Functional Programming, PP 9-17, 1984.
3) R.Halstead,Jr., T.Ito. "Parallel Symbolic Computing: Language, Systems and Applications, US/Japan Workshop", Springer LNCS 748, 1993.
4) T.Ito, R.Halstead,Jr. "Parallel Lisp: Languages and systems, US/Japan Workshop on Parallel Lisp ", Springer LNCS 441, 1990.
5) T.Ito, M.Matsui. A parallel lisp language PaiLisp and its kernel specification. Springer LNCS 441, 58-100, 1990.
6) T.Ito, T.Seino. P-continuation based implementation of PaiLisp interpreter. Springer LNCS 748, 108-154, 1993.
7) E.Mohr, D.A.Kranz, R.Halstead,Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. IEEE Trans. Parallel and Distributed systems, Vol.2, No.3, 264-280, 1991.