プログラミング－言語・基礎・実践－ 18－13
（１９９４．７．２１）

# マルチスレッドを用いたPaiLispインタプリタの実現と評価

川本 真一　伊藤 貴康

東北大学 情報科学研究科

マルチスレッド機能の使用を前提とした効率の良いPaiLispインタプリタの処理方式として、粒度適応型処理方式を考案し、それを用いてPaiLispインタプリタPaiLisp/MTを試作しその評価を行った。粒度適応型の処理方式では、粒度に合せてプリエンプティブスケジューリングまたはノンプリエンプティブスケジューリングのどちらかを選択し、またノンプリエンプティブスケジューリングの下で、強制タスク生成法(ETC: Eager Task Creation)または負荷分散法(LBP: Load Based Partitioning)のどちらか選択し、効率良く実行する。本稿は、粒度適応型処理方式の概要と、それを用いたPaiLisp/MTの実現と評価について述べる。

# Multi-threaded Implementation of PaiLisp Interpreter and Its Evaluation

Shin-ichi KAWAMOTO　Takayasu ITO

Depertment of Computer and Mathematical Sciences
Graduate School of Information Sciences
Tohoku University

For efficient implementation of PaiLisp interpreter using multi-thread mechanism, we propose a method of granuality adaptive processing and implemented PaiLisp interpreter PaiLisp/MT using this method. To execute parallel programs effectively, this system will select preemptive scheduling or non-preemptive scheduling and eager task creation(ETC) or load based partitioning(LBP) adapting to granuality of the programs. This paper explains the method of granuality adaptive processing, the design and implementation of PaiLisp/MT with its experimental evaluations.

## 1 はじめに

PaiLisp は Scheme をベース言語とする共有メモリ型並列 Lisp 言語である[1]。並列マシン Alliant FX/80 上でのインタプリタ PaiLisp/FX[2]が作成され実用に供されているが、スレッド機構を備えた並列 UNIX マシン上でのマルチスレッドを用いた効率の良い PaiLisp インタプリタ PaiLisp/MT を設計、試作し、評価を行ったのでその概要を報告する。

PaiLisp/FX では、PaiLisp プロセスを実行の単位とし、PaiLisp プロセスが停止または終了になるまで実行をし続けるノンプリエンプティブスケジューリング方式を取っているため、(1) 粒度の大きいプログラムでは、プロセッサが有効に利用されず効率が悪い。また、並列構文に対して常に PaiLisp プロセスを生成する ETC(Eager Task Creation) 方式を取っているため、(2) 細粒度プログラムでは、多くの PaiLisp プロセスが生成され、生成のオーバーヘッドが蓄積されるため、実行効率が悪い。

Multilisp においても同様な問題があるが、Concert Multilisp や Mul-T では細粒度プログラムに対して負荷分散法 (LBP: Load Based Partitioning) や遅延タスク生成法 (LTC: Lasy Task Creation) で対応している[3]。

この論文では、マルチスレッド機構を備えた並列UNIX マシン上での実現を前提として、次のような特徴を有する粒度適応型処理方式を考え、PaiLisp インタプリタ PaiLisp/MT の設計・試作を行った。

粒度適応型処理方式

1) 粗粒度プログラムに対しては、スレッド機構の特徴である高速なコンテクスト切り替え機能を生かし、PaiLisp プロセスを少しずつ平等に実行するプリエンプティブスケジューリングによって実行する。2) 中粒度プログラムに対しては、従来通りノンプリエンプティブスケジューリングによって実行する。2) 細粒度プログラムに対しては、ノンプリエンプティブスケジューリングの下で、システムの負荷によってプロセス生成を行うか否かを決定する負荷分散法 (LBP) を適用する。

すなわち、実行するプログラムの粒度に合せて、プリエンプティブスケジューリングとノンプリエンプティブスケジューリングを選択し、ノンプリエンプティブスケジューリングの下では、ETC と LBP を使い分ける。

## 2 PaiLisp とその処理系

2.1 で PaiLisp について述べ、2.2 で PaiLisp/FX の処理方式とその問題点について述べ、2.3 でスレッド機構を用いた粒度適応型の処理方式について説明する。

### 2.1 PaiLisp

PaiLisp[1]は共有メモリアーキテクチャを前提として Scheme に様々な並列構文を導入して設計された並列 Lisp 言語である。PaiLisp の主な並列構文を以下に示す。

(spawn $e$) $e$ を評価する子プロセスを生成する。
(suspend) 適用したプロセスの実行を停止させる。
(exlambda ($x_1$ .. $x_n$) $e_1$ .. $e_m$) 排他的関数クロージャを作る。PaiLisp プロセス間の相互排除に利用される。
(call/cc $proc$) Scheme の継続を並列に拡張した P-continuation を生成し、それを引数として一引数関数 $proc$ の適用を行う。
(par $e_1$ .. $e_n$) $e_1,..,e_n$ の評価を並列に行う。
(future $e$) future 値と呼ばれる $e$ の仮の値を即座に返し、$e$ の値を計算する子プロセスを生成する。
(pcall $f$ $e_1$ .. $e_n$) $e_1,..,e_n$ の評価を並列に行なった後、関数 $f$ を適用する。
その他 par-and,par-or,pcond 等がある。

### 2.2 PaiLisp/FX の処理方式

PaiLisp の実用的な処理系として、8 台のプロセッサを持つ共有メモリマシン Alliant FX/80 の Concentrix OS 上で動作するインタプリタ PaiLisp/FX[2]がある。また、PaiLisp のためのオブジェクトシステム PaiObject も設計試作されている[5]。PaiLisp/MT は PaiLisp/FX のプロセス概念及び、仮想マシンをベースとして作成されているので、PaiLisp/FX の処理方式とその問題点について述べる。

PaiLisp/FX

PaiLisp/FX における並列実行の単位である PaiLisp プロセスとは、プロセス ID、実行状態、現在の値、残りの計算、排他資源の占有状況の 5 つの要素より構成される逐次 Lisp オブジェクトである。PaiLisp プロセスは図 1のように状態遷移する。

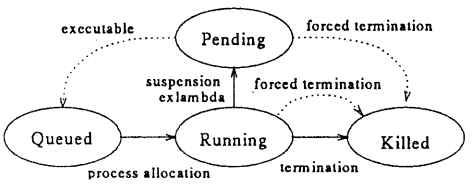



図 1: PaiLisp プロセスの状態遷移

Queued は実行可能であるがまだ実行されていない状態、Running は実行中。Pending は suspend や exlambda 等によって停止した状態。Killed は終了状態である。PaiLisp プロセスは spawn や future などの並列構文によっ

て生成される。

PaiLispプロセスはレジスタマシン[6](以後RMと略)と呼ばれる仮想マシン上のインタプリタによって実行される。一つのRMはConcentrix OSのUNIXプロセスとして実現されており、このUNIXプロセスがプロセッサと同数生成されて、それぞれのプロセッサで並列に動作する。RMはすべてのRMに共有されたFIFO方式のnewキューとそのRMに固有のLIFO方式のresumeキューを管理する。PaiLisp/FXの構成を図2に示す。

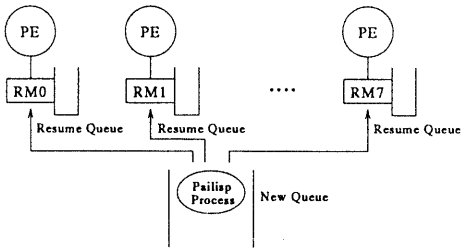



図2: PaiLisp/FXの構成

新しく生成されたPaiLispプロセスはnewキューに入れられ、停止中のプロセスが実行可能になるとresumeキューに入れられ、それぞれRMの割り当てを待つ。RMは、1)resumeキュー、2)newキュー、の順でプロセスキューを探索し、PaiLispプロセスが存在したらそれを取って実行する。一端実行を開始すると、それがPending又はKilled状態になるまで実行し続けるノンプリエンプティブスケジューリングを採用している。現在実行中のPaiLispプロセスがPending又はKilled状態になると、先に述べた順でプロセスキューを探索する。PaiLisp/FXにおける粗粒度プログラムと細粒度プログラムの並列実行を4つのプロセッサ(PE)を使用した場合について説明する。

**粗粒度プログラムの場合**

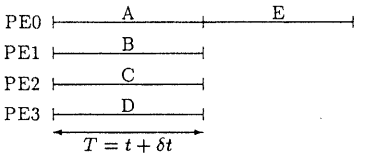
粗粒度プログラムとして次を考える。

```
(par (A) (B) (C) (D) (E))
```

式(A)~(E)は互いに独立で、それぞれ実行時間は$t$である場合を考える。PaiLisp/FXはPaiLispプロセスのスケジューリングをノンプリエンプティブに行うので、実行の様子は図3のようになり、(A)~(D)が同時に実行された後に(E)が1つ実行されて終了する。全体としては$2T$の実行時間がかかる。(E)が実行されている最中は他の3つのRMは実行すべきプロセスが無く遊んでしまい、プロセッサ資源を有効に活用できない。

**細粒度プログラムの場合**

細粒度プログラムはPaiLispの並列構文の適用回数の多いプログラムであるということができる。並列構文によるPaiLispプロセスの生成には必ずコストがかかる。PaiLisp/FXでは並列構文に対して常にPaiLispプロセスを生成するETC(Eager Task Creation)方式を用いているので、細粒度プログラムの場合は生成されるPaiLispプロセス数が非常に多く、オーバヘッドが多数積算されて実行効率が悪い。




($\delta t$はPaiLispプロセス生成のオーバヘッド)

図3: PaiLisp/FXによる粗粒度プログラムの実行

### 2.3 スレッド機構を用いたPaiLisp/MTの処理方式

マルチスレッド機構を備えた並列UNIXマシンの使用を前提として、PaiLisp/FXでの問題を解決する一方式として考えられたのが、粒度適応型の処理方式である。

従来のUNIXプロセスのコンテクスト切り替えのコストに比べ、スレッドのコンテクスト切り替えのコストは非常に小さい。このような特徴を持つスレッド機構を用いることを前提としたPaiLisp/MTの粒度適応型処理方式は次のような動作をする。

(1) 粗粒度プログラムに対しては、スレッド機構の特徴である高速なコンテクスト切り替え機能を生かして、PaiLispプロセスを少しずつ平等に実行するプリエンプティブスケジューリングによる実行を行う。先の粗粒度プログラムの例では、(A)~(E)をそれぞれ$T/4$ずつの部分A1~A4,..,E1~E4に分け、その部分を図4のように公平に実行すると、すべてのRMを有効に動作させることができ、処理時間も$5T/4$とPaiLisp/FXの$2T$に比べ短い。

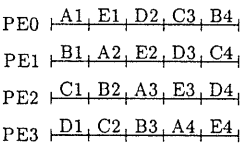



図4: PaiLisp/MTによる粗粒度プログラムの実行

(2) 中粒度プログラムに対しては、PaiLisp/FXが採用していたノンプリエンプティブスケジューリングによって実行を行う。

(3) 細粒度プログラムに対しては、スケジューリングは

ノンプリエンプティブ方式で行い、システムの負荷によってプロセス生成を行うか否かを決定する負荷分散法(LBP)を適用する。これにより、PaiLispプロセスの生成数が減り、従ってPaiLispプロセス生成のオーバヘッドの積算量が減り、効率的な実行を行うことができる。

スケジューリングを粗粒度の場合とその他の場合で使い分ける理由としては、中粒度から細粒度プログラムでは多くのプロセスが生成されるため、プリエンプティブスケジューリングの下ではPaiLispプロセスのコンテクスト切り替えの回数が多い。従って、PaiLispプロセスのコンテクスト切り替えのオーバヘッドが積算され、ノンプリエンプティブスケジューリングよりもかえって遅くなるからである。また、LBPの適用を細粒度の場合に限るのは、粗粒度から中粒度のプログラムのプロセス生成の抑制は、かえって実行効率を落すためである。

粒度の判定とスケジューリングの切り替え

粒度適応型処理方式での粒度は生成されたPaiLispプロセスの数によって判断する。すなわち、プロセス生成数が少なければ粗粒度、多ければ細粒度、中間なら中粒度であるとする。PaiLisp/MTは、入力されたプログラムをまず粗粒度であると見なしプリエンプティブスケジューリングによって実行する。生成されたPaiLispプロセスの数を数えるカウンタ$pc$を持っており、PaiLispプロセスが生成されるたびにそれをインクリメントする。$pc$の値が閾値$N_1$より大きくなると、スケジューリングをノンプリエンプティブ方式に切り替えて実行を続ける。さらに、$pc$が閾値$N_2$より大きくなると負荷分散法(LBP)を適用する。$pc$はプログラムの実行が終了した時点で0にリセットされる。一端ノンプリエンプティブスケジューリングやLBPによる実行を始めると、LBPの適用をやめたり、プリエンプティブスケジューリングに戻ることはない。

$pc$の値はプロセス生成と共に徐々に増加するため、中粒度から細粒度プログラムでも$pc$の値が$N_1$以下のときは、プリエンプティブに実行される。しかし、生成されるプロセスの数が大きければその影響は少ないものと考えられる。また、LBPの適用についても$pc$が$N_2$以下のときは、ETCによって実行され、その後LBPが適用される。このようにプログラムの先頭からではなく遅延してLBPを適用する方法を遅延負荷分散法(DLBP: Delayed Load Based Partitioning)と呼ぶ。後で述べるように、DLBPはLBPよりも優れた性質を持つ。

## 3 PaiLisp/MTの実現

本章では、まず3.1でスレッドによるスケジューリングの実現方法について、3.2で遅延負荷分散法とその実現について述べ、3.3では2つのスケジューリングを理論的に解析する。

### 3.1 スレッドを用いたスケジューリングの実現

仮想マシンRMのスレッドによる実現

スレッド機構は、従来のUNIXのプロセスをその実行環境であるタスクと実行の流れであるスレッドに分割したものであり、スレッドの環境の管理はタスクが行うため、UNIXプロセスに比べて、その生成や切り替えが速いという特徴がある。そこで、PaiLisp/FXではUNIXプロセスとして実現されていたRMをスレッドによって実現し[1]、さらにこの高速な切り替え機能を生かして、PaiLispプロセスのコンテクスト切り替えをスレッドの切り替えによって実現した。PaiLisp/MTのノンプリエンプティブスケジューリングは、PaiLisp/FXのUNIXプロセスとして実現されていたRMをスレッドとして実現し、それをプロセッサと同数生成して、各プロセッサ上で実行させることによって実現される。

プリエンプティブスケジューリングの実現

プリエンプティブスケジューリングは、PaiLispプロセスのコンテクスト切り替えを行う。切り替えにはそのPaiLispプロセスの実行に関する情報を保存する必要がある。RMはPaiLispプロセスの実行環境そのものであり、その実行の情報をすべて持っている。そこで、PaiLispプロセスを直接切り替るのではなく、それを実行しているRMを切り替る方式を採用した。RMはスレッドとして実行されるため、RMのコンテクスト切り替えはスレッドのコンテクスト切り替えとして実現される。あらかじめ生成されるプロセスの数よりも多くのRMスレッドを用意しておく。各RMスレッドはOSによって少しずつ切り替えられながらPaiLispプロセスをキューから取り出して実行する。コンテクスト切り替えはOSが自動的に行う。

スケジューリングの切り替えの実現

プリエンプティブ方式とノンプリエンプティブ方式のシステム上の違いは、RMスレッドの数がプロセッサ数より大きいか同じかというということである。従って、プリエンプティブからノンプリエンプティブへの切り替えは、RMスレッドのうちのいくつかを停止させ、その数をプロセッサ数と同数にすれば良い。ただし、停止させたRM上でかつて実行されPending状態になっていたPaiLispプロセスの実行が再開される場合に備え、そのRMが持つresumeキューにプロセスが入ったと

[1] PaiLispプロセスを1つのスレッドによって実行するPaiLisp/LWPの試作も行われているが、PaiLispプロセスの生成の度にスレッドが生成され、そのオーバヘッドのためあまり効率が良くない。そこで、今回の試作ではこのような方法を用いた。

きだけは再起動して処理を行う。このような機能を持ったRMは次のように実現されている。

```
 1: while (1) {
 2:   mutex_lock(rqlock);
 3:     process = resume_queue_take();
 4:     if ( process != Nil ) {
 5:       mutex_unlock(rqlock);
 6:       resume_proc(process); }
 7:     else if ( rm_sleep_flag ) {
 8:       condition_wait(sleepcd,rqlock);
 9:       mutex_unlock(rqlock); }
10:     else {
11:       process = new_queue_take();
12:       if ( process != Nil ) {
13:         mutex_unlock(rqlock);
14:         mew_proc(process); }
15:       else {
16:         condition_wait(sleepcd,rqlock);
17:         mutex_unlock(rqlock); }}}
```

RMはまずresumeキューを捜し(3行目)、PaiLispプロセスが存在すれば実行する(6行目)。そうでなければ、スケジューリング切り替えのフラグを確認し(7行目)、そのフラグが立っていればcondition変数を用いて停止する(8行目)。そうでなければnewキューを捜し(11行目)、プロセスが存在すれば実行する(14行目)。2つのキューに存在しない場合には実行すべきプロセスが無いのでcondition変数を用いて停止する(16行目)。resumeキューやnewキューにPaiLispプロセスが入ると、condition_signalによってRMの実行が再開される。

### 3.2 遅延負荷分散法(DLBP)とその実現

**負荷分散法を導入する並列構文**

LBPは並列構文に対しシステムの負荷によってPaiLispプロセスを生成しない場合もある。従って、複数のプロセスが副作用によって通信しながら計算を進めていくようなプログラムの場合、プロセス生成が行なわれないとデッドロックに陥いる可能性がある。そこで、LBPを導入するのは副作用の使用を前提としないfutureとpcallについて行うのが妥当であると考えられる。このうち今回はfuture構文について行った。

**LBPの処理方式**

LBPの実現は、newキューの中のPaiLispプロセスの数によって負荷を判定する方法を採用した。つまり、newキューに入っているプロセスの数がある閾値$T$より大きいとき、(future $e$)は$e$をそのまま実行し、それ以下のときは$e$を計算するPaiLispプロセスを生成する。プロセス生成を最小限に抑えるため、閾値$T$は0とした。ただし2.3で述べたように、PaiLisp/MTでは、粗粒度や中粒度プログラムに対するLBPの適用を避けるため、生成されたプロセスの数が閾値$N_2$を越えた時点からLBPの適用を始める遅延負荷分散法(DLBP)を用いる。

**遅延負荷分散法(DLBP)の特徴**

DLBPはLBPに比べ、生成されるプロセスの粒度が揃っており、PaiLispプロセスの生成数が少ないという特徴がある。例えば、図5のような2進木データを再帰的に並列に処理する場合を考える。

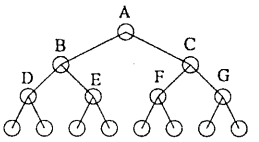



図5: 二進木

各接点で、future構文によって右側の枝を計算するプロセスを生成し、親は左側の枝の計算を続ける。LBPの場合は、実行開始時のnewは空なので、接点Aではプロセスが生成される。次に親がBに行ってfutureを実行しようとするが、Aで生成された左の枝を計算するPaiLispプロセスがすでに他のRMによって実行されていれば、Bでもプロセスは生成される。しかし、それがまだキューに溜っていればBでのプロセス生成は起らない。その他のノードも同様である。つまり、LBPではタイミングによって分割される場所が変る。DLBPで例えば$N_2 = 3$のとき、接点ABCではETCによって必ずプロセスが生成され、それ以下のノードではLBPが適用される。2進木は上位ノードほど処理コストが大きいため、上位ノードをバランス良く分割することが望ましい。この点で、タイミングによらないDLBPはLBPに比べて優れている。また、DLBPは上の方の接点を均等に分割するため処理が均一化され、結果としてPaiLispプロセスの生成数がLBPよりも少ない。

**遅延負荷分散法(DLBP)の実現**

future構文におけるDLBPは次のように実現される。

```
1: if ( N2<pc && newq_length!=0 )
2:     go(eval_dispatch);
3: else
4:     go(eagaer_future);
```

現在までに生成されたプロセスの数$pc$が閾値$N_2$より大きく(1行目)、newキューの中にプロセスが存在すれば、futureの引数をそのまま評価するために、PaiLispの評価ルーチンである`eval_dispatch`[2]に飛び、そうでなければ常にプロセスを生成する従来のfuture構文を実現する`eager_future`ルーチンに飛ぶ。

### 3.3 スケジューリングの理論的考察

プリエンプティブ方式とノンプリエンプティブ方式の比較を理論的に行い、スケジューリングの切り替えの閾値の決定法を示す。次のような互いに独立した $job_1$ から $job_n$ を並列に実行するプログラムを考える。

(par ($job_1$) ($job_2$) ··· ($job_n$))　　··· PRG1

ここでプロセッサ数を $N_p$、RM 数を $N_r$ であるとする。またプリエンプティブスケジューリングにおける RM の 1 回の切替にかかるコストを $T_o$、一回のスライスの間隔を $T_s$ とする。

**job の粒度が揃っている場合**

すべての job の実行時間を $T$ であるとする。ノンプリエンプティブ方式による実行時間 $T_{nps}$ は、プロセス数 $n$ が $N_p$ 以下ならば $T$、$N_p < n \leq 2N_p$ なら $2T$,··· であるから、次のように近似される。

$$T_{nps} \approx \lceil \frac{n}{N_p} \rceil T$$

次にプリエンプティブの場合を考える。$1 \leq n \leq N_p$ のとき RM スレッドのコンテクスト切り替えは起らないので、実行時間は $T$ である。$N_p < n < N_r$ の場合、一回のタイムスライスにおける実行時間はオーバヘッドを含むため $T_s + T_o$ である。すべてのプロセスのタイムスライス回数の和は $\lceil nT/N_pT_s \rceil$。これを $N_r$ 個の RM で実行するため、実行時間 $T_{ps}$ は次のように近似される。

$$T_{ps} \approx \begin{cases} T & (1 \leq n \leq N_p) \\ \lceil \frac{nT}{N_rT_s} \rceil (T_s + T_o) & (N_p < n < N_r) \end{cases}$$

ここで、$T_{ps}$ を最も小さくするには $T_s = T/N_p$ とすれば良い。つまり、プリエンプティブスケジューリンの元では見かけの粒度に比べ、実質的な実行時の粒度を $T/N_p$ として実行するとき最も高速に実行される。しかし一般に実行時間 $T$ を予測することは困難であり、また RM の切り替えを OS にまかせるため $T_s$ はある一定の値に決まる。従って、job の実行時間が長くなると切り替えの回数が多くなって効率が落る。2 つの式 $T_{nps}$ と $T_{ps}$ をグラフにすると図 6 となる。縦軸は $T = 1$ としたときの実行時間、横軸はプロセス数 $n$、$N_p = 4$、$T_s = T/4 = 1/4$、$T_o = 1/40$ であるとした。実線がノンプリエンプティブ方式、直線がプリエンプティブ方式である。プロセス数が少ないときプリエンプティブ方式のほうが速いが、傾きがノンプリエンプティブより急なためプロセス数が多くなるとかえって遅くなる。従って、スケジューリング切り替えに関し次が導かれる。

<スケジューリング切り替えの閾値の決定法>

プリエンプティブ方式がノンプリエンプティブ方式に比べて高速なのは生成されるプロセスの数が閾値 $N$ より小さいときであり、この値は実験結果から図 6 のような図を書くことによって求められる。実験結果から求めた閾値 $N$ を、スケジューリング切り替えの閾値 $N_2$ として用いる。

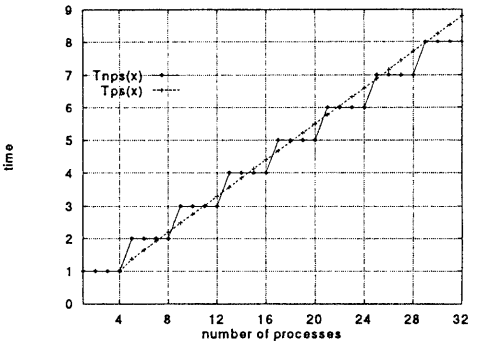



図 6: 実行時間の比較 (粒度が揃っている場合)

**job の粒度が揃っていない場合**

job1 が他に比べて大きい場合、そのグラフは図 7 のようになる。

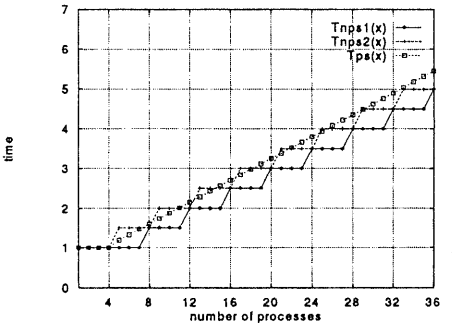



図 7: 実行時間の比較 (粒度が揃っていない場合)

ただし縦軸は長い job の実行時間を 1 としたときの実行時間であり、長い job を基準とし、短い job はその 1/2、タイムスライスの間隔はその 1/8、切り替えのオーバヘッドはその 1/80 であるとした。ノンプリエンプティブ方式の場合は長い job の実行される場所によって実行時間が変る。図 7 の下側の階段状の実線は長い job が先頭で実行された場合であり、上側の階段状の点線は長い job が一番最後に実行された場合である。プリエンプティブ方式は点線の直線であり長い job の実行場所によって変化しない。このようにプロセスの粒度が揃っていない場合は、ノンプリエンプティブ方式のほうが高速に実行される場合もある。

## 4 PaiLisp/MT の評価

PaiLisp/MT の実現は OMRON LUNA88k の MACH 2.5 上の C-Thread ライブラリを用いて行った。3 章でも述べたとうり、PaiLisp/MT は 2 つのスケジューリングを閾値 $N_1$ によって切り替え、DLBP の適用を閾値 $N_2$ によって制御するので、まずこれらの閾値を予備的な実験によって決定する。4.1 ではプリエンプティブ方式とノンプリエンプティブ方式の切り替えの閾値 $N_1$ を求め、4.2 で DLBP の適用開始の閾値 $N_2$ を定める。そして最後にそれらの値を取り入れた PaiLisp/MT システムの評価を行う。

### 4.1 スケジューリング切り替えのための閾値の決定

切り替えの閾値 $N_1$ を決定するために、プリエンプティブスケジューリングを採用した PaiLisp/MTp と、ノンプリエンプティブスケジューリングを採用した PaiLisp/MTnp を作成し、実行時間の比較を行った。評価には、3 章の理論的解析に用いたプログラム PRG1 を job1 = job2 = ⋯ = jobn とし、また job の数を 1 個から 24 個まで変えて実行しその実行時間を計測した。実験結果を横軸を job(プロセス) の数、縦軸を実行時間とするグラフによって表すと図 8 となる。

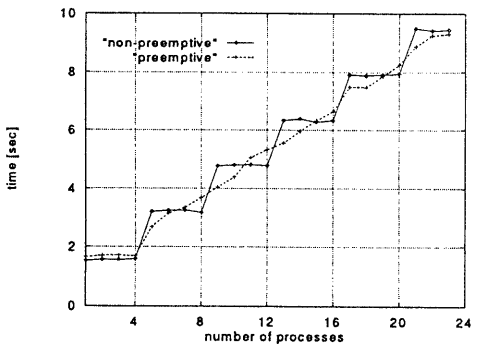



図 8: PRG1 の実行時間の比較

実線はノンプリエンプティブ方式、点線はプリエンプティブ方式である。PaiLisp プロセスの数が $n$ における実行時間 $T(n)$ を区間 $4i \le n < 4(i+1)$ $(i = 1, 2, ..)$ において平均し、その値の大小関係が逆転する区間 $i$ は 4 と求まるので、切り替えの閾値 $N_1$ は 16 程度ということになる。

### 4.2 DLBP のための閾値の決定

DLBP が適用されるのはスケジューリングがノンプリエンプィブの場合であるので、4.1 節で試作した PaiLisp/MTnp を用い、その future 構文に DLBP を導入して実験を行った。細粒度プログラムとしてはフィボナッチ 20 を並列に計算する ffib を、中粒度プログラムとしては並列化クイックソート qsort を用いた。それぞれ、ETC による実行では 10945 個と 485 個の PaiLisp プロセスが生成される。実験結果を表 1 に示す。

表 1: DLBP の閾値決定のための実験結果

| 閾値 ($N_2$) | ffib | qsort |
|---|---|---|
| 16 | 1.71(101) | 2.40(116) |
| 100 | 1.72(110) | 2.41(156) |
| 500 | 1.79(528) | 2.34(485) |

各欄の数字は実行時間 [sec] であり、カッコ内は生成された PaiLisp プロセスの数である。細粒度プログラムは PaiLisp プロセスの生成をなるべく抑えるため、閾値は小さいほうが良い。この傾向は ffib の結果に現れている。一方中粒度プログラムはなるべくプロセス生成を抑えたくないので、閾値は大きいほうが良い。この傾向は qsort にあらわれている。結局この相反する 2 つの条件の中間を取って、その閾値 $N_2$ を 100 と決定した。

### 4.3 PaiLisp/MT の評価

先の実験結果より、スケジューリングの切り替えの閾値を 16、DLBP 適用開始の閾値を 100 として、PaiLisp/MT を実現した。PaiLisp/FX 方式の PaiLisp インタプリタを OMRON LUNA88k 上で実現し、それと PaiLisp/MT との比較を行った。3 章で述べたプログラム PRG1 を使って粗粒度から中粒度にかけての実行時間を計測し、それをグラフにした (図 9)。

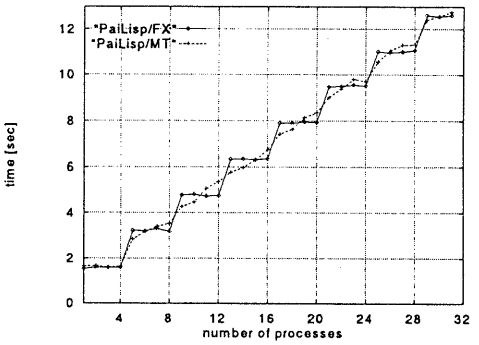



図 9: PaiLisp/MT の評価 1

ただし、縦軸は実行時間 [sec]、横軸は PaiLisp プロセス数である。実線は PaiLisp/FX 方式、点線が PaiLisp/MT を表している。プロセス数が 16 以下の時は、PaiLisp/MT の平均実行時間が PaiLisp/FX 方式の平均実行時間よりも短く、効率的な実行になっているとい

える。ただし、PaiLisp プロセス数が 16 から 28 付近の間では PaiLisp/FX 方式のほうが速い。これは PaiLisp/MT がはじめの 16 個のプロセスをプリエンプティブに実行し、その後残りのプロセスをノンプリエンプティブに実行するためである。しかしプロセス数が増えるにつれてその影響は少なくなっていく。

次に、中粒度から細粒度プログラムの実行時間を比較する。評価プログラムは 4.2 で使用した ffib と qsort の他、8 クイーン問題を並列に解く queen、future によるストリームを用いて並列に素数を求める prime の 4 つを使用した。プロセッサは 4 台使用した。実行結果を表 2 に示す。

表 2: PaiLisp/MT の評価 2

| | ffib | queen | qsort | prime |
|---|---|---|---|---|
| 逐次 | 6.31 | 21.40 | 7.24 | 12.53 |
| PN | 0 | 0 | 0 | 0 |
| PaiLisp/FX 方式 | 3.28 | 6.54 | 2.33 | 3.86 |
| PN | 10945 | 5508 | 485 | 2498 |
| PaiLisp/MT | 1.72 | 5.86 | 2.41 | 3.88 |
| PN | 110 | 178 | 156 | 2499 |

各項目の上段は実行時間 [sec]、下段は生成される PaiLisp プロセスの数である。細粒度プログラムの ffib や queen は、DLBP を採用した PaiLisp/MT が速い。特に ffib は粒度が細かいためその効果が大きく、約 2 倍の高速化を達成している。同じ細粒度プログラムでも prime は実行時間、プロセス生成数ともに PaiLisp/FX 方式との差がないが、これは生成された PaiLisp プロセスが実行されたときにのみまた PaiLisp プロセスが生成されるプログラムであり、future 実行時は常に new キューが空なので DLBP の効果が現れない。このような場合でも PaiLisp/MT は PaiLisp/FX 方式より遅くなることはない。中粒度プログラムの qsort の場合は、PaiLisp プロセスの生成が抑制されるので、PaiLisp/MT のほうが遅くなってしまう。

以上の結果から、プロセス数が 16 よりも小さい粗粒度プログラムとプロセス数が 1000 を越えるような細粒度プログラムの場合、概ね PaiLisp/MT は PaiLisp/FX 方式に比べて効率的に実行できるが、中粒度プロセスの場合は逆に多少遅くなる結果となった。

## 5 おわりに

PaiLisp/MT の処理方式と、マルチスレッド機構を用いた実現及び評価について述べた。粗粒度プログラムのプリエンプティブスケジューリングによる実行は、ノンプリエンプティブスケジューリングに比べて、プロセスの粒度が揃っている場合は良い結果を得ることができるが、そうでない場合には返って遅くなることを示した。今後、優先度などを用いることによってこれに対処することが課題である。

細粒度プログラムに対しては、LBP を修正した DLBP によって ETC よりも効率良く実行できることを示した。ただし、LBP にはデッドロックの可能性があるため、今後 LBP よりもさらに効率的でデッドロックフリーの LTC の導入について検討したい。

PaiLisp/MT の実現には MACH の C-Thread を用い、プリエンプティブスケジューリングのプロセス切り替えをスレッドの切り替えによって実現した。この方法はコンテクスト切り替えのメカニズムを OS 側にまかせるため実現が非常に容易であり、また性能もそれほど悪くない。ただし現在はプロセッサ数が 4 台であり、それに対してプリエンプティブスケジューリングが適用できる範囲は 16 個程度となったが、プロセッサ数がこれより増えた場合には、より切り替えのオーバヘッドが小さな機構を使用しないと効率的なプリエンプティブスケジューリングを実現できない。今後 SUN の LWP や OSF/1 の P-Thread による実現を試みたい。

## 参考文献


1) Takayasu ITO Manabu MATSUI. A parallel lisp language PaiLisp and its kernel specification. In *Parallel Lisp: Languages and Systems US/Japan Workshop on Parallel Lisp*, number 441 in LNCS. Springer-Verlag, 1989.

2) 清野 智弘 伊藤 貴康. Pailisp の並列構文の実現法と評価. 情報処理学会論文誌, 34(12), 1993.

3) Eric Mohr David A. Kranz and Robert H. Halsted Jr. Lazy task creation: A technique for increasing the granuarity of parallel programs. Number Vol 2, No. 3 in IEEE TRAN. Parallel and Distributed systems, July 1991.

4) Suresh JagannathanJim Philbin. A foundation for an efficient multi-threaded scheme system. *ACM lisp and functional programming*, 6, 1992.

5) 田村 清朗 伊藤 貴康. Pailisp と PaiObject の処理系. 情報処理学会記号処理研究会, 93-SYM-70, 1993.

6) Harold Abelson Gerald Jay Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, 1985.