記 号 処 理 59－7
（1991. 3. 11）

マルチプロセッサUnixマシンにおけるmutilispの実現

岩崎 英哉

東京大学工学部計数工学科

本稿では，マルチプロセッサUnixマシン上において真の並列処理システムを実現する際の問題点をいくつか指摘し，実際の実現例として，並列Lispの一方言mutilisp処理系の試作について述べる．従来のUnixではシステムコールの重さ，カーネルの機能そのもの，機能面の細かさに問題がある．試作したmutilisp処理系では，プロセス管理のためのキューをプロセッサ毎に保持し，プロセス管理の上位部を，個々のプロセッサに密着した局所スケジューリングと，システム全体にわたる大域スケジューリングに分けることによって，これらの問題点に対処した．また，若干の実験結果についても簡単にふれている．

mutilisp on a Multiprocessor Unix Machine

Hideya Iwasaki

Department of Mathematical Engineering and Information Physics,
Faculty of Engineering, University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo, 113, Japan

In this paper, we discuss some characteristic problems with the multiprocessor Unix operating system when implementing real parallel programming systems. An implementation of the mutilisp system, which is a parallel Lisp dialect, on a multiprocessor Unix machine is also proposed as a concrete solution. To meet the problems, each processor has its own queue of Lisp processes along with the special Lisp process (local scheduler) which locally manages each queue. The local schedulers also cooperate with other special Lisp process (global scheduler) for higher level process management such as load balancing. Finally, some experimental results are also discussed.

1. はじめに

筆者らは並列処理機能を言語仕様として持つ Lisp 方言 mutilisp[1,2] の処理系を，Sun3 などの Unix マシン上で疑似並列版として実現してきた [3]．最近は汎用マルチプロセッサ Unix マシンが一般化してきたことにより，疑似並列ではない“真の”並列処理システムを構築する環境が整いつつある．このような状況に対応し，並列言語の実現法を検討し実際に試作・実験をおこなうことには大きな意味がある．

最近，複数のプロセッサを持つ汎用 Unix(系) マシン上で作動する mutilisp システムを試作したので，本稿ではこの処理系について，Unix 上で実現する際の問題点等をまじえて報告する．

本稿では，Unix および Unix 系の OS をまとめて Unix と呼ぶこととし，その上の (通常の意味の) プロセスを Unix プロセスと呼ぶ．一方，mutilisp の中の並列動作のひとつひとつを Lisp プロセスと呼び，Unix プロセスと明確に区別する．また，マルチプロセッサマシン上に実現した mutilisp 処理系は，4.4. 節で述べるように二種類のバージョンがあるが，これらをまとめて (特に区別する必要がない場合)“本処理系”とかく．

また，“真の並列”と“並列”のふたつの言葉を以下のように使い分ける．“真の並列”は複数の計算が異なるプロセッサで (物理的に) 並んで進行していく場合に用いる．一方“並列”は複数の計算が (物理的・論理的にかかわらず) 並んで進行してしていく場合に用いる．したがって，並列は真の並列を含むことになり，また，プロセッサひとつによる疑似並列は，並列の一種ではあるが真の並列ではない．

2. mutilisp の概要

本節では，mutilisp の概要 [4] を簡単に示す．mutilisp は Utilisp に並列処理を導入・拡張して開発したもので，Lisp プロセスを Lisp オブジェクトとして扱っている．Lisp プロセスは動的に生成することができ，これにより Lisp プロセス間に親子関係が生じる．

mutilisp で最も重要な特徴は，複数の Lisp プロセス間での Lisp オブジェクトの意味的な共有を極力排除し，同時に環境 (シンボルと値・関数定義・属性との対応) の共有も排除した点である．各 Lisp プロセスは独自のシンボル空間をもつことにより，他 Lisp プロセスのシンボルとの衝突を回避し，独自の環境を保持する．Lisp プロセスの環境は，以下のような仕組みによって形成される．

- 環境の初期値は親プロセスのものを生来の環境として継承する．
- シンボルへの値・関数定義の設定などは，その Lisp プロセス内部の環境を変更し，そのシンボルに関する生来の環境を隠す．

Lisp プロセス間通信は，共有の排除に適合したメッセージ伝達によっておこない，そのための関数として `send`(送信)，`receive`(受信) がある．`receive` で受信すべきメッセージがない場合，`receive` を呼び出した Lisp プロセスはブロックする．

疑似並列版の mutilisp は，MC68000 系プロセッサのための，一種のマクロアセンブリ言語 [5] で記述されており，Sun3 などの上で作動している．この処理系は高速であり，MC68000 系と類似した命令セットをもつプロセッサへの移植も比較的容易であるが，Sparc の様なプロセッサへの移植には向いていない．今回の試作では，使用したマシンの CPU は MC68020 ではあるものの，将来の拡張・移植等を考慮し，C 言語による Utilisp/C[6] 処理系を基とし，これに並列処理部を追加・拡張して記述した．

3. マルチプロセッサの Unix における問題点

本節では，マルチプロセッサ計算機の Unix 上で mutilisp などの真の並列処理システムを実現する際に生じる問題点についていくつか考察する．ここでの議論においては，各々の並列動作はある種のデータを共有しつつ実行を進めることを前提とする．本稿では並列 Lisp 処理系を扱っているが，本節における議論は他の言語 (たとえばグラフリダクションを並列におこなう関数型言語システム) においても同様に通用する．

3.1. プロセッサへの計算の割り当て

汎用マルチプロセッサ Unix マシンには，プロセッサが複数個あることをユーザには意識させないような OS の設計のものが少なくない．このような OS では，プロセッサへの計算 (Unix プロセスなど) の割り当てはカーネルが面倒をみるので，ユーザはプロセッサをその都度指定する必要がない．しかし，真の並列処理システムを実現する場合には，このような“皮”はあまり有難くない．

一方最近では，ひとつの Unix プロセスの中での複数の並列動作 (thread, lightweight process[7] など

と呼ばれる) を記述するためのライブラリを提供するシステムもある．しかし複数の thread が同時に“実行中”となりうることが仕様にきちんと含まれていなければ，真の並列処理システムへの利用には不向きである．

真の並列処理システムを実現するためには，ユーザの指定したプロセッサに計算を割り当て，複数の計算を同時に実行中とすることができるような機能が求められる．その割り当ての最小単位としては，

- Unix プロセス;
- thread;

が考えられる．ただし後者の場合は，OS によっては，従来の thread の定義を変更しカーネル自身が thread を扱うようにする必要がある．

3.2. 相互排除

並列に進む複数の計算が共有データを操作する際には相互排除の問題が生ずる．従来の mutilisp のように，単一のプロセッサ上で疑似並列実行をおこなうシステムでは，Lisp プロセス間のコンテクスト切り替えをおこす場所・タイミングを限定する等の手段により，この問題をある程度回避することができた．

マルチプロセッサの場合は，ロック / アンロックを用いて，きわどい領域の相互排除をおこなわざるを得ず，その実現効率はシステム全体の性能に影響を与える．相互排除の実現法としては，

- セマフォ関連のシステムコール semop の利用;
- 機械語レベルの命令 tas(test and set) の利用;

がある．Unix のシステムコールは相当重い仕事であるので，前者は効率的・実用的な実現とは言い難い．ただし，ロック獲得に失敗した計算をカーネルの中でブロックさせることができるので，ビジーウェイトによる CPU の浪費は避けることができる．

これとは逆に，後者は高速ではある反面，ロック獲得の失敗の際の対処が難しい．ひとつの方法は，プロセッサを放棄することであるが，この時には，次の 3.3. 節で述べる問題が生じる．

3.3. プロセッサの放棄

上で述べたように，相互排除のためのロック獲得で失敗した場合，実行していたプロセッサでは，ロックの獲得以外にやるべきことがないことがある．また，計算を進めていく過程で，あるプロセッサにおいて，実行すべき Lisp プロセスが全くなくなることもある．たとえば，実行可能な Lisp プロセスの数がプロセッサ全体の数より少なくなった場合，そのプロセッサが実行していた Lisp プロセスがすべて (メッセージ受信などで) ブロックした場合，などがこれに当たる．

このような時にそのプロセッサにおける計算は，最終的にはロックが獲得できるまで，あるいは，実行可能な Lisp プロセスが出現するまで待たなければならないが，マルチユーザで使用する計算機環境では CPU を放棄して待つのが望ましい．また他のプロセッサは，共有データのロックを解除した時，ある Lisp プロセスを実行可能にした (たとえばメッセージ待ちの Lisp プロセスにメッセージを送信した) 時などに，CPU を放棄した計算に対してシグナル等でその事実を知らせる必要がある．

Unix においては，“CPU を放棄して待つ”および“知らせる (シグナルを送る)”ことは共にシステムコールを用いておこなう．しかし，シグナルを送る場合，上で述べたような状況ごとにシステムコールを呼ぶのは，3.2. 節で指摘したように効率面を考えると現実的ではない．また，各プロセッサの実行のタイミングによっては，シグナルが相手に伝わらない可能性がある．たとえば，実行可能キューを各プロセッサで共有している場合を考える．実行可能キューが空の時に CPU を放棄して待つ部分を，

```
1:  lock(ready_queue);
2:  while ( ready_queue == EMPTY ) {
3:    flag = 1;
4:    unlock(ready_queue);
5:    pause();
6:    lock(ready_queue);
7:    flag = 0;
8:  }
```

と実現し，実行可能キューへプロセスを入れる部分を，

```
9:  lock(ready_queue);
10: enqueue(some_lisp_process,
                    ready_queue);
11: if ( flag == 1 )
12:   sendsig();
13: unlock(ready_queue);
```

と実現したとする．ここで，pause はシグナルを受信するまでプロセッサを放棄する関数，sendsig はプロセッサを放棄している計算にシグナルを送る関数である．このとき，プロセッサを放棄しようとした計算の 4 行目と 5 行目の間で，別のプロセッサでの計算が 9 行目から 12 行目を実行してしまうと，最初

の計算にはシグナルが伝わらないことになる．これは，Unixのシステムコールで不可分に実行される部分が，真の並列処理に利用するには機能面において細かすぎることに原因がある．(ただしこの例の場合，最初のシグナルは即座に伝わらなくとも，別プロセッサで発した“二回回以降の”シグナルは伝わる．)

また，system V系のUnixではシグナルの受信によってシグナルのハンドラがリセットされてしまうので，ハンドラの再設定までの間に隙が生じる．という問題点もある．

したがって，プロセッサの放棄をインタプリタのレベルで実際に無駄なく実現するのは難しい．結局，

- インタプリタではCPUを放棄せずにビジーウェイトで待つ;
- シグナルが即座に伝わることはあきらめ，上述のプログラムのように，“一回遅れ”のケースがあってもやむを得ないものとする;

とするのが，現実的・実用的な策であろう．ただし，マルチユーザの環境下でビジーウェイトを用いる場合は，浪費を少なくするように，ロックを減らしたりスケジューリングを工夫すべきである．一方後者は，二回目以降のシグナルの発生するタイミング，すなわち，実現する処理系でのプロセッサ横取り(preemption)の間隔に，性能が大きく依存する．また，両者を組合せる(最初の適当な期間だけはビジーウェイトする)，という解決法が適当な場合もある．

4. mutilispでの実現方式

ここでは，前節での議論を踏まえて試作したmutilisp処理系の実現法について述べる．今回は二種類のバージョンを作成したが，これらを第一版，第二版と呼ぶことにする．両者はプロセス管理の方法が異なるが，詳細は4.4.節で述べる．

4.1. ハードウェアとOS

今回我々が用いた計算機は，Charles River Data Systems社のUniverse400[8]である．これは以下のようなハードウェアの特徴をもつ，共有メモリ型の密結合マルチプロセッサマシンである．

- CPUはMC68020(16.67MHz)+MC68881;
- 2個のCPUを搭載(最大4個まで拡張可能);
- 各CPUに8キロバイトのキャッシュ;
- 16メガバイトの共有メインメモリ.

オペレーティングシステムはUnosというものであるが，基本的な機能としてはリアルタイムのsystem V系Unixである．(したがってUnosのプロセスについてもUnixプロセスという言葉を使う．) 3.1.節で述べたように，Unosは，マルチプロセッサであることをユーザが意識しなくともすむ設計になっている．それでも，マルチプロセッサに関連した以下のような特徴がある．

- 各Unixプロセスには CPU mask という属性が付随しており，これはその Unix プロセスが実行されうるプロセッサを指定する．デフォルトではすべてのビットが on になっており，これはどの CPU で実行してもよいことを意味する．
- それぞれのプロセッサ に対して，そのプロセッサ上でのみの 実行を (CPU mask によって) 要求しているプロセスのみを実行する (“専用”状態) か否か (“非専用”状態) かを指定できる．はじめは，すべてのプロセッサが非専用状態である．

前者は実行される対象である Unix プロセス側からみた属性であり，後者は，計算を実行するプロセッサ側からみた属性である．前者の CPU mask を操作するのが，set_cpu_mask システムコールである．(他に，CPU mask を得るための get_cpu_mask システムコールもある．) 後者の設定をおこなうシステムコールもあるが，本処理系ではこれを利用していない．

4.2. 並列実行の指定

上で述べたように，Unosではプロセッサへの計算の割り付けは，Unixプロセスの単位でのみユーザに提供されている．したがって，必要な回数だけ子Unixプロセスを生成することによって，各プロセッサ上で別々のUnixプロセスとしてmutilispインタプリタを動かし，各インタプリタは共有メモリによって情報を共有することとした．具体的には，以下のような手順になる．

1. シェルから起動されたmutilispインタプリタは，shmgetシステムコールにより，必要な量の共有メモリを確保し，各種初期設定をおこなう．
2. プロセッサ数よりひとつ少ない数だけ子Unixプロセスを生成(fork)する．
3. 各子Unixプロセスは，親の確保したメモリを共有し，さらに必要な初期設定をおこなう．
4. 各Unix プロセスは (親子共に) set_cpu_mask システムコールにより自分の担当するプロセッサに貼り付き，トップループに入る．

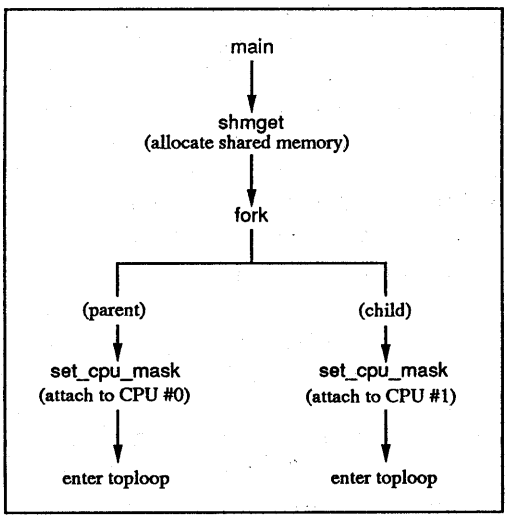

図 1: プロセッサへのインタプリタの割り当て

今回の試作ではプロセッサが 2 個なので，子 Unix プロセスはひとつだけ生成する (図 1)．各 Unix プロセスのトップループでは，実行可能 Lisp プロセスのキュー (以下単に“実行可能キュー”と呼ぶ) の先頭から Lisp プロセスとひとつ取り出してはそれを実行する，ということを繰り返す．

4.3. 情報の共有と相互排除

mutilisp では動的に Lisp プロセスを生成することができ，Lisp プロセスのプロセッサへの割り付けを静的に決定することは困難である．そのため，ユーザが生成したすべての Lisp プロセスは，すべてのプロセッサ上において実行可能である必要がある．結局，“Lisp オブジェクト (ポインタ)” がすべてのプロセッサで同じ意味を持たなければならないので，本処理系では，ヒープ領域をどのプロセッサからもアクセスできる共有メモリにおくことにした．ただし，ヒープの割り付けのためのポインタも共有するかどうかは選択の余地がある．今回の処理系ではこれも共有したが，ヒープを分割するなどしてプロセッサ間の干渉を軽減する実験もおこなう必要があろう．

本処理系において，相互排除の対象となるのは，

- ヒープからの割り付け;
- 特定のプロセスへのメッセージの送信;
- プロセス生成時のスタックの割り付け;
- 実行可能キューへのアクセス (第一版のみ);

である．mutilisp は，Lisp オブジェクトの共有を排除しているので，シンボルへの値などの設定，コンスセルのかきかえ等は，相互排除をする必要がない．これは，mutilisp の言語仕様が効率的な処理系の実現に向いていることを意味する．

相互排除を実現するのにシステムコールを用いるのは，3.2. 節で述べたように実行効率をそこなうので，本処理系ではこの部分のみ tas 命令を用いたアセンブリ言語で記述した．3.3. 節で説明した通り，ロックの獲得に失敗した場合にプロセッサを放棄するのは難しい．そこで，

- 実行可能キューにある別の Lisp プロセスを走らせるべくコンテクストを切り替える;
- ロックが獲得できるまでビジーウェイトする;

などの方法を考えたが，本処理系では，ひとつの Lisp プロセスがロックを獲得している時間はさほど長くないので，ビジーウェイトで待つようにしている．

4.4. スケジューリング

疑似並列版の mutilisp 処理系では，プロセス管理を上位・下位のふたつに分けることによって，システムの自由度を高めていた．上位レベルでは，スケジューラと呼ばれる特別な Lisp プロセスが，スケジューリングを含むプロセス管理をおこなう．スケジューラのプログラムは Lisp で記述されており，ユーザが機能を自由に拡張・変更できる．下位レベルは，Lisp インタプリタによる，実行可能キューを用いたラウンドロビン方式のスケジューリングである．

本処理系の第一版としては，上述の疑似並列版のプロセス管理をそのままマルチプロセッサにあてはめたものを実現した．この場合，すべてのプロセッサは同一の実行可能キューを共有し，相互排除をしてこのキューにアクセスする．また，スケジューラはこのキューの管理を含む，全体的なプロセス管理をおこなう．

一方，実行可能キューへのアクセスは頻繁に起こるので，これをプロセッサ毎に保持することによって，(疑似並列版と同様にして) アクセスの相互排除による性能の低下を防ぐ事ができると期待される．そこで第二版の処理系では，プロセッサ毎に別々の実行可能キューを持つことにした．ただし，計算が進むにつれて，プロセッサによってキューの長さ (実行可能 Lisp プロセスの数) に少なからぬ差がでることもありうる．

第二版の場合，スケジューラひとりですべてのプロセッサの実行可能キューを管理することはできない．そこで，上位レベル (Lisp プログラムのレベル)

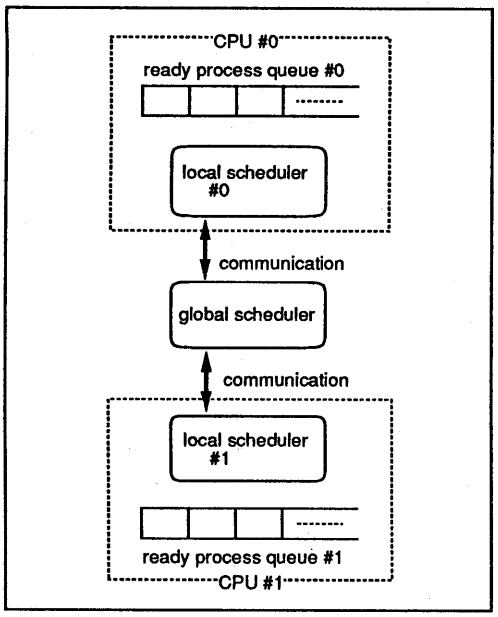

図 2: 第二版におけるスケジューラの分割

のプロセス管理をさらにふたつの階層，すなわち大域的なスケジューリングと局所的なスケジューリングに分割し，それぞれを担当する Lisp プロセス (ひとつの大域スケジューラとプロセッサと同数の局所スケジューラ) を用意した (図 2).

局所スケジューラは，自分の担当するプロセッサがあらかじめ定められており，そのプロセッサ上でのみ実行される．局所スケジューラの役割は，担当プロセッサに付随する実行可能キューの管理を主とした，局所的な Lisp プロセス管理である．その意味で，局所スケジューラと担当プロセッサの関係は，疑似並列版のスケジューラと (唯一の) プロセッサの関係に類似している．また局所スケジューラは，大域スケジューラおよび他プロセッサの局所スケジューラと協力して，プロセッサ間の負荷分散もおこなう．負荷分散の例として，あるプロセッサ P_i の実行可能キューが空になった場合の挙動を以下に示す (図 3).

1. mutilisp インタプリタは，P_i の局所スケジューラ S_i に対して実行可能キューが空になったことをあらわす内部的なメッセージ (シンボル `no-ready-process`)[3] を届けて，S_i を実行可能状態にする．
2. S_i が上記の内部的メッセージを受け取ると，大域スケジューラ G に対して，負荷分散の依頼を意味するシンボル `get-process` を送信する．
3. G は適当なプロセッサ P_j $(j \neq i)$ を選び，それに付随する局所スケジューラ S_j に対して，実行可能 Lisp プロセスを分けるように依頼する意味のシンボル `get-process` を送信する．
4. S_j は P_j の実行可能キューを操作して，そこにある Lisp プロセスから (あれば) 適当なものを選び G に送信する．
5. G はこれを受け取るとそのまま P_i に送信する．
6. P_i は受け取った Lisp プロセスを実行可能キューにつなげる．

図 3 では，i 番目のプロセッサの実行可能キューが空になった時，j 番目のプロセッサが自分の実行可能キュー (p1 ～ p4 の 4 個の Lisp プロセスがある) から 2 個 (p1 と p2) を分け与えた例を示している．

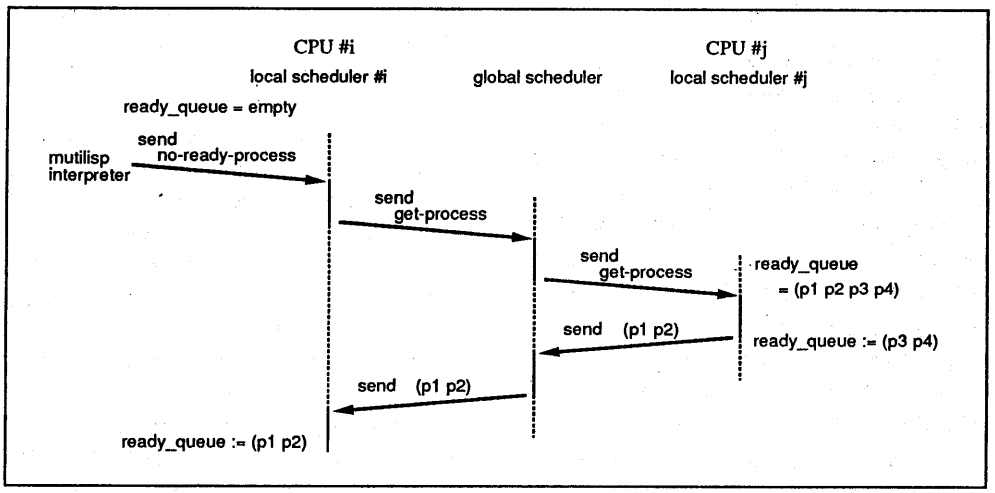

図 3: 実行可能キューが空の時の負荷分散

表1: tarai-n の実行時間 (単位: 1/50 秒)

	疑似並列版	第一版	第二版
tarai-4	220 (100%)	226 (103%)	226 (103%)
tarai-5	5961 (100%)	6161 (103%)	6189 (104%)

表2: n-queen 問題の実行時間 (単位: 1/50 秒)

	疑似並列版	第一版	第二版
(queens 7)	368 (100%)	384 (100%)	399 (100%)
(pqueens 7) (ワーカ数2)	473 (129%)	317 (82.6%)	322 (80.7%)
(pqueens 7) (ワーカ数3)	475 (129%)	320 (83.3%)	318 (79.7%)
(queens 8)	1793 (100%)	1852 (100%)	1899 (100%)
(pqueens 8) (ワーカ数2)	2127 (119%)	1274 (68.8%)	1216 (64.0%)
(pqueens 8) (ワーカ数3)	2130 (119%)	1322 (71.4%)	1240 (65.3%)

この方法の利点は，負荷分散の戦略をいろいろと変えることができる点にある．上の説明中“適当に”と書いた部分がこれにあたるが，たとえば，特定の問題に適したスケジューリングをおこなうこと等が考えられる．

5. 実験結果

ここでは，前節で述べた2種類の処理系と疑似並列版の処理系，すなわち，

- 単一の (共有された) 実行可能キューを用いる処理系第一版;
- プロセッサ毎に異なる実行可能キューを用いる処理系第二版;
- 上の両処理系と同様に Utilisp/C を基に作成した，プロセッサをひとつのみ使う疑似並列版の処理系;

のインタプリタについて，若干のプログラムを (コンパイルせずに) 実行して性能を測定した結果を示す．

疑似並列版および第一版の測定ではスケジューラを起動せず，スケジューラの影響を除いた．また，第二版における局所スケジューラは，大域スケジューラから負荷分散の依頼 (get-process) がくると，

- 自分の実行可能キューが空ならば Lisp プロセスを分け与えることができないので，nil を返事とする;
- そうでなければ，実行可能キューの先頭にある Lisp プロセスを返事として分け与え，残りを実行可能キューに設定しなおす;

という単純なものにした．

最初の実験は，(tarai 2n n 0) をひとつだけ逐次的に計算するプログラム tarai-n である．この結果を表1に示す．

この結果をみると，マルチプロセッサに対応するためのオーバーヘッドは，疑似並列版と比較して数%ですんでいることがわかる．

次の実験は，n-queen 問題である．queens はこれを逐次的に解くプログラム，pqueens は並列に解くプログラムである．なお pqueens は，マスター—ワーカ (master–worker) 形式のプログラム [9] である．これでは，あらかじめマスタ Lisp プロセスといくつかのワーカ Lisp プロセスを生成しておいて，並列計算はマスタを経由してワーカに分配する．ワーカは，与えられた計算を終えると結果を返し，次にやるべき仕事をマスタから受け取る，ということを繰り返す．マスタへの計算の依頼・マスタからワーカへの計算の分配・計算結果の回収はメッセージ送受信によっておこなう．queens と pqueens のプログラムの構成は基本的には同じであり，異なるのは，queens では組み込みの map 関数を用いるところの一部を，pqueens ではマスタを経由してワーカに仕事を分配し結果を回収する並列 map 関数を用いる，という点である．

n-queen 問題の結果を表2に示す．ここで，マスタとワーカの生成に要する時間は pqueens の実行時間には含まない．

第一版と第二版の実行時間の違いは，実行可能キューへのアクセスの相互排除のコストと，大域・局所両スケジューラによる負荷分散のコストとの差が主

な要因である．pqueensの結果を見ると，$n=7$では両者の実行時間にはさほど差がないが，$n=8$では第二版の方が若干良い．また，ここでの測定においては大域・局所スケジューラのプログラムはコンパイルしていないが，これをコンパイルすることによって負荷分散に要する時間はさらに減少する．またプロセッサ数がさらに増えると，相互排除を減らした第二版の方が有利である．以上を考慮すると，解く問題・負荷分散の戦略にも依存するが，第二版のプロセス管理が第一版と比較してまさっていると言うことができる．

6. おわりに

本稿では，マルチプロセッサUnixマシン上での真の並列処理システム実現の問題点を議論し，さらに具体例として，Universe400上で試作したmutilisp処理系について紹介した．

UnosではUnixプロセスを特定のプロセッサに割り当てることが一般ユーザに可能であったので，試作・実験は比較的楽におこなうことができた．しかし3節で述べたように，Unixは真の並列を実現するには，システムコールの重さ，機能の不足，不可分に実行される機能が細かい，といった面がある．

試作した処理系については，プロセス管理・スケジューリングでは，キューを分散して保持し，Lispプロセスのレベルでのプロセス管理を局所スケジューラと大域スケジューラに分けるという(第二版の)方針の性能がほぼ確認できた．このように実現することは，マルチプロセッサUnixマシンにおけるマルチユーザ環境下でのプロセッサ資源の節約に結びつく．また，疑似並列版でのプロセス管理の自然な拡張でもあり，今後疎結合型マルチプロセッサに対応する場合の出発点にもなりうる．

今後，GCの方式についての検討，(特にプロセッサ数が増えた場合について)更なる実験をおこなうと同時に，疎結合型マルチプロセッサ上における並列Lispの本格的な実現にも取り組んでいきたいと考えている．

参考文献

[1] 岩崎: Lispにおける並列動作の記述と実現，情報処理学会論文誌，Vol.28，No.5，pp.465–470 (1987).

[2] Iwasaki, H: mutilisp: A Lisp Dialect for Parallel Processing, *Proc. US/Japan Workshop on Parallel Lisp (Lecture Notes in Computer Science 441)*, Springer-Verlag, pp.316–321 (1990).

[3] 岩崎，寺田，湯浅: Unix上で作動するmutilispシステム，記号処理研究会研究報告，48–3 (1988).

[4] Iwasaki, H.: *mUtilisp Manual*, METR 88–11, Department of Mathematical Engineering and Information Physics, University of Tokyo (1988).

[5] Wada, E.: History of UtiLisp Hacking, *J. Inf. Process.*, Vol.13, No.3, pp.276–283 (1990).

[6] 田中: Utilisp/Cのインタプリタ，記号処理研究会研究報告，53–3 (1989).

[7] *SunOS System Services Overview*, Sun Microsystems, Inc. (1988).

[8] *VCP-4000MP User's Manual*, Charles River Data Systems, Inc. (1987).

[9] Carriero, N and Gelernter, D: How to Write Parallel Programs: A Guide to the Perplexed, *Comput. Surv.*, Vol. 21, No.3, pp.323–357 (1989).