マルチプロセッサ Unix マシン上における並列言語処理系の実装法の検討†

岩崎 英哉††

本論文では，並列処理言語の処理系を Unix 系共有メモリ型マルチプロセッサ計算機上に実装する際に直面するいくつかの問題点について議論する．ここでは，マルチユーザの下で CPU 資源を使う環境を想定する．さらに，並列処理言語の実際の実装例を示し，考察も加える．実現例を示すのは，並列性を言語仕様として持つ Lisp 方言 mutilisp である．従来の Unix では，システムコールの重さ・機能そのものや機能面の細かさ，並列性記述のためのライブラリが単一プロセッサ向きである点などが，マルチプロセッサにおける処理系作成上の問題点となる．mutilisp ではインタプリタ本体で効率的に解決するのが困難な問題については，これをインタプリタ上で動いている Lisp プログラムに通知する機構を導入し，そのプログラムに処置を委ねる，という解決法をとる．その結果，問題解決のためには，上位部の特定のプログラム間の関係のみに注目すればよくなり，たとえば CPU の放棄も容易に実現でき，処理系の柔軟性が向上することになる．また，本論文では，処理系の性能についても簡単にふれ，mutilisp でのアプローチが有効であることを示した．本論文で提案する手法は，実行時の動的スケジューリングをおこなうシステムで有効であり，多くの言語処理系に対して応用が可能である．

1. はじめに

最近，汎用マルチプロセッサマシンが一般に使われるようになってきたことにより，真の並列処理言語処理系を構築する環境が整いつつある．それらのオペレーティングシステム (OS) としては，相変わらず Unix☆系を採用しているものが数多い．

単一プロセッサかマルチプロセッサかにかかわらず，多くのシステムソフトウェアの内部には並列性が内在する．これを効率よく記述・実行するのを支援するために，ひとつの Unix プロセスの中での複数の並列動作（スレッド，軽量プロセスなどと呼ばれるが，本論文では“スレッド”という名で総称する）を記述するためのライブラリ[1)~3)]も最近では一般的に用いられるようになってきた．本論文ではこのような状況をふまえ，並列処理言語の処理系を Unix 系共有メモリ型マルチプロセッサ計算機上に実装する際に直面する問題点について議論し，実際の実現例を示し，考察を加えることを目的とする．ここでは，マルチユーザで計算機を利用するような環境を想定している．

本論文で示す実現法が有効なのは，実行時における動的スケジューリングをおこなうシステムである．このような言語システムのひとつの例として，本論文では解釈実行を基本とする並列記号処理言語に特に注目する．記号処理言語ではごみ集め (GC) を避けることはできず，特に並列 GC は重要な研究課題であるが，本論文のスコープには含めない．実装例としてとりあげる言語は，並列処理機能を言語仕様として持つ Lisp 方言 mutilisp[4)~6)]である．

本論文では，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（Mach[7)]でいえばタスクに相当する）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して用いる．

また，並列処理言語処理系を，“下位部”と“上位部”のふたつのレベルに分けて考える．下位部とは，処理系の“核”ともいえる部分で，たとえばCなりアセンブリ言語などの記述言語で実現されている部分をさす．mutilisp でいえば，インタプリタ本体に相当し，この部分は，ユーザによる変更は不可能である．上位部とは，並列処理言語自身による“ライブラリ”などの形態でユーザに提供されている部分をさす．mutilisp でいえば，mutilisp 自身で記述されシステムで用意されている Lisp プロセス(群)が，これに相当する (4.3 節参照)．

本論文では，まず，第2章で mutilisp の特徴，および今回の実装にあたって使用したマシンの特徴を簡単に紹介する．第3章では，mutilisp のような並列処

† Implementation of Parallel Processing Language System on a Multiprocessor Unix Machine by HIDEYA IWASAKI (Department of Mathematical Engineering and Information Physics, Faculty of Engineering, University of Tokyo).

†† 東京大学工学部計数工学科

☆ Unix は米国 AT&T Bell 研究所の商標である．

理言語を共有メモリ型マルチプロセッサ Unix マシンに実装する際の問題点をいくつか提起し，一般的な解決法などを含めた議論をおこなう．第4章では，mutilisp の実装例を示し，前章で提起した問題点への対処法を論じる．第5章では，実装した処理系についての性能評価を試みる．第6章では，第3章から第5章まで述べたことについて，考察を加える．

2. 作業環境

2.1 mutilisp の概要

mutilisp は Utilisp[8),9)] に並列性を導入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陽に扱う．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 オブジェクトの共有を極力排除し，同時に環境（シンボルと値・関数定義等との対応）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）と receive（受信）がある．receive において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．

ここで注意すべきことは，“共有”には

- 言語仕様のレベルでの共有；
- 処理系実現のレベルでの共有；

のふたつのレベルが存在する点である．mutilisp で排除するのは，前者のレベルでの共有である．後者については処理系依存であるが，本論文で述べる共有メモリ型マルチプロセッサ上での実現では，プロセッサ間である種のデータ（後述）を共有している．

本論文で示す mutilisp の試作では，C言語による Utilisp/C[10)] 処理系を基とし，これに並列処理部を追加・拡張したインタプリタとして記述した．本処理系では，実行可能（実行待ち）状態の Lisp プロセスを，キュー（“実行可能キュー”と呼ぶ）により管理する．インタプリタのメインループでは，実行可能キューの先頭から Lisp プロセスを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（4.1節参照）．

2.2 ターゲットマシンのハードウェアと OS

われわれが用いた計算機は，Charles River Data Systems 社の Universe 400[11)]である．これは以下のような特徴のハードウェアをもつ，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 MC 68020 (16.67 MHz)＋MC 68881；
- 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 と呼ばれており，その基本的な機能はリアルタイムの 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．）Unos には，マルチプロセッサに関連して以下のような特徴がある．

- 各 Unix プロセスには“CPU マスク”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（番号）を指定する．デフォルトではすべてのビットがオンになっており，これはどのプロセッサ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 マスクによって）要求しているプロセスのみを実行する（“専用”状態）か否（“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 マスクを設定するのが set_cpu_mask システムコールであり，mutilisp の実装で利用している．

3. Unix 上に実装する際の問題点

本章では，共有メモリ型マルチプロセッサ Unix マシン上で mutilisp などの真の並列処理言語処理系を実現する際に生じる問題点を提起する．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．

3.1 計算の割り当てと並列実行の指定

汎用マルチプロセッサ Unix マシンの OS には，プ

ロセッサが複数個存在することをユーザには意識させない設計のものが少なくない．このような OS では，カーネルがプロセッサへ計算の割り当てをおこなう．ユーザ側からみると，並列におこなわれる計算が同一プロセッサで実行されるとかえってオーバヘッドが大きくなる，などの事情があるので，プロセッサへの計算割り当ての機能が提供されている方が便利なことが多い．Universe 400 の場合は，通常はユーザがプロセッサへの計算割り当てについて注意を払う必要はないが，それが必要な場合のために，2.2 節で説明したような機能がユーザに開放されている．

一方，スレッドのライブラリを提供するシステムもある．ただし，これがマルチプロセッサ上での処理系作成に利用できるためには，複数のスレッドが同時に"実行中"となりうることがライブラリの仕様に含まれている必要がある．

3.2 相互排除

並列に進む複数の計算による共有データへのアクセスの相互排除の実現手法は，システム全体の性能に影響を与える．その実現法としては，

1. セマフォ関連のシステムコール (semop) の利用；
2. test and set に相当する機械語命令の利用；
3. 相互排除ライブラリ（提供されていれば）の利用；

が考えられるが，各々には以下のような得失がある．

1 は，Unix のシステムコールという重い作業によるので，効率的な実現法とはいい難い．ただし，ロック獲得に失敗した計算を Unix 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プロセッサの浪費は避けることができる．

2 の機械語命令の利用は，高速ではある反面，ロック獲得の失敗の際の対処を，すべてユーザレベルでおこなわなければならない．また，test and set を利用する部分はアセンブラで記述しなければならず，処理系の移植性がそこなわれるが，通常は数行程度ですむので，さほど問題にはならない．

方法 3 を採用するか否かは，ライブラリの実現法に依存する．たとえば，相互排除がシステムコールによって実現されていれば，事情は 1 と同じである．この方法は，ライブラリの実現法が改善されれば，（再コンパイルするなどして）それをすぐに取り込むことができるのが利点である．ただし，相互排除のためのライブラリ関数の標準がまだ存在しない点が，移植性の障害となりうる．

また，相互排除自体の回数を少なくすることも重要である．そのための一般的な方法は，相互排除の対象となるデータを，"プロセッサごとに局所的なデータ"と"大域的な（プロセッサ間で共通の）データ"の二段階にわけて管理し，相互排除は後者へのアクセス時のみに限定することであろう（4.2 節参照）．

3.3 プロセッサの放棄とスケジューリング

計算の進行過程で，あるプロセッサ P_i 上である条件 C が成立しない（たとえば，ロックを獲得できない，実行待ちの仕事がない，など）ために，P_i では C の成立を待つ以外なくなること（これを以下"待機状態"と呼ぶ）がおこりうる．このような場合には，以下の処置が必要となる．

- P_i での計算は，C が成立するまで待機する；
- C が成立するようになった場合には，P_i で待機している計算はそのことをなるべく早く知り，待機状態から脱却する．

前節で述べたセマフォ関連のシステムコールを用いないならば，前者はプロセッサを放棄することで実現し，後者は，他プロセッサにおける計算（たとえば，ロックを獲得していた計算）から，C が成立するようになったこと（獲得していたロックを解除したこと）を通知してもらうのが理想であろう．Unix においては，これらはシステムコール☆により実現する．

ところが実際は，システムコールによって不可分に実行される機能が，われわれの目的には細かすぎる，という問題点がある．そのために"隙"が生じ，各プロセッサの実行のタイミングによっては，正しく作動しないことがおこりうる．

たとえば P_i での計算において，"プロセッサを放棄することを決定し，そのための準備をおこなう"ことと，"放棄するための システムコールを呼ぶ"ことは不可分に実行できないので，両者の間に他の計算から"通知"がおこなわれれば，放棄する計算にはその通知が正しいタイミングでは伝わらない（これを"通知もれ"と呼ぶ）ことがある．このような場合には，P_i での計算は次の通知が到着するまで待たされ，通知の受信が"一回遅れ"になってしまう．

また，第二の問題点として，Unix のシステムコールは Unix プロセスの単位で作用する点がある．したがって，並列計算を複数のスレッドでおこなう場合に

☆ "放棄"は，pause（シグナルの到着待ち），read（ファイルディスクリプタからの入力）など，"通知"は，kill（シグナルの送信），write（ファイルディスクリプタへの出力）などを用いる．

は，システムコールで提供される機能がスレッド単位に作用するような対策が必要になる．

上で説明したとおり理想的な解決法は問題点が多いので，待機状態に陥った場合の対策は，

- ビジーウェイトで待機する；
- システムコールを用いるが，“通知もれ”を認めることとする．すなわち，運悪く隙をつかれたために通知の伝わるのが“一回遅れ”となってもやむを得ないものとする；
- 待機状態の処置を，処理系の下位部でおこなわずに，上位部でおこなうようにする；

のいずれかとするのが，現実的であろう．ここで第一・第二の方法は，処理系下位部での解決策である．第一・第二両法，あるいは第一・第三両法の折衷案（最初の適当な時間だけはビジーウェイトし，それでも C が成立しなければ，第二・第三の方法をとる）も考えられる．

第一の方法をマルチユーザの環境下で用いる場合は，ビジーウェイトによるプロセッサの浪費を少なくするように，C が不成立である時間が短いように処理系を工夫すべきである．たとえば，ロックの獲得の例では，きわどい領域の範囲をなるべく狭くして，ロックを獲得している時間が短くなるようにする．

第二の方法は，通知もれが稀にしかおこらないことを前提とすることはもちろんであるが，通知もれのために永久に待機してしまうことがないように留意する必要がある．性能は，実現する処理系での“通知”発生の時間間隔に大きく依存する．また，システムコールは重い作業なので，その発行回数をなるべく減らすような工夫も必要である．

第三の方法は処理系上位部に処置を委ねるもので，本論文で紹介する mutilisp 処理系で利用している手段である．この方法では，待機状態に陥ったことを，処理系下位部から，処理系上位部で作動しているプログラム（これを“スケジューラ”と呼ぶ）に知らせるための手段を用意しておく．スケジューラはこの知らせを受けて，適切な処置をおこなう．

実行待ちの仕事（Lisp プロセスなど）がなくなった場合に第三の方法を適用すれば，プロセッサ放棄と負荷分散とを同時に扱うことができる．たとえば，プロセッサごとに専用のスケジューラ（局所スケジューラ）を用意しておく．局所スケジューラは，待機状態に陥った知らせを受けると，他の局所スケジューラと通信しあい，実行待ちの仕事を分けてもらう．分ける仕事がない旨の返事を受け取れば，プロセッサを放棄する．これを解除するのは他スケジューラの責任であり，それは他プロセッサで分け与える仕事ができた時とする．4.4 節で，具体例を示し，この方法の得失の検討もおこなう．

4. mutilisp の実現方式

本章では，前章で提起した問題点の解決例として，Universe 400 上で実現した mutilisp 処理系の構成法を示す．

4.1 計算の割り当てと並列実行の指定

2.2 節で述べたように，Unos は，プロセッサへの計算の割り付けを Unix プロセスの単位でユーザに提供する．したがって，必要な個数だけ子 Unix プロセスを生成し，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，以下のような手順をとる．

1. 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するなどの，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，メインループに入る．

4 でのシステムコールは Unos に特有で一般的なものではないが，他の Unix マシンに移植する際には，これを“何もしない”関数（あるいはマクロ）として定義しておけばよいので，移植性の妨げにはなら

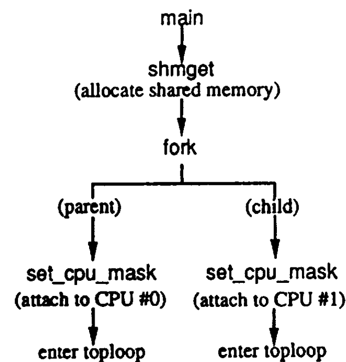


図 1 インタプリタを構成する Unix プロセス
Fig. 1 Unix processes in mutilisp interpreter.

ない.

本処理系を実現したハードウェアではプロセッサが2個なので，子 Unix プロセスはひとつだけ生成する．各プロセッサに分配された Unix プロセスのメインループでは，2.1 節で述べたように，実行可能キューから Lisp プロセスを取り出してはそれを実行することを繰り返す（図1）.

4.2 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るので，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．また，プロセッサの負荷がかたよらないようにする必要もある．そこで，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が可能であるようにした．そのためには，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．

相互排除の実現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の面で問題があるので，本処理系ではこの部分のみ test and set 命令を用いたアセンブリ言語で記述した．実際，この部分は4行程度の簡単なプログラムである．

また，3.2 節の最後で述べたように，相互排除の回数を少なくするため，その対象となるデータを二段階にわけて管理するのを基本方針とする．たとえば，ヒープ領域からのオブジェクトの割り当てについては，各プロセッサが，ヒープからある一定の大きさの領域（局所的なヒープのかたまり）をまとめて確保しておいて，通常はその領域からオブジェクトを割り当て，相互排除は局所的なヒープの確保時のみおこなう．同様に，実行可能キューはプロセッサごとに局所的に保持し，相互排除をなくすと同時に，3.3 節で述べた“プロセッサ放棄”への対処もおこなっている．この点について，次節以降で詳しく議論する．

4.3 スケジューラ

本 mutilisp 処理系では，プロセス管理を，処理系下位部における管理と処理系上位部における管理のふたつに分け，システムの自由度を高めている．

下位部は 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．前節で述べたように，実行可能キューはプロセッサごとに保持するので相互排除の必要はない．

上位部では，スケジューラと呼ばれる特別な Lisp

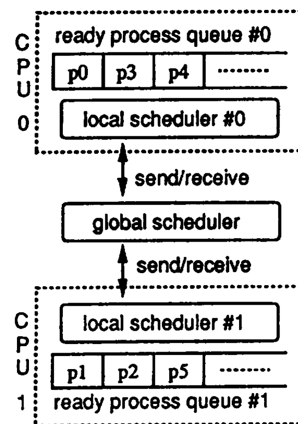


図 2 大域スケジューラと局所スケジューラ
Fig. 2 Global scheduler and local schedulers.

プロセス群が，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 mutilisp 自身で記述され，ユーザが機能を自由に拡張・変更することができる．上位部でのプロセス管理は，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）.

各局所スケジューラは3.3節で説明したもので，自分の担当するプロセッサ上でのみ実行される．この役割は，担当プロセッサに付随する実行可能キューの管理（これにアクセスするのは局所スケジューラのみ）を主とした，局所的なLispプロセス管理である．負荷分散の際の局所スケジューラ間のメッセージ交換は，大域スケジューラを経由しておこなう．一方，大域スケジューラは，どのプロセッサでも実行されうる．

4.4 負荷分散とプロセッサの放棄

本節では，大域・局所両スケジューラの協力による負荷分散，プロセッサ放棄の方法の大枠を示す．（細かい例外処理などについては脚注で述べる．）以下の説明中，i 番目のプロセッサを P_i，その実行可能キューと局所スケジューラをそれぞれ Q_i，S_i とする．また，大域スケジューラは G とする．

あるプロセッサ P_i で Q_i が空になると，mutilisp インタプリタは，S_i に対し内部的なメッセージ[6]（シンボル noready）を届ける．S_i はこれを受け取ると，G に対して負荷分散を依頼（シンボル getproc を送信）し，返事の受信態勢に入る☆.

☆ まれにここでビジーウェイトが必要なこともある．

G は S_i から getproc を受信すると，適当な S_j $(j \neq i)$ を選び，実行可能 Lisp プロセスを分けるように依頼する（シンボル getproc を送信）．S_j はこれに対し，Q_j から適当な Lisp プロセスを外して（複数でも可）G に送るが，分け与える Lisp プロセスがなければ，その旨（シンボル pause）を G に送信する．後者の場合には，後で分ける余裕ができた時点で，S_j は（前者の場合と同様にして）G に Lisp プロセスを送信する．

G は，S_j からの返事をそのまま S_i に送る．ただし返事が pause ならば，プロセッサ放棄の指令の意味である．この場合，後で S_j から Lisp プロセスが送られてきたら，これを S_i に送信し☆，同時に組み込み関数を呼んで P_i での計算を放棄状態から脱却させる．

局所スケジューラ S_i は，G への負荷分散の依頼の返事として Lisp プロセスを受け取れば，これを Q_i につなげて負荷分散が完了する．pause を返事として受け取れば，プロセッサ放棄の組み込み関数☆☆を呼ぶ．放棄状態が後に G によって解除されると，G から Lisp プロセスが送信されているはずなので，これを受け取って Q_i につなげる．

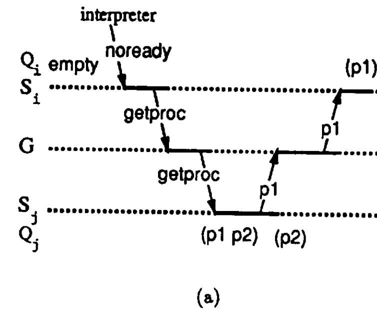
図3では，プロセッサ i の実行可能キュー Q_i が空になった時，プロセッサ j が自分の実行可能キュー Q_j（p1, p2 の2個の Lisp プロセス）から p1 ひとつを分け与えた例を示している．図中，時間軸は横方向で，実線部は当該スケジューラが実際に動いていることをあらわし，S_i の線の上，S_j の線の下にその時点での各プロセッサの実行可能キューをあらわす．図3(a)はプロセッサ放棄がおこらなかった場合，図3(b)は S_i がプロセッサを放棄した場合である．

この方法の第一の利点は，3.3 節で述べた待機状態の処置を，Lisp プログラムのレベル（大域・局部スケジューラのプログラム）に委ねている点である．Lisp プログラムのレベルであっても3.3節で述べた“隙”の問題は存在するが，本方法によって，“隙”を（局所，大域）両スケジューラ間の関係のみに限定することができるので，両スケジューラの合意のもとに，局所スケジューラがプロセッサを放棄する，限られた場合にのみビジーウェイトを用いる．などの，無駄を抑えた対処法を講じることができる．

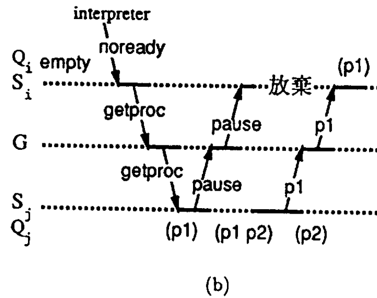
また第二の利点は，負荷分散の戦略をいろいろと変えることができる点にある．特定の問題に適したスケジューリングをおこなうことも可能である．

☆ 実際にはプロセッサを放棄している局所スケジューラならばそのどれを選んでもかまわない．

☆☆ プロセッサ放棄と放棄からの解除はパイプに対する read と write で実現している．



(a)



(b)

図 3 負荷分散
Fig. 3 Load balancing.

第三の利点として，プロセッサの実行可能キューが空になると局所スケジューラから大域スケジューラにメッセージが送信されるので，デッドロックの検出が容易である点があげられる．

5. 実験結果

本章では，前章で述べた mutilisp 処理系の性能を計測し，実装法の簡単な評価を試みる．なお，ここでのプログラムはすべて，コンパイルせずに解釈実行した．また，計測中には他にアクティブな Unix プロセスは走らせず，CPU 能力のほぼすべてが計測するプログラムの実行に使われるようにした．局所スケジューラは負荷分散の際には自分の実行可能キューの先頭にある Lisp プロセスをひとつだけ分け与える，というものにした．

比較の対象とした他の処理系は，

- プロセッサひとつのみを用いる疑似並列版の mutilisp 処理系
- すべてのプロセッサでひとつの実行可能キューを共有する mutilisp 処理系

であり，前章で述べた処理系と同じく Utilisp/C を基

にして作成した．これらについては，スケジューラを起動しない場合と，単純なラウンドロビン方式のスケジューリングをおこなうデフォルトのスケジューラを起動した場合とについて測定をおこなった．

以後，疑似並列版は pp (pseudo parallel)，実行可能キュー共有版は sq (single queue) と略記し，スケジューラを起動しない場合には‘0’を，起動した場合には‘1’を後につけて（pp0, sq1 などのように）区別する．また，第4章でのべた処理系は mq (multiple queue) と略記する．sq0, sq1 では実行可能キューの相互排除のコストがかかるが，一方，mq ではスケジューラによる負荷分散のコストがかかる．

実験で用いた題材は，8クィーン問題，および，素数生成である．並列に解くプログラムは，マスタ・ワーカ (master-worker) 形式[12]で記述した．これでは，あらかじめマスタ Lisp プロセスといくつかの（等価な）ワーカ Lisp プロセスを生成しておいて，並列計算はワーカに分配しておこなう．ワーカは，与えられた計算を終えると結果を返し次の仕事を受け取る，ということを繰り返す．問題に応じて，マスタは，ワーカに仕事を分配したり，ワーカから結果を回収したりする（図4）．ワーカへの計算の分配・計算結果の回収などはメッセージ送受信でおこなう．ここでの測定では，ワーカ数は2つにした．

8クィーン問題について，queens 8 は逐次的に解くプログラム，pqueens 8 は並列に解くプログラムである．queens 8 と pqueens 8 で異なる点は，queens 8 では組み込みの map 関数を用いるところの一部を，pqueens 8 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．マスタとワーカの生成に要する時間は pqueens 8 の実行時間には含めない．

素数生成では，15,000 以下の素数を，文献12)に示されているマスタ・ワーカ形式のアルゴリズムを用いて求める．ワーカのひと仕事は，与えられた数 n から $n+grain$ までの範囲の素数を求めることであり，結果はマスタに通知する．ここで，$grain$ はワーカでのひと仕事の大きさを定める定数であり，これを20にした場合（primes 20）と50にした場合（primes 50）について，並列版の処理系において計測した．

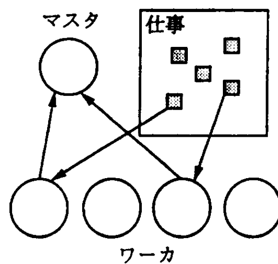


図4 マスタ・ワーカ形式
Fig. 4 Master-worker model.

表1 各問題の実行時間（単位：1/50 秒）
Table 1 Execution times (In 1/50 second).

	queens 8 t	pqueens 8 T (t)	primes 20 T (t)	primes 50 T (t)
pp0	1789	2119(2119)	4517(4517)	4277(4277)
pp1	1920	2217(2217)	4849(4849)	4569(4569)
sq0	—	1275(1275)	2780(2780)	2586(2586)
sq1	—	1414(1414)	3207(3207)	2976(2976)
mq	—	1224(1174)	2610(2567)	2371(2355)

T：経過時間，t：CPU 時間

結果を表1に示す．並列版で使用したプロセッサ数は2であり，ごみ集めに要する時間は含めない．また，並列版では，答を求めるまでに経過した時間 (T) と，その中で実際に消費した CPU 時間 (t) の双方を示す．（mq 以外では，$T=t$ である．）以後，特定の処理系における T, t の値を参照する時には，T_{sq0}, t_{mq} のように処理系の略称を添字につけてあらわす．

経過時間でみると，mq での pqueens 8 は，pp1 の queens 8 の約 64%，pp0 の queens 8 に対しても 70% 未満の性能を示している．また，全体としてみると，T_{mq}/T_{sq0} は 96% 以下，T_{mq}/T_{sq1} は 80〜87% 程度である．mq で実行可能キューをプロセッサごとに保持して負荷分散をおこなう効果が出ており，本論文で述べた手法の有効性が示されている．同時に，mq における負荷分散のコストが大きくないことも，この結果は示している．

実際，mq でのプロセッサ放棄・負荷分散のコストを見積もると，以下のようになる．pp0 は，スケジューラを起動せず実行可能キューへのアクセスの相互排除もおこなわないので，t_{pp0} は本質的に必要とされる CPU 時間の近似値とみなすことができる．したがって，$v=2t_{mq}-t_{pp0}$ は，負荷分散を含めた大域・局所両スケジューラの仕事のコストの合計となる．負荷分散により移動した Lisp プロセス数を N_p とすれば，$m=v/N_p$ は，Lisp プロセスひとつを移動させるコストの尺度となる．また，消費された CPU 時間全体に占める負荷分散の割合 r は，$r=v/2t_{mq}$ となる．

これらの値を，表2に示す．問題・スケジューリン

表 2 mq における負荷分散
Table 2 Load balancing in mq.

	pqueens 8	primes 20	primes 50
N_w	64	750	300
N_p	26	104.2	29
N_y	13	52.1	14.5
m	8.8	5.9	14.9
r	9.8%	12%	9.2%

N_w: ワーカへの仕事の総数
N_p: 負荷分散により移動した Lisp プロセス数
N_y: プロセッサひとつあたりの放棄回数
m: 移動 Lisp プロセスあたりの負荷分散のコスト (単位 1/50 秒)
r: CPU 時間における負荷分散の割合

グ戦略により，これらの値は変動するが，約 10% の CPU 時間が負荷分散に費やされている．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 mq における負荷分散の手間はさらに軽減することが期待される．

mq におけるプロセッサ放棄の効果に注目すると，全体の経過時間からみると，放棄による CPU の節約は数%程度となっている．しかし，状況・問題によっては大きな効果が期待できると考えられる．

6. 考　　察

本章では，ここまでの議論をふまえて，並列言語の実装について，OS から見た側面，および，実装する言語から見た側面の両面から考察し，同時に他の研究との関連についても述べる．

6.1 OS からの側面

並列言語処理を実現する際に有力な手段は，ひとつの Unix プロセスの中のスレッドを利用することである．多田ら[3]は，移植性の高いスレッドライブラリを提案しており，単一プロセッサ Unix マシン上での高速なコンテクスト切替えを実現した．ただし，各スレッドのきめ細かい制御，(たとえば，あるスレッドがターミナルからの入力待ちになったときに別のスレッドにコンテクストを切替える，など) は提供していない．Sun マイクロシステムズ社は，Sun OS 4.0 以降で Lightweight Process ライブラリ[1]を提供している．このライブラリでは，スレッドの制御に関する豊富な機能を提供する反面，移植性は低い．両者に共通する問題点は，仕様上，複数のスレッドが同時に実行中とはならないため，マルチプロセッサに対応するためには，機能拡張が必要な点である．

Mach (C Threads ライブラリ[2]) では，ひとつのタスク (Unix プロセスに相当) 中に，複数のプロセッサ上で並行に動く複数のスレッドの存在を許す．C Threads で記述した並列記号処理言語システムの例には，オムロン社の Luna-88K 上で作動する並列論理型言語 Fleng[13] がある．ただし，これでは，計算のプロセッサへの割り当てまでユーザが指定することはできず，OS にまかせるほかない．また，相互排除のためのライブラリ関数には，ビジーウェイトを用いているものもある．

本実装ではスレッドを用いず，メモリを共有した複数の Unix プロセスをプロセッサごとに専門に張りつける，というアプローチをとった．ただし，Unix プロセス間で，オープンしたファイルの状態までを共有することはできない．したがって，Lisp プロセスがファイル入出力を行う場合は，そのファイルをオープンした Unix プロセスにおいてのみ実行しなければならない，という実装上の制約が課せられる．いずれにしても，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．

6.2 言語からの側面

本論文で提案した手法では，OS の機能の制約のために処理系の下位部で完全に解決するのが困難な問題を，処理系上位部のプログラムに処置を委ねる．ここでは，実行時の動的スケジューリングが重要な役割を果たす．必要なのは，待機状態に陥ったことなどを処理系上位部 (局所スケジューラ) に知らせる機構，上位部の (スケジューリング) プログラム(群)が陽に計算の状態を変化させる (プロセッサを放棄する，実行可能キューを操作する，など) ことができる機構である．その意味で，記号処理言語に限らず，多くの言語処理系において本手法を応用できると考える．

mutilisp は，2.1 節で述べたように，Lisp プロセス間での Lisp オブジェクトの共有を排除している．したがって，シンボルへの値・関数などの設定，rplaca などによる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 が，効率的な処理系の実現に言語仕様の面から支援していることを意味する．また，変数を共有して並列に計算を進めていく Multilisp[14]，Qlisp[15] などの並列 Lisp システムと異なり，mutilisp における Lisp プロセス間通信はメッ

セージ伝達をベースにしている．したがって，本実装ではインタプリタ内部（下位部）と上位部にある Lisp プロセスとのインタフェースは，下位部から発生する特殊なメッセージとして，自然な形でシステムに取り入れることが可能となった．

mutilisp の上位部は，ユーザによるプログラミングが可能であるので，本システムではユーザの選択の自由が大きい．今回の実装においても，局所・大域スケジューラのアルゴリズム，実行可能 Lisp プロセスが消滅したときの処置，などをいろいろと変えて実験することができた．上位部でのプロセス管理は，局所スケジューラと大域スケジューラに分けたが，これは，単一プロセッサ上での疑似並列版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．

7. おわりに

本論文では，Unix 系共有メモリ型マルチプロセッサマシン上に，真の並列言語処理システムを実装する際の問題点を議論し，具体例として，Universe 400 上で試作した mutilisp 処理系について述べた．

ここで提案した方法では，OS の機能の制約等から発生する問題を，なるべく処理系の内部のみで処理しようとせず，処理系上位部の，具体的にはスケジューラのプログラムで処理をおこなうようにつとめるのを基本方針とした．そのために，疑似並列版（pp）あるいは実行可能キュー共有版（sq）と比較して，本方法の版（mq）が必要としたインタプリタ本体に対する拡張は，複数のスケジューラ（大域・局所）の登録，Lisp プロセスに対する "CPU マスク" の設定（実行する CPU の指定）など，比較的小さな手間ですんだ．この点からも，本論文で示した手法は他の並列言語システムにおいても有効であると考える．

われわれは，スケジューラのプログラムをいろいろとさしかえることにより，様々なスケジューリングや負荷分散のアルゴリズムを実験することができた．今後は，さらに実験をすすめていくと共に，この柔軟性を生かしたシステムの記述・実現法を検討していくことが課題である．さらに，疎結合型マルチプロセッサ上における実現法の検討もすすめていきたい．

謝辞 Universe 400 システムを利用する環境を提供してくださった東京大学の和田英一名誉教授，川崎製鉄(株)の関係者諸氏に感謝する．

参考文献

1) Sun OS System Services Overview, Sun Microsystems, Inc. (1988).
2) Cooper, E. C. and Draves, R. P.: C Threads, CMU-CS-88-154, Carnegie Mellon University (1988).
3) 多田好克，寺田 実：移植性・拡張性に優れた C のコルーチンライブラリー実現法，電子情報通信学会論文誌 D-I, Vol. J 73-D-I, No. 12, pp. 961-970 (1990).
4) 岩崎英哉：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).
5) Iwasaki, H.: mutilisp: A Lisp Dialect for Parallel Processing, *Proc. US/Japan Workshop on Parallel Lisp* (*LNCS 441*), pp. 316-321, Springer-Verlag (1990).
6) 岩崎英哉，寺田 実，湯浅 敬：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).
7) Accetta, M. et al.: Mach: A New Kernel Foundation for UNIX Development, *Proc. Usenix Technical Conference*, pp. 93-112 (1986).
8) 近山 隆：Utilisp システムの開発，情報処理学会論文誌，Vol. 24, No. 5, pp. 599-604 (1983).
9) Wada, E.: History of UtiLisp Hacking, *J. Inf. Process*, Vol. 13, No. 3, pp. 276-283 (1990).
10) 田中哲朗：SPARC の特徴を生かした UtiLisp/C の実現法，情報処理学会論文誌，Vol. 32, No. 5, pp. 684-690 (1991).
11) VCP-4000 MP User's Manual, Charles River Data Systems, Inc. (1987).
12) 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).
13) Nilsson, M. and Tanaka, H.: FLENG Prolog—The Language which Turns Supercomputers into Parallel Prolog Machines, *Proc. 5th Logic Programming Conference* (*LNCS 264*), pp. 170-179, Springer-Verlag (1986).
14) Halstead, R.: Multilisp: A Language for Concurrent Symbolic Computation, *ACM Trans. Prog. Lang. Syst.*, Vol. 7, No. 4, pp. 501-538 (1985).
15) Gabriel, R. and McCarthy, J.: Qlisp, *Parellel Computation and Computers for Artificial Intelligence*, pp. 63-89, Kluwer Academic Publishers (1988).

(平成 4 年 1 月 27 日受付)
(平成 4 年 9 月 10 日採録)

岩崎　英哉（正会員）

1960年生．1983年東京大学工学部計数工学科卒業．1988年同大学院工学系研究科情報工学専攻博士課程修了．同年東京大学工学部計数工学科助手，現在に至る．工学博士．記号処理言語，関数型言語，並列・分散処理システムなどの研究に従事．日本ソフトウェア科学会，ACM 各会員．