マルチプロセッサシステム PARK 上での並列 Prolog 処理系の実現†

松田秀雄†† 小畑正貴††† 増尾剛††††
金田悠紀夫†† 前川禎男††

著者らの提案による並列論理型言語 PARK-Prolog とその処理系の実現について述べている．実現は著者らの作成したマルチプロセッサシステム PARK (要素プロセッサは 68000) 上で行った．PARK-Prolog では逐次型 Prolog のセマンティックスにプロセスの生成，プロセス間の同期・通信といった並列実行機能を付け加えて並列実行を記述する．これにより逐次型 Prolog 処理系の実現で用いられる最適化技法をそのまま利用し，さらに並列実行により実行速度を向上させることをねらっている．変数束縛環境はプロセスごとに独立で共有はされない．プロセス間の同期・通信はチャネルを介しての送信・受信により行われる．送信には受信と同期を取る同期型と同期を取らずに実行を継続する非同期型を設けた．PARK-Prolog 処理系の実現ではコンパイル時に出力される中間言語命令について述べている．68000 のような汎用マイクロプロセッサで高速に実行するためデリファレンス，トレイルの処理を省く nocheck 宣言を提案している．実行速度は append と reverse の逐次実行で約 18KLIPS であった．8クィーンをプロセッサ3台で実行した時では1台の時と比べ 2.6 倍の速度向上であり，Quintus Prolog, DEC-10 Prolog を上回る実行速度が得られた．

1. はじめに

論理型プログラミング言語についての研究がここ数年の間に急速に発展してきており，数多くの応用プログラムが論理型プログラミング言語で記述されるようになってきた．それと共に論理型プログラムの実行のより一層の高速化について研究する必要性が増してきている．このためには，実行の並列化が有効であると考えられており活発に研究が進められている．

論理型プログラムを並列実行する手法にはこれまでに種々のものが提案されている．それらは，大きく分けて論理式に内在する並列性を取り出す方式と，言語のセマンティックスに積極的に並列性を取り込んでいく方式に分かれる．

前者は AND 並列/OR 並列といった方式であり，純粋 Prolog (Prolog から副作用のある組込み述語を除いたもの) でプログラムを記述するのが一般的である．著者らは以前にこの方式で K-Prolog という並列 Prolog 処理系の実現を行っている[1]．この方式ではデータベース検索など全解探索の必要な問題を容易に記述できる反面，並列実行の制御手続きの記述が難しい．

後者の方式では，並列度の調節，プロセス間の同期の機能をもった並列論理型言語により並列実行を記述する．本論文では，PARK-Prolog と呼ばれる並列論理型言語の提案を行っている．PARK-Prolog では逐次型 Prolog のセマンティックスに，プロセスの生成，プロセス間の同期・通信といった並列実行機能を付け加えることにより，論理型プログラムの並列実行を記述する．この方式では逐次型 Prolog の実現で用いられる最適化技法 (インデキシング，TRO など) をそのまま利用でき，さらに並列実行でのプロセッサ台数効果により実行速度の向上をねらえる．

並列論理型言語としては，他に Concurrent Prolog[2]，PARLOG[3]，GHC[4] といった言語が提案されている．これらの言語は読出し専用表記，モード宣言，ガードなど並列実行を制御する機能を持っているため並列オペレーティングシステムの記述といったシステム記述に使うことができる．しかし，これらの制御機能が逆に別解の探索を制限するため，全解探索の必要な問題を直接記述するのは容易ではなく，プログラム変換などの手法[5]を用いる必要がある．これに対して PARK-Prolog では，逐次型 Prolog のセマンティックスを基礎としているため，全解探索の必要な問題もそのまま記述することができる (プログラム変換などの操作を行う必要がない)．その他プロセス間通信の手段の違いなどからくるいくつかの相違があるがこれらについては PARK-Prolog の並列処理方式のところ

† Implementing Parallel Prolog System on Multiprocessor System PARK by HIDEO MATSUDA (Department of Systems Engineering, Faculty of Engineering, Kobe University), MASAKI KOHATA (Department of Electronic Engineering, Faculty of Engineering, Okayama University of Science), TSUYOSHI MASUO (NTT Software Laboratories, Nippon Telegraph and Telephone Corporation), YUKIO KANEDA and SADAO MAEKAWA (Department of Systems Engineering, Faculty of Engineering, Kobe University).

†† 神戸大学工学部システム工学科
††† 岡山理科大学工学部電子工学科
†††† 日本電信電話(株)NTT ソフトウェア研究所

で述べる.

PARK-Prolog の実現は著者らの試作したマルチプロセッサシステム PARK (Parallel Processing System of Kobe University) 上で行った. PARK では, 要素プロセッサとしてモトローラ社の 16 ビットマイクロプロセッサ MC 68000 を使用しており, 要素プロセッサ複数個を共通バスで結合することにより構成している. プロセッサ間のデータ通信は, すべてのプロセッサから参照可能な共有メモリを介して行う.

以下ではまず PARK の構成について述べた後, PARK-Prologの並列処理方式とその処理系のPARK上での実現について述べ, 最後にいくつかの評価プログラムを実行した結果により PARK-Prolog 処理系の性能評価を行う.

2. PARK の構成

2.1 全体構成

PARK については文献6)などで報告しているので, 本論文ではその概略を述べるだけにとどめる.

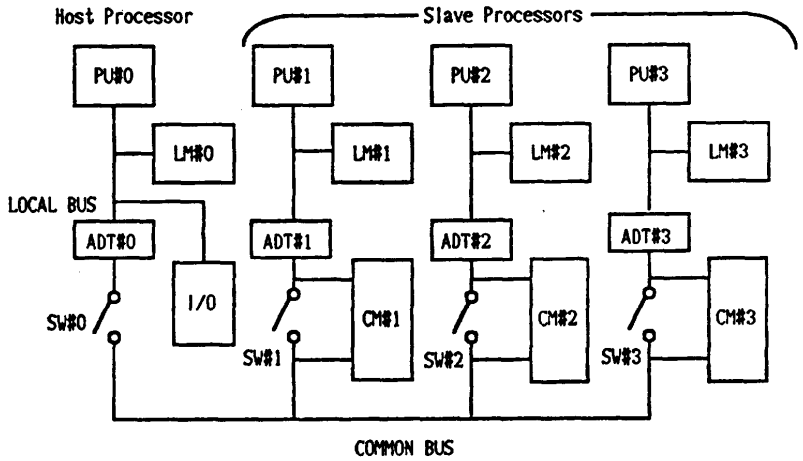
図 1 に PARK の構成を示す. PARK はバス結合型のマルチプロセッサシステムであり, ホスト／スレーブの2種類のプロセッサから成っている. ホストはユーザとの対話処理／ファイル入出力などの入出力処理を主に行い, スレーブはプログラムの並列実行を行う. 試作段階の現在のところホスト1台, スレーブ3台の構成であるが, 設計上はホスト, スレーブあわせて 16 台までを単に共通バス (common bus) に接続するだけで増やせるようになっている.

各プロセッサは, プロセッシングユニット (PU), ローカルメモリ (LM), アドレス変換 (ADT), バススイッチ (SW) から成っている. また, スレーブには共有メモリ (CM) が接続されている. PU には, 68000 (クロック 8 MHz) を使用している. なお, バス幅はローカルバス (LOCAL BUS) が 53 本, 共通バス (COMMON BUS) が 59 本となっている. その内容はアドレス線 23 本, データ線 16 本であり残りがメモリ制御, 割込み制御, バス調停 (共通バスのみ) といった制御信号線である.

PARK は先に著者らの作成した, PU にインテル社製マイクロプロセッサ 8086 を使ったマルチプロセッサシステム[1]に似た構成となっている. PARK の設計は, このシステム上での K-Prolog の作成経験をもとに行った. 8086 のシステムと比べて, 共有メモリの構成を変更し容量を増やすことにより, より大きな論理型プログラムを実行できるようにした. また, アドレス変換機構を追加しており, これによりマルチプロセス処理を行うときのメモリ管理を高速化することができる.

2.2 共有メモリ

PARK では, バス結合型システムの欠点であるバス競合を抑えるため, 共有メモリをバンク分けしプロセッサごとに分散配置している. プロセッサは自分に配置されているメモリバンクについてはバス競合を起こさずに参照できる. それぞれのバンクは別々のアド



PU: processing unit (MC68000, 8MHz), LM: local memory (host 256KB, slave 128KB), CM: common memory (512KB) ADT: address translation unit, SW: bus switch.

図 1 PARK のハードウェア構成
Fig. 1 Hardware structure of PARK.

レス空間に割り当てられている．さらに，共有メモリは放送機能を持っている．アドレス空間上に，共有メモリ1バンク分の大きさを持つ放送領域があり，ここに書込みを行うことによりすべてのバンクに同一のデータが一斉書込みされる．プロセッサ間で共有されるデータの更新に放送を利用することにより共通バスの使用回数を減らすことができる．

ローカルメモリの参照，共有メモリの参照，放送はこの順番で参照時間が増大するようになる．これらの参照時間の具体的な値は，PU と共有メモリとが異なるクロックで動作しているためタイミングによって違うが，最良値と最悪値で単純に平均すれば順に 500 ナノ秒，約 1.6 マイクロ秒，約 2.7 マイクロ秒となる（バス競合の時間は考えていない）．これらの参照の使い分けはすべてソフトウェアで行われるので，実行速度を向上させるためには，データの局所性を高めて共有メモリに対するローカルメモリの使用頻度を増やすようにしなければならない．

2.3 アドレス変換

PARK では，共有メモリへのアドレス入力に対して1レベルのアドレス変換を行っている．変換はページ単位で行われ（1 ページ 8KB），68000 の持つ 16 MB のアドレス空間中の任意アドレスに共有メモリのページをマッピングすることができる．マッピングは読出しと書込みとで別々に行われ，書込み保護などの機能をページ単位で実現できる．アドレス変換は高速スタティック RAM をテーブルとして使うことにより実現されており，1 CPU クロック（125 ns）以内に変換できる．

3. 並列処理方式

3.1 プロセス

本論文で述べる PARK-Prolog の並列処理方式は，プロセスの概念を用いて記述される．ここでプロセスとは，ゴールの逐次的な実行を指す．

プロセスの実行するゴールは，プロセスごとに異なるがプログラムはすべてのプロセスについて共通とする．変数束縛の環境はプロセスごとで別々に持つ（多環境方式）．プロセスの生成やプロセス間通信の際には，実行すべきゴールあるいは通信されるデータの環境がコピーされて受け渡される（未定義変数の値セルもコピー先の領域に移される）．

プロセス間通信は，チャネルを介して行われる．occam[7] のチャネル，Delta-Prolog[8] のイベントといった方式と違い，チャネルの生成は実行時に動的に行われる．また，一度に複数のプロセスが1つのチャネルにデータを送信することを許す（1チャネルでの非決定的マージ）が，チャネルからの受信は1つのプロセスに限定する多対1の通信となっている（一度に複数のプロセスが1つのチャネルから受信を行うとエラーとなる．この判定は最初の受信時にチャネルにそのプロセスの ID を書き込み，以後それとあっているかどうか調べることにより行える）．文献 6) では，通信を1対1で行う単一 send-receive と多対多で行うマルチ send-receive の2種類用意していた．今回，通信をこのような形にしたのは，多対多の通信にはチャネル上の送信データを受信が行われた後も取り除けない（次の受信で使われるかもしれない）といった欠点があるためである．多対1の通信では，受信のたびに送信データをチャネル上から取り除くことができメモリ消費を抑えられる．

単一環境方式の GHC[4] と比べて多環境方式の PARK-Prolog はプロセス間通信が大きく異なっている．GHC では共有変数の更新・参照により通信が行える．PARK-Prolog の方式は，PARK のローカル／共有／放送の3つのメモリ参照方式を意識してプログラムを書くことができるように設定した（後述するように一時変数をローカルメモリに，ローカル／グローバル変数を共有メモリに，チャネルを放送領域に割り当てている）．共有変数を介した通信では容易な，差分リストを用いたストリーム通信，通信の相手先やその数を実行時に動的に変えるといった手法は記述が難しい．その代わり，ストリームのマージは容易であり（1つのチャネルに複数のプロセスから送信できる），送信データは受信が終了するとチャネルから消滅するためメモリ消費も小さい．また TRO (tail recursion optimization) や，グローバルスタックのちり集めを，他のプロセスの環境を参照せずに局所的に行うことができる．

3.2 記　　法

PARK-Prolog では，逐次実行部分の記述は逐次 AND ‘,’，逐次 OR ‘;’ の2つの2項演算子で記述する．並列実行の記述のため，次のような記法が用意されている．

（1） プロセスの生成

記法：ゴール1　//　ゴール2

　　fork (ゴール)

‘//’ はゴールとゴールとを結ぶ2項演算子として使

い，ゴール1，ゴール2のそれぞれを実行するプロセスを生成する．構文の上では並列 AND であるが，論理的な AND 関係ではなく，一方のゴールが失敗しても他方のゴールの実行に影響を及ぼさない．また，両方のゴールに共通の変数があってもプロセス生成後は別の変数となる．これらの変数で値の一貫性を保つ，ゴールの実行の結果（ゴールの解またはゴールの実行の成功／失敗といった情報）を親プロセスに返すといった処理は次のプロセス間通信機能を使ってプログラム中で明示的に記述する必要がある．並列 AND をこのように制限された形にした理由は，プロセス間にまたがる失敗や，多環境のもとで共有変数の値の一貫性を保つといった処理を実現するのが容易でないためである．fork は引数として与えられたゴールを実行するプロセスを生成する組込み述語である．なお，プロセスのプロセッサへの割当てはプログラムでは指示せず処理系側で行うことにしている．

（2） チャネルの生成／消滅

記法：mkchan([チャネルリスト])…生成
rmchan([チャネルリスト])…消滅

プロセス間通信のためのチャネルを生成／消滅させる組込み述語である．mkchan ではチャネルリスト中にある変数の個数だけチャネルが生成され，チャネルへのポインタが各々の変数に代入される．rmchan ではチャネルへのポインタが代入された変数のリストを受け取り，それらのチャネルを消滅させる．チャネル領域のちり集めは，今のところ行っていないので rmchan を実行しないかぎりチャネルの数は単調に増加することになる．

（3） チャネルへの送信

記法：チャネル！送信データ……同期型
チャネル！！送信データ……非同期型

チャネルにデータの送信を行う組込み述語である．指定したチャネルに対して既に受信が行われていなければ，実行を中断 (suspend) する同期型とそのまま送信を行って次の実行に移る非同期型の2つの送信方法がある．同期型の送信は，際限なくループを回るプロセス (perpetual process) から他のプロセスにプロセス切替えを行うのに使える．同期型では，送信プロセスの情報が送信データとともにチャネルに書込まれる．

（4） チャネルからの受信

記法：チャネル？受信パターン

チャネルからデータを受信する組込み述語である．送信が先に行われていなければ実行を中断 (suspend) する．チャネル上のデータを受信パターンとの統一化 (unification) により受信を行う．ただし，統一化での変数への値の代入方向は送信から受信への一方向のみである．統一化が失敗したときには，この述語の実行の失敗としてバックトラックする．文献6)で述べたリトライ条件を削除したので，送信プロセスの実行を失敗させることはできない．もし，どうしても送信プロセスの実行を失敗させたいときには，その通信の後で逆方向の通信を行うなどしなければならない．送信が同期型か非同期型かは統一化の後，送信データ中の送信プロセス情報があるかどうか調べることにより識別する．同期型で送信されている場合は，受信の後で送信プロセスの実行を再開 (resume) する．

4. 処理系の実現

4.1 中間言語

PARK-Prolog の処理系は，コンパイル方式により実行を行う．プログラムを最終的に 68000 の機械語にまでコンパイルするが，途中で中間コードを出力する2段階コンパイルとしている．中間コードは，以下に述べる中間言語の命令セットに基づいて生成される．中間言語命令はアセンブラのマクロ命令として定義されており，中間コードから機械語へは単にマクロの展開を行う．最適化の処理については，中間言語レベルで行っており，後で述べるようにモード宣言により冗長な型判定命令を除くようにしている．また，一時変数の使用や転送を行う中間言語命令のうち冗長な命令の除去など，簡単な最適化を行っている．これは，今のところ中間コードに対して人手で行っている．

中間言語では，逐次実行と並列実行のための命令がそれぞれ用意されている．逐次実行のための命令では，Warren の仮想 Prolog マシン (WAM : Warren Abstract Machine) の命令セット[9]を参考にして設計した．68000 のような汎用のマイクロプロセッサでも高速に実行させるため，文献10) で述べられているのと同様の手法を用いている．すなわち，WAM では命令の中で行っていたタグ判定，デリファレンス，トレイルといった処理を取り出して独立した命令とする，モードビットによって動作を変えていた get/unify 命令をリードとライトの各モードでそれぞれ独立した命令とする，モード宣言の中でゴール引数の入出力に加えて型も宣言する，型（未定義変数，リファレンス，nil，アトム，整数，リスト，構造体）ごとに専用の命

令を設けるなどである．しかし，命令の基本的動作はWAM と全く同じであり特に新しい命令を設けたわけではない．

文献 10) では，このほかに notrail 宣言を設けトレイルの処理を省くことにより，決定的な節の実行の高速化を行っている．PARK-Prolog ではさらに進んで，デリファレンスとトレイルの両方の処理を省く nocheck 宣言を設けている．これはゴール引数の値がリファレンスであった場合（またはモード宣言により出力変数であると指定されている場合），未定義変数を直接指していると仮定するものである（未定義変数は自分自身へのリファレンスで表す）．この時，デリファレンスは必要なくなる．また，バックトラック時に未定義変数に戻すアンドゥ操作を行わなくても未定義変数と仮定して強制的に代入を行うので，トレイルも行う必要がなくなる．

モード宣言，nocheck 宣言を行った決定的 apdend のプログラムとそのコンパイル例を**図 2** に示す．図2の(b)，(c)で命令のオペランドの数字はすべて一時変数の番号を指す．一時変数のうち，番号 0, 1, 2 の変数がゴール引数として使われている．図2で case_type 命令は型による条件分岐命令で，統一化の候補節の選択（インデキシング）に使われている，case_type 命令の引数は，初めが変数番号で，以下は順に型が未定義変数以外のリファレンス，未定義変数，nil，アトム，整数，リスト，構造体の時の分岐先を示す．モード宣言中の型記述の指定があるところだけ分岐先があり，それ以外は空欄になっている．空欄になっている型のチェックは行われない．変数の型が分岐先の指定されたすべての型のどれにもあてはまらなかった時はエラーとしている．このため，モード宣言では節のヘッド引数の型とは関係なく，ゴール引数のすべての可能な型を書かなければならない．モード宣言中に指定されている型と同じ型のヘッド引数を持つ節がないときは，case_type 命令の分岐先は fail（失敗処理ルーチンの先頭番地に対応する）となる．なお，モード宣言がない場合には入出力，型記述のすべての可能な組合せで宣言されているとする．

notrail 宣言は非決定的な実行をする節から呼び出される節では使えないのに対して，nocheck 宣言は先に述べた仮定が成り立つ限り非決定的な実行を含むプログラム（例えば８クィーン）でも使うことができる．nocheck 宣言が使えるかどうかは熟練した Prolog プログラマならある程度判定できると思われるが，一般的には難しい．上述の仮定は，未定義変数どうしの統一化（例えばゴール引数中の２つの異なる未定義変数どうしの統一化）があると成り立たなくなる．実行時の動的な解析によって，nocheck 宣言が使えるかどうか判定する必要がある．この解析は変数への値の代入の可能性のある中間言語命令の中で，代入の度にチェ

```
mode append(in(nil,list),in(nil,list),out(nil,list)).
append([],Y,Y).
append([V|X],Y,[V|Z]) :- append(X,Y,Z).
```

(a) 決定的appendのプログラム
Program of deterministic append

```
app@3:
 case_type 0,app@3_0,,app@3_1,,,
           app@3_2
app@3_0:
  deref     0
  execute   app@3
app@3_1:
  trail_check  2
  get_x_val_w  1,2
  proceed
app@3_2:
  get_list_r     0
  unify_x_var_r  3
  unify_x_var_r  0
  trail_check    2
  get_list_w     2
  unify_x_val_w  3
  unify_x_var_w  2
  execute       app@3
```

(b) nocheck宣言なし
Without nocheck declaration

```
app@3:
  case_type 0,,,app@3_0,,,app@3_1
app@3_0:
  get_x_val_w   1,2
  proceed
app@3_1:
  get_list_r    0
  unify_x_var_r 3
  unify_x_var_r 0
  get_list_w    2
  unify_x_val_w 3
  unify_x_var_w 2
  execute       app@3
```

(c) nocheck宣言あり
With nocheck declaration

図 2 コンパイルコードの例（決定的 append）
Fig. 2 Example of compiled codes (deterministic append).

ックするようにしておくことにより行える．コンパイル時のオプションによってこのチェックのコードを生成するかどうかを指定できる．

並列実行の中間言語命令は，プロセスの生成／制御の命令，プロセス間通信のデータ送信／受信命令とからなる．プロセスの生成，プロセス間通信のコンパイル例を**図3**に示す（図3のC，Xはそれぞれ変数C，Xを表す一時変数の番号を指す）．プロセス生成の情報（実行開始アドレスと引数の値），プロセス間通信の送信データはMSB (Message Buffer) に put_MSB 命令によりいったんコピーされてから取り出される．送信時にMSB にコピーされた未定義変数の値領域，リスト，構造体の要素データは受信の際，受信プロセスのグローバルスタック中にコピーされる．

4.2 プロセスの実現

プロセスを構成する各種の領域を，**表1**に示す．プロセス制御ブロック (PCB) が中心であり，その他の領域は PCB からいったんプロセス作業ブロック (PWB) を介して参照される．プロセスを管理する上で，他のプロセスから参照する必要のある情報（プロセスの ID，相互排除用セマフォ，PCB 間のリンク，状態，割当て先プロセッサ番号）を PCB に入れ，それ以外の情報はすべてローカルメモリ中の PWB にいれ参照時間を短縮している．

プロセスの管理はモニタによって行われる．モニタは，プロセスの生成／消滅，実行中断／再開，スケジューリングといった機能を提供する[11]．モニタは，プロセスの状態を決め，それに応じてプロセスのPCBを自由 (free) リスト，スケジュール (scheduled) リスト，実行可能 (ready) リストにまとめる．スケジュールリストと実行可能リストは循環リストであり到着順にプロセスを並べたキューを構成する．

プロセスは，生成時にどれかのスレーブプロセッサに割り当てられると，以後そのスレーブプロセッサでのみ実行される．モニタおよび自由／スケジュール／実行可能リストは各スレーブプロセッサにそれぞれ置かれる．プロセスのスレーブプロセッサへの割当ては，その時点で最も負荷の軽いスレーブプロセッサを選んで行われる．割り当てられているプロセスの数を負荷としている．プロセスのスケジューリングは実行可能プロセス優先としている．まず，実行可能リストにあるプロセスが実行され，実行可能リストが空の時だけ，スケジュールリストからプロセスを選ぶ．実行可能プロセスの方が先に消滅することが多いので，この方法により実行中の負荷，つまり割当てプロセスの数を減らすことができる．

プロセスの生成 fork(p(a))

```
allocate_MSB    2
put_entry_MSB   p@1,0
put_atom_MSB    a,1
fork
```

送信（同期型） C!a

```
allocate_MSB      1
put_atom_MSB      a,0
lock_chan         C
resume_receiver   C
put_message_MSB   C
suspend_sender    C
unlock_chan       C
switch
```

送信（非同期型） C!!a

```
allocate_MSB      1
put_atom_MSB      a,0
lock_chan         C
resume_receiver   C
put_message_MSB   C
unlock_chan       C
```

受信 C?X

```
    lock_chan         C
    check_message     C,L1
    suspend_receiver  C
    unlock_chan       C
    switch
    lock_chan         C
L1: get_message_MSB   C
    get_x_var_MSB     X,C
    resume_sender     C
    deallocate_MSB
    unlock_chan       C
```

図3 コンパイルコードの例（並列実行用組込み述語）
Fig. 3 Example of compiled codes (Builtin predicates for parallel execution).

表1 プロセスを構成する領域
Table 1 Areas to construct process.

	割当てメモリ	内　容
PCB	放送領域	プロセスの状態，PWBのアドレス
PWB	ローカルメモリ	レジスタ退避・一時変数領域 他の領域へのポインタ，スタック
ENV	共有メモリ	ローカル変数，グローバル変数 選択点，トレイル
CODE	ローカルメモリ	オブジェクトコード
CHAN	放送領域	チャネル
MSB	放送領域	メッセージバッファ

PCB: Process Control Block, PWB: Process Working Block, ENV: Environment, CHAN: Channel, MSB: Message Buffer

5. 性能評価

プロセッサ1台で要素100個のリストの append と要素30個のリストの reverse を実行した結果を**表2**に示す．表2で，optimize/no optimize は4.1節で述べた最適化のうち，一時変数を取り扱う get/put 命令の最適化を行った／行わなかったことを示す．この最適化により約2割実行時間が短縮している．また，nocheck/no trail/check はデリファレンスとトレイル

の両方の処理を省いた／トレイルのみ省いた／省かなかったことを示す．notrail で約2割，nocheck で約3割実行時間が短縮している．表2より，プロセッサ1台の逐次実行では，最も速い場合 (optimize, nocheck) で約 18 KLIPS, 最も遅い場合 (no optimize, check) でも約 11 KLIPS の実行速度が得られていることがわかる．

表 2 append と reverse の実行時間
Table 2 Execution time of "append" and "reverse".

	optimize			no optimize		
	nocheck	notrail	check	nocheck	notrail	check
APPEND-100	5.45	6.26	7.87	6.86	7.68	9.29
REV-30	28.3	32.2	39.6	34.8	38.8	46.2

(単位はミリ秒)

8 Queen-all (8 クィーンの全解探索) を実行した結果を**表 3** に示す．nocheck 宣言をつけ，さらに get/put 命令の最適化を行って実行している．並列実行では**図 4** に示したプログラムを実行した．8 クィーンの解を7クィーンの解を求めるプロセスを8つ生成することにより求める．解の個数のカウントと並列実行の終了判定にプロセス間通信を利用している．表3で，同期通信とはこのプロセス間通信をすべて同期型で行ったときの実行時間を示し，非同期通信とは通信をすべて非同期型で行ったときの実行時間を示す．実行時間の計測に際しては入出力を行う組込み述語 (write と nl) を省いている．

表 3 8クィーンの実行時間
Table 3 Execution time of 8 Queen.

逐次実行	並列実行			
	プロセッサ台数	1台	2台	3台
2930	同期通信	3009	1645(1.8)	1298(2.3)
	非同期通信	2989	1501(2.0)	1137(2.6)

(単位はミリ秒，カッコ内は速度向上比)

同期，非同期通信の実行時間の差はそれほどないが，台数効果の点でやや非同期通信がまさっている．同期通信では，受信が先に行われていないとき送信が行われると，送信プロセスの実行中断となりプロセス切替えが生じる．同期通信と非同期通信の差はこのプロセス切替えのオーバヘッドによるものと考えられる．

参考までに述べると，8クィーン問題を実行するのに汎用機上の高速な Prolog 処理系である DEC-10 Prolog (DEC SYSTEM-2060, コンパイラ版) では 1624 ミリ秒[12], Quintus Prolog (VAX-11/785) では 3660 ミリ秒かかる (PARK は Prolog の実行専用のハードウェアを備えていないため，これら汎用機上の Prolog 処理系と比較している)．本論文の処理系は実験的なものでこれらの実用的な処理系とは単純に比較できないが，クロック 8 MHz の 68000 上の処理系でこれだけの実行速度が得られたのには大きな意義があると考えられる．

なお，著者らが以前行った K-Prolog[1] と実行速度を比較すると，プロセッサ1台の時の4クィーンの全解探索が K-Prolog で 4678 ミリ秒 (パイプライニング並列) と 3931 ミリ秒 (OR 並列)，PARK-Prolog で 28 ミリ秒 (同期通信) と 27 ミリ秒 (非同期通信) であり，100 倍以上の速度向上が得られている (台数効果は最高値で K-Prolog が 2.3 倍，PARK-Prolog

```
nocheck.

mode queen(in(int),out(int)).
queen(N,S) :-
  qlist(N,L), mkchan([C]), fork(q1(L,C)), qa(N,C,S,0).

mode qlist(in(int),out(nil,list))
qlist(0,[]) :- !.
qlist(N,[N|L]) :- N1 is N-1, qlist(N1,L).

mode qa(in(int),recv(atom),out(int),in(int)).
qa(0,C,S,S) :- !, rmchan([C]), write(S), nl.
qa(N,C,S,S1) :- C?A, qa1(N,C,S,S1,A).

mode qa1(in(int),recv(atom)),out(int),in(int),in(int)).
qa1(N,C,S,S1,end) :- !, N1 is N-1, qa(N1,C,S,S1).
qa1(N,C,S,S1,sol) :- S2 is S1+1, qa(N,C,S,S2).

mode q1(in(nil,list),send(atom)).
q1(L,C) :- select(L,U,V), fork(q2(V,[U],C)), fail.
q1(_,_).

mode q2(in(nil,list),in(list),send(atom)).
q2(X,Y,C) :- q3(X,Y,Q), write(Q), nl, C!sol, fail.
q2(_,_,C) :- C!end.

mode q3(in(nil,list),in(list),out(nil,list)).
q3([],Y,Y) :- !.
q3(X,Y,Q) :- select(X,U,V), safe(U,Y,1), q3(V,[U|Y],Q).

mode select(in(nil,list),out(int),out(nil,list)).
select([U|V],U,V).
select([X1|X2],U,[X1|V]) :- select(X2,U,V).

mode(in(int),in(nil,list),in(int)).
safe(_,[],_).
safe(U,[P|Q],N) :-
  U =¥= P+N, U =¥= P-N, M is N+1, safe(U,Q,M).
```

図 4 Nクィーンのプログラム
Fig. 4 Program of N Queen.

が 1.9 倍). これは K-Prolog 処理系がインタプリタ方式でプロセッサがクロック 5 MHz の 8086 であったこともあるが, 表 3 で逐次実行とプロセッサ 1 台の並列実行の時間の差があまりないことからもわかるように, ゴールの逐次実行をプロセスとしてプロセスの生成・切替えを少なくした PARK-Prolog の方式によるところが大きいと考えられる.

6. おわりに

本論文では, 並列論理型言語 PARK-Prolog とその処理系のマルチプロセッサシステム PARK 上での実現および性能評価について述べた. PARK-Prolog は, 逐次型 Prolog の最適化技法をそのまま利用し, さらに並列実行でのプロセッサ台数効果により実行速度を向上させることをねらって設計されている. PARK-Prolog の処理系の性能は, append と reverse の逐次実行で約 18 KLIPS であった. また, 4 クィーン全解探索の並列実行では以前に著者らが実現した K-Prolog と比べると 100 倍以上の実行速度の向上が見られた. さらに, 8 クィーンの全解探索を並列実行するとプロセッサ 3 台での実行で 1 台の時と比べ 2.6 倍の速度向上が得られ, 汎用機上の高速な逐次型 Prolog 処理系である DEC-10 Prolog, Quintus Prolog の実行速度を上回った. これらの結果により, 逐次型 Prolog の最適化技法を並列実行下でも生かして論理型言語の実行速度を向上させるという著者らのねらいは達成されたと考える.

謝辞 Quintus Prolog のデータを送っていただいた新世代コンピュータ技術開発機構 (ICOT) の龍 和男氏に感謝する. 本研究は一部, 文部省科学研究費補助金 (奨励研究(A)) によっている.

参考文献

1) 松田秀雄, 田村直之, 小畑正貴, 金田悠紀夫, 前川禎男: 並列 Prolog 処理系 "K-Prolog"の実現, 情報処理学会論文誌, Vol. 26, No. 2, pp. 296-303 (1985).
2) Shapiro, E. Y.: A Subset of Concurrent Prolog and Its Interpreter, ICOT Technical Report No. 3 (1986).
3) Clark, K. L. and Gregory, S.: PARLOG: Parallel Programming in Logic, Research Report DOC 84/4, Imperial College (1984).
4) Ueda, K.: Guarded Horn Clauses, *Proc. of the Logic Programming Conference '85 9.3*, pp. 225-236 (1985).
5) 上田和紀: 全解探索プログラムの決定的プログラムへの変換, 日本ソフトウェア科学会第 2 回大会論文集, pp. 145-148 (1985).
6) 松田秀雄, 小畑正貴, 増尾 剛, 金田悠紀夫, 前川禎男: 並列 Prolog マシン PARK について—ハードウェア構成と Prolog 処理系—, *Proc. of the Logic Programming Conference '85, 2.4*, pp. 39-46 (1985).
7) INMOS Limited., occam プログラミングマニュアル, 啓学出版, 東京 (1984).
8) Pereira, L. M. and Nasr, R.: Delta-Prolog: A Distributed Logic Programming Language, *FGCS '84*, pp. 283-291 (1984).
9) Warren, D. H. D.: An Abstract Prolog Instruction Set, SRI Technical Note 309 (1983).
10) 田村直之, 浅川康夫, 小松秀昭, 黒川利明: JSI AI ワークステーション(6)—Prolog コンパイラの最適化技法, 第 33 回情報処理学会全国大会論文集, IQ-6 (1986).
11) 松田秀雄, 増尾 剛, 金田悠紀夫, 前川禎男: マルチプロセッサシステム PARK の OS について, 情報処理学会オペレーティング・システム研究会資料, 33-7 (1986).
12) 奥乃 博: The Report of The Third Lisp Contest and The First Prolog Contest, 情報処理学会記号処理研究会資料, 33-4 (1985).

(昭和 62 年 2 月 3 日受付)
(昭和 62 年 12 月 9 日採録)

松田 秀雄 (正会員)
昭和 34 年生. 昭和 57 年神戸大学理学部物理学科卒業. 昭和 59 年同大学院工学研究科システム工学専攻(修士課程)修了. 昭和 62 年同大学院自然科学研究科システム科学専攻(博士課程)修了. 昭和 62 年より神戸大学助手(システム工学科). 学術博士. 論理型プログラミング言語, 並列推論マシンなどの研究に従事. 日本ソフトウェア科学会会員. ICOT PIM ワーキンググループ委員.

小畑 正貴 (正会員)
昭和 32 年生. 昭和 55 年神戸大学工学部電子工学科卒業. 昭和 60 年同大学院自然科学研究科システム科学専攻(博士後期)修了. 学術博士. 現在, 岡山理科大学講師. 計算機アーキテクチャ, 並列処理などの研究に従事. 電子情報通信学会会員.

増尾　剛（正会員）

昭和36年生．昭和60年神戸大学工学部システム工学科卒業．昭和62年同大学院システム工学専攻修士課程修了．同年NTT入社．現在NTTソフトウェア研究所所属．知能的プログラミング環境の研究に従事．

金田悠紀夫（正会員）

昭和15年生．昭和39年神戸大学工学部電気工学科卒業．昭和41年神戸大学大学院電気工学専攻修士課程修了．昭和41年電気試験所（現電総研）入所．電子計算機研究に従事．昭和51年神戸大学工学部システム工学科，現助教授．工学博士．コンピュータシステムのハードウェア，ソフトウェアの研究に従事．高級言語マシン，並列マシン，AIに興味を持っている．

前川　禎男（正会員）

昭和6年生．昭和29年大阪大学工学部通信工学科卒業．昭和34年同大学院工学研究科博士課程修了．大阪大学助手を経て，昭和36年神戸大学助教授（電気工学科）．昭和47年同大学教授（電子工学科）．現在，同大学システム工学科勤務．システム情報講座担当．この間，システム理論，高級言語マシン，人工知能などの研究に従事．工学博士．電子情報通信学会，電気学会，計測自動制御学会，人工知能学会などの正会員．