ICOT Technical Report: TR-572

TR-572

KL1とPIMOS

近山　隆

July, 1990

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191－5
Telex ICOT J32964

Institute for New Generation Computer Technology

KL1 と PIMOS

KL1 and PIMOS

近山 隆
Takashi Chikayama
(財) 新世代コンピュータ技術開発機構
Institute for New Generation Computer Technology

Abstract

第五世代計算機システム研究開発プロジェクトの一部として行なっている並列推論システムの研究開発状況について，プログラム言語とオペレーティングシステムを中心に報告する．

第五世代計算機システムプロジェクトは高度知識情報処理に必要な基礎技術を確立することを目的としている．その中で並列推論システムは知識情報処理のために強力な処理能力を提供するハードウェアと，そのハードウェアを効率的に運用するためのソフトウェアの基礎技術の確立を目的としている．

従来の並列ソフトウェア技術は逐次処理ソフトウェア技術の延長線上に並列技術を位置付けていた．我々は並列推論システム研究開発にあたって，こうした従来のアプローチでは高並列処理のためのソフトウェア技術確立には不適当であると考え，並列処理のためのソフトウェア技術を構築する道をとった．

本稿では従来のアプローチの問題点を解析し，そうした問題点を第五世代計算機プロジェクトではどのように解決した (あるいはするつもり) かについて述べる．また，研究開発の現状と計画についても報告する．

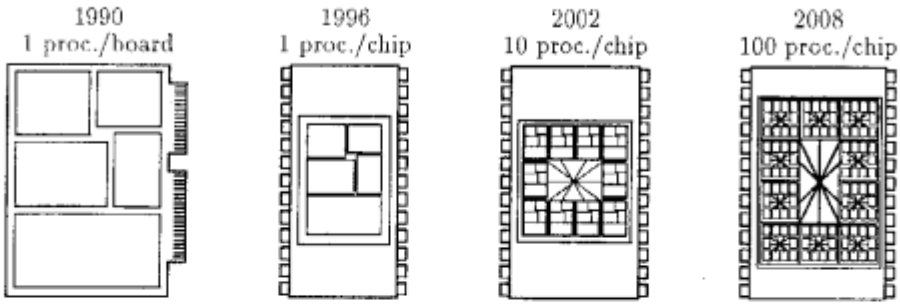


図 1: ハードウェアの集積度向上予測

1 はじめに

第五世代計算機システムプロジェクトは高度知識情報処理に必要な基礎技術を確立することを目的としている．その中で必要になる技術は大別して以下のふたつである．

- 知識処理に必要な高度な問題解決技術
- 高度な問題解決に必要な大容量処理能力の提供

第五世代プロジェクトの一部である並列推論システムはこの後者のためのハードウェア，ソフトウェアの両面での技術の確立を目指すものである．

近年のハードウェア技術の進歩は，およそ 3 年に 4 倍の実装密度向上をもたらしてきた．現在の技術では大型計算機に比べ得る能力の計算機を回路基板 1 枚に実装することができる．近年の実装密度向上速度をいくぶん控え目に 6 年で 10 倍と見積もって外挿すると，2008 年には 100 台の計算機がチップひとつに入ることになる．チップのコストに占めるデザインコストの割合がますます増大している状況からして，チップ上に小規模なプロセサのコピーを多数実装する方法は，同じ大きさの大規模なプロセサを設計するのに比べてはるかに容易になるものと予想され，もし両者が同程度の性能になるならばマルチプロセサシステムの方がコスト面で有利になる．このことから，21 世紀初頭には絶対的な処理能力だけではなく，小さなシステム，たとえば携帯型の計算機などにおいてもコスト面で並列処理が有利になる可能性が大きい．

これに対し，並列処理ソフトウェア技術はまったく未熟な状況にある．特に知識処理領域に属する複雑な問題を解くためのソフトウェア技術はまったく不満足な状態にある．これは既存の逐次処理技術の改良拡張として並列処理ソフトウェア技術を構築しようとする従来のアプローチに問題があるものと我々は考える．そこで我々は，並列処理のためのソフトウェア技術を，アルゴリズム，プログラム言語，オペレーティングシステムなどすべてについて再構築する方針をとった．

以下では従来のアプローチにどのような問題があるかを解析し，それに基づいて我々がどのようなアプローチをとったかを述べる．また，並列推論システム，特にプログラム言語 KL1 とオペレーティングシステム PIMOS の開発状況と計画について述べる．

2 並列処理ソフトウェア技術へのふたつの道

表 1: 並列処理ソフトウェア技術へのふたつの道

	従来の方法	必要な方法
技術	逐次技術の延長	新たな並列技術
並列実行	指定時	通常
逐次実行	通常	指定時
結果のシステム	ぎこちない	自然
技術的連続性	○	×

並列処理ソフトウェア技術の確立にはふたつの道が考えられる．従来多くとられてきた方法は，逐次処理技術を並列処理向きに拡張しようという方針である．もうひとつは，並列処理ソフトウェア技術を一から作り直そうというもので，第五世代プロジェクトではこの方針をとっている．

従来の方針は既存の逐次処理技術の延長線上にあるので，移行をよりスムースにできるという利点がある．実際，小規模な並列システム向きにはこの方針は有利かも知れない．しかし，大規模並列処理を目指すには不適当である．本節ではなぜこの方針が大規模並列処理には不適当なのかを解析する．また，並列ソフトウェア処理技術を最初から再構築しようという我々の方針はどのようなものなのかについても述べる．

2.1 アルゴリズム

ディジタル計算機の登場以来，広範囲の問題について多くの効率的なアルゴリズムが提案され用いられてきた．従来の並列処理へのアプローチはこうしたアルゴリズムを並列処理向けに仕立て直そうというものである．

逐次処理アルゴリズムを並列化するにはふたつの問題がある．ひとつは，多くの場合逐次的に実行せざるを得ない長いパスがあり，そのために並列性が上がらないということである．しかし，この問題はアルゴリズムを多少手直しすることで解決できる場合が多い．

もっと大きな問題は，多くの逐次アルゴリズムは一様に一定の手間でアクセスできるメモリがあることを前提としていることである．大規模並列システムでどこからでも一定の手間でアクセスできるメモリを「一定の手間」をあまり大きくせずに実現することは困難というより不可能である．このようなアルゴリズムを並列処理向きに手直しするのは不可能である．このため，逐次処理向きには最適なアルゴリズムが，並列処理としてはまったく使いものにならないことは多い．

並列システムのためのアルゴリズムは最初から考え直す必要がある．この際並列度と共に，通信の局所性を常に考慮する必要がある．

逐次アルゴリズムの改訂版も，もちろん有力な候補である．だが，元のアルゴリズムが高効率であることは必ずしも並列化した版に反映しないことを忘れてはならない．むしろ，高度に最適化アルゴリズムは，逐次処理向きの最適化が進み過ぎており，効率的に並列化する可能性を失っていることも多いのである．

2.2 言語

従来, 逐次処理向きのプログラム言語にいくつかの機能を付加し, 並列処理が可能なものにする方針が多くとられてきた.

このような素性の言語では, 特別な言語機構を用いて指定した時にだけ並列処理が行なわれる. こうした機構は言語の一部ですらなく, 単にライブラリルーチンとして用意しているだけの場合もある. このようにわざわざ並列処理を指定する方法をとると, 並列処理プログラムはぎこないものとなり, 後で負荷分散や通信オーバヘッド削減のためにプログラムを再構成するのが非常にやりにくくなる.

このような言語を用いるもうひとつの問題は, 同期のバグが頻発することである. こうした言語では同期についても, 特別な機構かライブラリルーチンの呼び出しでで指定した時だけ行なわれる. 並列プログラムでの同期のバグの主要原因は以下のふたつである.

1. まだ値を書いていない変数を読んでしまった
2. まだ値を読んでいない変数を上書きしてしまった

データそのものでの同期機構のない言語では, かわりに実行順序による同期を用いなければならない. 同期の責任は言語システムではなくプログラマにあるので, 第一種の同期バグの原因となる. 第二種のバグの原因は上書き (代入) が言語仕様に入っていること自体である.

並列処理システム用のプログラム言語は最初から並列実行が自然に書き表せるように設計すべきである. 言語は本質的に並列, つまり, 言語の提供する機構はすべて原則として並列実行を前提とし, 必要ならそこだけ逐次性を指定すべきである.

同期の問題を解決するためには, 同期は言語機構の中に暗黙のうちに含まれているようにすべきであり, プログラム言語のレベルでは変数の上書きなどがあるべきではない.

2.3 オペレーティングシステム

従来の並列システム用オペレーティングシステムは, 逐次システム用のものの基本設計をそのままに, 並列実行のための機構を追加したようなものだった. たとえ逐次システム用のものであっても, オペレーティングシステムは (疑似) 並列実行を前提として設計されているのでこのようなことが可能なのである.

しかし, この方法にはふたつの大きな問題点がある. ひとつは, オペレーティングシステムのユーザインタフェースが, プログラムの逐次実行を前提としたものになっていることである. たとえば, オペレーティングシステムに依頼した処理の完了は, 依頼の手続き (スーパバイザコール) の完了として通知されるようになっている. オペレーティングシステム自体は並列処理を前提に作られていても, その下で動く応用ソフトウェアは逐次実行だったため, これでも問題なかったのである. ところが, 大規模並列システムになると応用ソフトウェアもその中で並列動作するようになり, これはプログラム言語について上述したと同様の問題を引き起こす.

もうひとつの問題は, 逐次システム用のオペレーティングシステムでは管理方針が逐次処理や一定の手間でアクセスできるメモリに依存する最適化が行なわれていると

いうことである．逐次システムではすべての管理情報を一括集中管理するのが簡単であるし効率的である．これは大規模並列システムにとっては最悪の方法である．大規模並列システムであるプロセサ (またはプロセサのグループ) に管理を集中したりすると，そのプロセサがしなければならない管理業務が多くなり過ぎ，システム全体のボトルネックになってしまう．また，システム中のすべての活動が管理しているプロセサとの間で通信を行なうことになり，通信のボトルネックにもなる．

もしこの問題がオペレーティングシステム内部の管理方式だけに閉じているのなら，ユーザインタフェース仕様を変えることなくコンパチビリティを保ったままシステムを書き換えることもできる．しかし，そうしたオペレーティングシステムが提供する機能の仕様は管理方式を強く反映したものになってしまている．プロセスをプロセス番号という整数値で表すなどは，集中管理の影響の例である．

並列システムのためのオペレーティングシステムは，ユーザインタフェースを含めて再設計すべきである．ユーザインタフェースは並列プログラム言語と整合性が良くなるように設計すべきで，逐次実行をインタフェース設計の前提にすべきではない．ボトルネックを防ぐために，できる限り管理は分散化すべきである．この方針もユーザインタフェース設計に影響を及ぼす．

3 並列推論システムの設計

上述した並列処理ソフトウェア技術へのふたつのアプローチの比較に基づき，第五世代プロジェクトの並列推論システムの設計にあたっては，新たに必要な技術体系を構築する道を選んだ．この節では設計に当たって考慮した特徴的な諸点について述べる．

3.1 アルゴリズム

並列アルゴリズム設計においては，アルゴリズムの並列度に加えて，実際に利用可能な並列度 (簡単にいえばプロセサの数) を考慮すべきである．いま計算量が $c(n)$ (n は問題のサイズ) で並列度が $p(n)$ のアルゴリズムがあったとする．このアルゴリズムを用いた実行総時間 $t(n)$ は通信オーバヘッドを除き理想的な分散を仮定すると以下の式で表される．

$$t(n) = c(n)/p(n)$$

この $t(n)$ は種々のアルゴリズムを比較する基準になるだろう．この基準では計算量が多くてもそれを上回る並列度があるアルゴリズムが良いアルゴリズムと判定される．

しかし，これは $p(n)$ が現実に利用可能な並列度 p_p より小さい場合のことである．利用可能な並列度も考慮すると，実行時間は以下の式のようになる．follows.

$$t(n) = c(n)/\min(p(n), p_p)$$

現実に使える並列性に限りがあると，速く $c(n)$ が大きくなるようなアルゴリズムは $p(n)$ にかかわらず大きな n については不利である．

このことから，十分大きな問題については総計算量が少しでも大きくなった並列アルゴリズムよりも，逐次アルゴリズムの方が良いということになる．したがって，並列アルゴリズムの設計に当たっては，しばしば混合型の戦略をとる必要がある．つまり，

高いレベルでは並列アルゴリズムを用い，現実の並列性を使い切ってしまった後は逐次アルゴリズムに切替えるわけである．

3.2 言語

並列推論システムの核言語として，並列論理型言語 KL1 [2] を設計した．KL1 は GHC [7] のフラット版を元に，さまざまな拡張を施した言語である．

GHC は生まれながらの並列言語で，暗黙の同期機構を持ち副作用はないので，並列処理システム用言語設計の基礎としては理想的である．

KL1 の Flat GHC に対する拡張点は，以下のふたつに分類できる．

- メタレベル実行制御
- 効率的実行

本節はこうした GHC に対する拡張点について述べる．

3.2.1 メタレベル制御

オペレーティングシステムの記述のためには，メタレベルからプログラム実行を制御する機能は不可欠である．従来のオペレーティングシステムはこのために低レベルの機構を直接取り扱うことが多いが，これはプログラムのポータビリティを損なうので不適当である．メタレベル制御機能がオペレーティングシステムだけのものならそれでもよかろう．しかし，並列システムでは応用ソフトウェアでメタレベル制御を行ないたい場合が，逐次システムよりずっと多くなるのである．

荘園機能

従来のオペレーティングシステムでは，プロセスが実行管理の単位であると同時に並列性制御の単位でもあった．我々の設計では実行管理単位と並列制御単位をを別々にした．GHC のプログラムは普通数多くの細粒度プロセスが並列に互いに交信しあって実行を進める構成をとる．KL1 ではこのプロセス群をグループとして扱う「荘園」と呼ぶ構造を管理の単位として導入した．これにより，管理オーバヘッドは細粒度プロセスを直接管理単位とするのに比べて，はるかに小さくなる．

荘園は与えられたプログラムを実行するインタプリタだと考えて良い．荘園は二本のインタフェースストリームを持つ．一本は荘園から外に出ていくストリーム (報告ストリーム) で，プログラム実行に関わるいろいろな事象を報告するのに使う．もう一本は荘園に向かうストリーム (制御ストリーム) で，メタレベルの実行制御コマンドを与えるのに使う．荘園はいくらでもネストすることができるので，プログラムは任意レベルのメタ / オブジェクト階層を持つことができる．

実行時間のような基本的な計算資源も荘園機構を通して管理する．[1] 荘園は許された資源量を消費し尽くすまで実行を継続できる．資源を完全に消費し切ってしまう前に，資源の不足が報告ストリームから伝えられる．荘園を制御しているプログラムは，状況に応じて制御ストリームから資源消費許容量を増やしたり，実行を止めたりすることができる．

[1] 現在の処理系は物理的実行時間の代わりにゴール生成とリダクションの数を用いている．

このような機構は従来オペレーティングシステムが提供するのが普通だったので，プログラム言語のレベルに導入することには疑問もあろう．しかし，並列システムでは応用ソフトウェア内で資源管理が必要になることが多く，こうした機構の効率的な実現をプログラム言語の機能として提供し，応用ソフトウェアから直接利用できるようにするのが得策であろうと考えたのである．

優先度

もうひとつのメタレベル実行制御に関わる重要な拡張は，優先度機構の導入である．

逐次プログラム中の実行順序指定は，往々にして不可欠な実行順の指定ではなく，計算の優先関係を反映することがある．たとえば，ある問題を解くのにふたつの方法があったとしよう．方法 A は必ずしも解に至るとは限らないが，うまくいくにしろいかないにしろ，すぐ終るものである．方法 B ははるかに非効率だが，必ず解を見つけられる．逐次システムならこのような場合，まず方法 A を試し，だめだったら B を試す，というのが最良策だろう．

しかし，並列システムでは B を A の後で試すというのは最良策ではない．方法 A は利用可能な計算資源 (たとえばプロセサ) すべてを必要とするとは限らないからである．このような場合，方法 B も A と並列に試すべきである．だが，B の試行は結局不要であるとわかるかもしれないので，それが A の実行を邪魔してはいけない．このことは A に B より高い優先度を与えることで実現できる．

優先度機構は一般に「見込み計算」の制御に便利である．見込み計算とは，他の部分の計算結果によって必要になるかも知れないし不必要とわかるかも知れないような計算のことである．見込み計算を並列に実行するのは，本当に必要となる場合には有利である．見込み計算には見込みでない計算よりも低い優先度を与えるべきである．そうしないと，その実行によって本来必要な計算が使うべき資源を使ってしまい，完全に逐次実行を指定した場合より性能を落すことになりかねない．

実際，種々の実験的応用ソフトウェアの開発を通じて，優先度機構の有用性が検証されている．

3.2.2 効率的実行

ピュア Lisp やピュア Prolog のような，いわゆる「ピュア」な言語は実行が非効率な欠点を持っていた．これは言語処理系がお粗末だったということもあろうが，より本質的には言語の設計，特に副作用がないことに問題がある．こうした言語は破壊的な代入ができないため，配列要素の更新を一定時間で行なうことができない．このため，ランダムアクセスメモリの特性を十分引き出した手続き的アルゴリズムを，同じ計算量のまま記述することができない．

GHC もピュアな言語なので，同じ問題を抱えている．アルゴリズムについての節で述べたように，手続き言語と同じ計算量を実現できなければ，大きな問題については逐次システムより非効率になってしまう．

この問題を解決するために，KL1 では言語のセマンティクスをピュアに保ったままで配列要素を一定時間で更新できるようなプリミティブを導入した．このための組込み述語は以下のような形をしている．

```
set_vector_element(Vect, Index, Elem, NewElem, NewVect)
```

この述語は配列 Vect, インデクス値 Index と新しい要素 NewElem を与えると, そのインデクスの元の要素 Elem と, 元の配列とほとんど同じだが指定した要素だけが異なっているような新しい配列 NewVect を返す.

他にも元の配列 Vect への参照が残っていないとは限らないので, 単純に処理系を作ると, 言語のピュアなセマンティクスを保つには, まったく新しい配列を作りほとんどすべての要素をコピーすることになる. しかし, もしこの組込み述語に渡された配列への参照パスが最後のものであることがわかれば, 直接破壊的に更新しても問題ない. KL1 の実際の処理系では簡略化した効率的な参照カウント方式 [1] によってそのような状況を検出している. これにより, プログラムで配列への参照をひとつだけに保つ限り, 配列要素の更新は一定時間でできるようになっている.

ランダムアクセスメモリは単一参照の配列でエミュレートできるので, どんな手続き的アルゴリズムでも, 同じ計算量のまま KL1 で記述することができるわけである. もちろん単一参照に保つには並列性を犠牲にしなければならない. しかし, 同じ計算量を保たねばならないのは, 上述した通り, 実際に利用可能な並列度を使い尽くした後であるから, 問題はない.

3.3 オペレーティングシステム

並列推論マシンのオペレーティングシステム PIMOS [2] は上述の方針にしたがって開発した.

PIMOS は数多くの動的に生成したプロセスが互いにストリームで通信し合うような構成 [6] で, KL1 で記述した. PIMOS とユーザプログラムのインタフェースもやはりストリームである.

資源管理の単位はタスクと呼び, 上述の KL1 の荘園機構を用いて実現している. タスクは普通の荘園と同様いくらでもネストできる. ユーザのタスクひとつひとつに対応して PIMOS のプロセスがあり, これらのプロセス群はタスクの階層構造に対応する木構造をなしている. 各入出力デバイスにも PIMOS のプロセスが対応し, やはり木構造の一部になっている. この木構造中のプロセスは複数のプロセサに分散し, 管理のボトルネックを解消している.

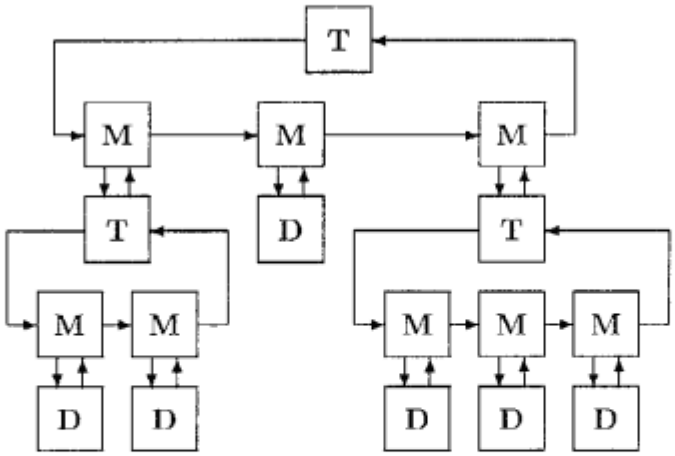
4 研究開発の現状と計画

並列推論マシンの実験機 Multi-PSI, その上の KL1 言語処理系 [5], そして PIMOS が利用可能になった直後から, いくつかの実験的応用ソフトウェア開発が始まった. 以下にそのうちのいくつかについて簡単に述べる.

詰め込みパズル: 全解探索問題. 問題解決手法自体は単純な OR 並列である. できる限りの並列実行による速度向上を得るために, 種々の負荷分散アルゴリズムを試した.

最短経路: 最適解探索問題. 用いたアルゴリズムは Dijkstra のアルゴリズムの並列版である.

詰め碁: 囲碁の死活問題を解く. 見込み計算を含む $\alpha\beta$ 探索の並列版を用いている.



T: タスク管理プロセス
M: 資源モニタープロセス
D: デバイスハンドラ

図 2: PIMOS の階層的資源管理

表 2: 利用中および計画中の並列推論システム

時期	システム	ハードウェア	プロセサ	ピーク性能
1987	PDSS	汎用機	1	~10 KRPS
1988	PIMOS/S	PSI-II	1	~150 KRPS
1988	PIMOS/M	Multi-PSI	64	~10 MRPS
1991	PIMOS/P	PIM	~512	>100 MRPS

RPS: リダクション毎秒

PAX: 自然言語のボトムアップ構文解析をする．プログラムは文法規則から自動的に生成したものである．

これらのうちにはプロセサ台数にほぼ比例する速度向上を得たが，あまり良好な速度向上をまだ得られていないものもある．最近ではより大規模で実用的な問題を扱うプロジェクトも始まっている．

現在 ICOT や関係研究機関では，KL1 言語処理系と PIMOS の最新版をのせた Multi-PSI 数セットを利用している．最大構成である 64 プロセサの Multi-PSI 上の KL1 処理系は，ピーク性能で約 10 メガリダクションの実行速度である．

新しい並列推論ハードウェア PIM [4] も開発途上である．再大規模のものは 512 台のプロセサで，Multi-PSI の十倍以上の性能を見込んでいる．

1990 年 5 月には 2 日間のワークショップ (1 日のチュートリアルつき) を開催し，並列推論システムのユーザとインプリメンタが集まって情報交換する機会を持った [3]．多くは開発途上であるが，種々の応用システムと，いくつかの新たな並列処理ア

ルゴリズムが報告された.

5 おわりに

第五世代計算機プロジェクトの並列推論システムの研究開発に当たって, 我々は従来の逐次処理技術の修正ではなく, 並列ソフトウェア技術を根本から再構成するアプローチを選んだ. ハードウェアの実験機, 言語処理系そしてオペレーティングシステムは応用ソフトウェア研究に提供済みで, いくつかの応用プロジェクトが進行中である.

我々の経験から, 暗黙の同期機構を持ち副作用のない言語を用いれば, 並列ハードウェア上で動作する並列ソフトウェアを記述するのは難しくないと言える. オペレーティングシステムの開発中も同期のバグはほとんど出なかった.

一方, プログラムを並列ハードウェアの上で効率良く動作させるのはまた別の問題である. 何にでも使える効率的な負荷分散方式は見つかっていない. 実用的な並列処理ソフトウェア技術を確立するには, まだまだ種々の応用ソフトウェアを通じた経験を蓄積していかねばなるまい.

謝辞

並列推論システムの研究開発は第五世代計算機システムプロジェクトの一環として行なっているもので, ICOT や協力機関のここに名前をあげるには多過ぎるほどの研究者が関わっている. これらの方々全員とシステムの利用者, わけても不完全な初期の版の忍耐強い利用者に謝意を表したい.

参考文献

[1] T. Chikayama and Y. Kimura. Multiple reference management in flat GHC. In *Proceedings of 4th International Conference on Logic Programming*, 1987.

[2] T. Chikayama, H. Sato, and T. Miyazaki. Overview of the parallel inference machine operating system (PIMOS). In *Proceedings of FGCS'88*, Tokyo, Japan, 1988.

[3] K. Furukawa and K. Taki. Proceedings of KL1 programming workshop '90. ICOT Technical Report, ICOT, 1990. *in Japanese.*

[4] A. Goto, M. Sato, K. Nakajima, K. Taki, and A. Matsumoto. Overview of the parallel inference machine architecture (PIM). In *Proceedings of FGCS'88*, Tokyo, Japan, 1988.

[5] K. Nakajima, Y. Inamura, N. Ichiyoshi, K. Rokusawa, and T. Chikayama. Distributed Implementation of KL1 on the Multi-PSI/V2. In *Proceedings of the Sixth International Conference on Logic Programming*, 1989.

[6] E. Shapiro and A. Takeuchi. Object oriented programming in Concurrent Prolog. ICOT Technical Report TR-004, ICOT, 1983. Also in New Generation Computing, Springer-Verlag Vol.1 No.1,1983.

[7] K. Ueda. Guarded Horn Clauses: A parallel logic programming language with the concept of a guard. ICOT Technical Report TR-208, ICOT, 1986.