オペレーティングシステム PIMOS と 核言語 KL1

近山 隆
(財) 新世代コンピュータ技術開発機構
東京都 港区 三田 1-4-28
chikayama@icot.or.jp

概要

第五世代計算機システム (FGCS) プロジェクトは, 高性能知識処理システムに必要な基礎技術の構築を目的とした, 日本の国家プロジェクトである. 並列推論システムサブプロジェクトは, 知識処理分野に必要な大容量処理能力を提供する並列ハードウェア技術と, その有効活用のためのソフトウェア技術を確立することを目指すものである. この中で基本ソフトウェアシステムは, 知識処理応用ソフトウェアを記述するのに適当なプログラム言語を提供し, 大規模並列計算機システム上でのプログラム実行やソフトウェア開発のための快適な環境を提供するためのものである.

並列ハードウェアを制御するための拡張を施した並列論理型言語をシステムの核言語として設計した. 好適なソフトウェア開発環境を提供するオペレーティングシステムを設計し, 核言語で実装した. 本論文は FGCS プロジェクトのこの領域における研究開発の概要を述べる.

1 はじめに

第五世代計算機システム (FGCS) プロジェクトは, 高性能知識処理システムに必要な基礎技術の構築を目的とした, 日本の国家プロジェクトである. プロジェクトの最終目標の達成に必要な技術の内, もっとも重要なのは以下のふたつである.

- 知識処理のための問題解決の諸技法
- その実装に必要な処理能力

並列推論システムは後者のためのハードウェア・ソフトウェア両面の技術を確立することを目的とする.

近年のハードウェア技術の進展により, 次世紀初頭にはマルチプロセサシステムが, 絶対的な処理能力だけではなくコスト性能比でも有利になることが予想される. また, 高性能知識情報処理システムに必要な計算能力を提供するためには, マルチプロセシング以外に方法はなさそうである.

ところが, 並列処理のためのソフトウェア技術は全く未熟な状態にある. ことに, 知識処理のような領域の複雑な問題を解く並列ソフトウェア構築技術は, 未だ到底満足いかないレベルにある. この現状の原因の少なくとも一部は, 従来の並列ソフトウェア技術へのアプローチ, すなわち既存の逐次処理技術を並列処理向けに修正していくアプローチにあった. こうしたアプローチではなく, 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を含む, 並列処理のために新たに設計し直したソフトウェア技術体系の確立が必要なのである.

こうした新技術体系の基礎として, 並列処理ハードウェア上でのプログラム実行制御機能を持った並列論理型言語を核言語として設計した. また, 並列応用ソフトウェアのための好適な環境を提供するオペレーティングシステムを設計し, 核言語を用いて実装した. 本論文は FGCS プロジェクトのこの領域での研究開発の概要を述べる.

本論文では以下, 設計方針を第 2 節に, 核言語の設計について第 3 節に, オペレーティングシステムの設計について第 4 節に述べる. この言語とオペレーティングシステムの使用経験について, 第 5 節に述べ, 今後の研究開発の方向について第 6 節に, 最後に結論を述べる.

2 方針

2.1 ミドルアウト・アプローチ

計算機システムの設計にあたっては, ふたつのまったく異なるアプローチが考えられる. ひとつはトップダウン・アプローチで, 解くべき問題から始めて次第に低レベルの計算機アーキテクチャや, 場合によっては電子デバイスに至るまで, 各レベルにおいて上位レベルにもっとも適合する設計を目指すものである. もう一方はボトムアップ・アプローチで, 利用可能なデバイス技術から始め, 低レベルの技術の最適な利用法を考えながら, 最後には適切な応用分野を見つけるというものである.

どちらのアプローチも, それだけではうまくいかない. トップダウン・アプローチでは, 各レベルの設計にあたって, より低レベルで適切に実装できることを洞察できなくてはならない. ボトムアップ・アプローチでは, 各レベルの設計にあたって上位レベルで何が必要になるのか, 最終的にはどのような応用分野に適したものになるのかが洞察できなくてはならない.

知識情報処理という広範で漠然とした目標を持つプロ

ジェクトでは，このような洞察力を持つことは難しい．そこで，我々はミドルアウト・アプローチ，つまりある中間レベルの設計を先に行ない，そこから上位と下位の両方向に同時に研究開発を進めるアプローチを取ることとした．もちろん中間レベルを適切に選ぶこと，そしてそのレベルを適切に設計することは容易ではない．しかし，このプロジェクトの場合，これ以外に適切な方法があるとは思えなかった．

2.2 核言語

中間レベルとして選んだのは，プログラム言語のレベルである．このレベルには以下のような利点がある．

- プログラム言語レベルは，応用ソフトウェアとハードウェア実装という両端のどちらからもそれほど遠くない．
- プログラム言語レベルは，他のレベルに比べて容易に厳格な仕様を与えることができる．

このミドルアウト・アプローチの出発点として設計したプログラム言語を核言語と呼ぶ [Ueda and Chikayama 1990]．

1982 年のプロジェクト開始時点でのプログラム言語の設計・実装技術は，核言語の設計を決めてしまうにはまだ未熟過ぎた．そこで，まず逐次システムを検討するところから研究を開始した．プロジェクトの前期 (予算年度で 1982–84) には，Prolog をベースとした逐次型核言語 ESP を設計し [Chikayama 1984]，前期と中期の当初の多くの研究開発の基盤とした．

次の版の核言語 KL1 の設計も，前期に並行して開始した．言語の試験的な設計と実装は中期の始めに行ない，中期 (1985–88) の間により本格的な実装まで完了した．この言語が後期 (1989–) にはさまざまな応用研究に用いられることとなった．本稿では以下，核言語とはこの第二世代の核言語 KL1 を意味するものとする．

2.3 論理型プログラミングの原則

論理型プログラミングの考え方は，プロジェクト全体の基礎となった．元々のプロジェクトの計画案における論理型プログラミングのイメージは，特定の言語 Prolog に強く影響されたようであるが，逐次システムから並列システムへの研究の発展にともない，並列論理型プログラミングのアプローチをとるように変わっていった．しかし「論理」を設計の中核的な原則とする方針は変わらなかった．

論理型プログラミングの原則は，設計にあたってのさまざまな候補の中から適切なものを選ぶ際に重要な役割を果たした．核言語の設計においては，数理論理学の意味での完全性はあきらめたが，健全性は設計の「規範」となった．[1] 核言語に対するさまざまな魅力的な機能拡張案は，検討を経た結果不健全であるという理由で捨てられた．一方，プログラムを論理式として解釈した際の意味を変えないような拡張は，より自由に導入していった．このような拡張は実行効率には影響してもプログラムの正しさには影響しないもので，言語の核となる部分からは明確に区別できる．

こうしたプログラムの論理としての解釈は言語の設計を首尾一貫したものとし，ひいてはその実装やプログラミングスタイルを一貫したものとするのに役立った．これについては以下に詳述する．

2.4 目的とするアーキテクチャ

現在，大型計算機に匹敵するような計算能力と適当な容量のメモリを持ったプロセサは，回路基盤一枚の上に実現できるようになっている．最近のハードウェア技術は，3 年ごとに 4 倍という回路の実装密度向上をもたらしてきている．これを外挿すると，次世紀の早い時期には 100 台程度のプロセサと適当な量のメモリがひとつのチップに納まることになる．一方，単一プロセサの性能も着実に進歩しているものの，同じ期間に 100 倍の向上を見ることは難しそうである．

実装密度の向上のために，より大規模な回路が実用的になってきているため，プロセサのコストの内で設計コストの占める割合がかなり高くなってきている．たとえ同じチップ面積を占めるシステムの性能が同じでも，マルチ・プロセサ・システムでは同じ設計を繰り返し使える点で，単一プロセサのシステムよりもコスト面で大きく有利になるだろう．このため，次世紀始めには，マルチ・プロセサ・システムは，絶対的な処理能力だけでなく，パームトップやリストウォッチ型などの小型計算機システムにおいても，コスト面で有利になっていくに違いない．

知識情報処理システムのような応用分野では，非均質な計算が必要になるため，計算機アーキテクチャも資源の柔軟な配置が可能なものが必要になる．また，大規模並列システムのためには，システムのアーキテクチャとしても高い拡張可能性が重要である．こうしたことから我々は，疎結合プロセサによる (あるいは密結合したいくつかのプロセサからなるクラスタが疎結合したような構造の) 均質的な MIMD アーキテクチャを目的とするアーキテクチャとして選んだ．

2.5 核言語のレベル

理想的なプログラム言語は，非常に高いレベルの記述が可能で，それを言語処理系が目的とするアーキテクチャに向けていっさい人手を借りずに最適化する，というような

[1] システムの健全性とは，導かれる結果がすべて与えられた公理の論理的帰結になっていることを意味する．一方，完全性とは，どのような論理的帰結も導ける，という意味である．

ものだろう．しかし，現在の技術ではそのような言語は夢でしかない．通信遅延を無視できない大規模疎結合並列計算機システムにおいては特にそうである．最適化でもっとも難しい部分は，処理の各部分をどこで (どのプロセサで) いつ (どんな順番で) 実行するかである．この問題はマッピングの問題と呼ばれている．

問題解決に用いる技法が比較的単純なら，必要な計算をあらかじめ予測することは難しくなく，コンパイラがマッピングを決めてしまうこともできる．しかし，知識処理のような高度な問題解決技法を必要とする領域では，次にすべき計算がそれまでの計算に依存することが多く，コンパイル時の最適化は不可能である．これまでの多くの研究からも，汎用的な自動マッピングは難しく，良好なマッピング・アルゴリズムはどのような問題解決技法を用いるかに大きく依存することがわかっている．

知識情報処理という領域では，ひとつでどんな問題も効率的に解けるような技法は知られていないので，ひとつのマッピング・アルゴリズムだけを用意するのは不適切である．知られているすべての技法をカバーするように，いくつもマッピング・アルゴリズムを用意しても足りない．この領域の研究はまだ初歩的段階にあり，新たな問題解決技法が近い将来次々に開発されていくであろうからである．このため，核言語のレベルは計算のマッピングをプログラム中に指定できるようなレベルとすることにした．

このように計算のマッピングをプログラマの責任にしてしまうと，プログラムがより難しい仕事になることは否めない．しかしこの困難は，高性能な知識情報処理システム技術の構築にあたって乗り越えていかざるを得ないものであると考える．多くの問題に適用できるマッピング・アルゴリズムが見つかったら，それはライブラリとして応用分野のユーザに提供することができる．プログラムの実行を制御できる核言語を用いれば，そういうライブラリを書くことは難しくないはずである．

2.6 新言語の設計

既存の論理型言語を基礎として，それに並列実行のための機能を付加し拡張する形で核言語を設計することもできたはずである．もっとも広く使われていた言語は当時 (現在もそうだが) Prolog だったので，そうした拡張のベースとしては Prolog は第一候補だった．

Prolog を並列システムのための言語に改造するにはふたつの方法があった．ひとつは，特に指定せずに自動的に計算のマッピングを決めてしまう方法だが，そういう方法をとらなかった理由は前述の通りである．もうひとつは，言語の機構を追加することによって並列計算を明示できるようにする方法である．しかし，元になる Prolog は逐次処理用に設計された言語なので，並列処理の指定は言語を複雑にし，プログラムの理解を困難にする．もっと大きな問題は，逐次実行を原則としてしまうと，より良いマッピングのためにプログラムを改訂するのが困難になる．マッピングを変えるには，プログラムの並列実行できる部分も変えなくてはならないからである．

もうひとつの問題は，同期を指示することの困難である．同期と条件判断を別々に行なうような言語では，未完成のデータに基づいて条件判断をしてしまう可能性がある．物理的に並列に動作するようなハードウェアでは，同じ現象を再現させるのが難しく，こうした同期の誤りを発見するのには多くの労苦を伴う．この問題を解決するには，同期と条件判断は別個のものにしてはならない．

結論として我々は，並列実行を自然に記述できるように，核言語をまったく新たに設計することとした．核言語は本質的に並列言語であるべきである．言語の機構は並列実行を原則とするものになっていて，逐次実行は必要なら明示すべきである．同期と条件判定はひとつの言語機構に一体化すべきである．

2.7 新オペレーティングシステムの設計

プロトタイプ並列推論システムは実験的なシステムではあるが，好適なソフトウェア開発環境を提供するオペレーティングシステムを用意することは必須である．必要な機能を提供する方法の一つとして，既存のオペレーティングシステムを並列推論マシンに移植することは考えられた．

当時利用可能だったオペレーティングシステムは (そして現在でも利用可能なもののほとんどは) もともと逐次システムのために設計したものに，後から並列システム上での実行に必要な機能を付け加えたものである．

このようなシステムには二つの大きな問題があった．ひとつは，オペレーティングシステムとユーザプログラムとのインタフェースが，相変わらず逐次実行を前提としている点である．たとえば，ユーザプログラムは要求したサービスの完了を，スーパバイザ呼び出しという手続きの完了によって通知されるようになっている．これは，ほとんどの応用ソフトウェアが逐次処理を基本とする言語で書かれている限りは問題ない．しかし，本質的な並列性を持った核言語で書いたソフトウェアとは相性が悪い．

もうひとつの問題は，オペレーティングシステムの管理方針が逐次処理向けに最適化されていることである．逐次システムや小規模の並列システムにおいては，管理情報を集中化するのがもっとも堅牢で効率的な手法である．しかし，これは大規模並列システムにおいては最適からは程遠い．もし大規模並列システムにおいて管理を単一のプロセサに集中させたら，そのプロセサの仕事が多くなり過ぎ，システム全体のボトルネックになるだろう．しかも，システム中のあらゆる活動のためにそのプロセサとの通信が必要になり，通信のボトルネックにもなる．

我々は，大規模並列システム向けに最適化したオペレーティングシステムを設計することも，高性能知識情報処理システムの技術の中核として必須であると結論し，オペレーティングシステムを全く新たに設計・実装すること

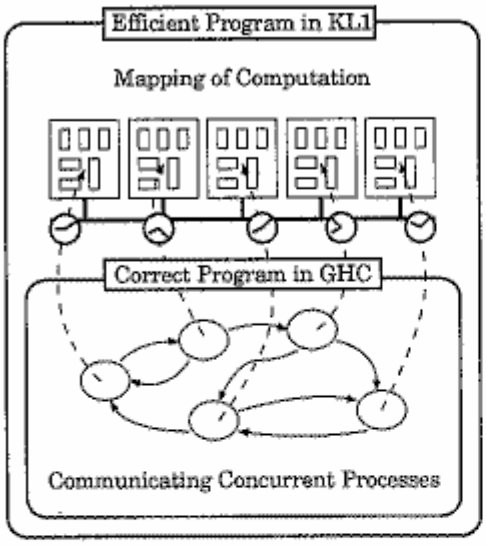

図 1: 核言語のふたつの階層

にした．核言語の設計と整合するように，逐次性をユーザインタフェース設計にとり入れないようにした．分散管理はボトルネックを避けるために不可欠である．分散管理の方針はオペレーティングシステムが提供するサービスの使用にも影響を与える．

3 核言語: KL1[2]

核言語 KL1 はふたつの階層からなる．基礎となる層は，並行論理型言語である Guarded Horn Clauses (GHC) で定義され，所期の結果を得るために何を計算しなければならないか，つまり正しいプログラムを記述するための機能を持つ．このレベルの記述では，計算のマッピングについて，所期の結果を得るのに必要な制約だけしか記述しない．この層の上に，KL1 言語全体ができあがっており，そこでは望ましいマッピングに基づき，計算をどのように行なうか，つまり効率的なプログラムを記述する．この正しさと効率性の分離，別の言葉でいえば，並行性と並列性の分離は，並列推論システムと知識情報処理との間のギャップを整合的に埋める役割を果たす．

3.1 並行論理型言語 GHC

本節では，核言語 KL1 の基礎となる，並行論理型言語 Guarded Horn Clauses について述べる．

3.1.1 並行論理型言語

核言語の設計は 1982 年のプロジェクト開始と同時に，まず適当な言語の枠組を探すところから始まった．並行論理型言語の枠組が適当な性質を備えていそうだったので，その言語族に属する数多くの言語を，核言語の基礎として検討対象とした．主なものとしては，Relational Language [Clark and Gregory 1981], Concurrent Prolog [Shapiro 1983] そして PARLOG [Clark and Gregory 1983] があった．この検討の結果，1984 年の終りに 新しい並行論理型言語 Guarded Horn Clauses (GHC) が誕生した [Ueda 1986].

GHC の基本的な枠組は他の並行論理型言語と同じである．第一に，GHC プログラムはガード付の節の集合である．次に GHC は don't-*know* 非決定性 (組込みの探索機能) を持たず，代わりに don't-*care* 非決定性を持つ．これによって，外界と関わりを持つようなリアクティブ・システムを記述できる．並行論理型言語でのリアクティブ・システムの記述は，論理のプロセス解釈 [van Emden and de Lucena Filho 1982] に基づく．ゴール (あるいはそこから導かれるサブゴールの集まり) はプロセスと見なすことができ，プロセスは共有変数とその値の結合を生成したり観測したりすることで互いに通信できる．多くの並行論理型言語と同じく，GHC のプロセス間で受け渡される結合は決定的である．つまり，いったん他のプロセスに結合を見せたら後で取り消すことはない．オペレーティングシステムのようなリアクティブ・システムでは，結合を通じて現実の外界と関わり合いを持つのだから，結合の決定性は不可欠である．言語に組み込まれた探索機能を持たないことにより，プログラム実行の詳細を指定できる．これはプログラムで計算のマッピングを指定するという方針と適合する．

3.2 GHC の特徴

他の並行論理型言語と比べて，GHC にはどのような利点があるのだろう． さまざまな並行論理型言語を検討する中で，我々はもっとも表現力の高い Concurrent Prolog をもっとも子細に検討し，プロトタイプ実装まで行なった [Miyazaki *et al.* 1985]．この経験から，言語におけるアトミックな操作の定義が明確になり，より簡単なアトミック操作を持つ言語を考えるようになったのである．

上述の通り並行論理型言語の重要な性質のひとつは，結合の決定性である．並行論理型プログラムはいずれも，インプット・リゾリューションを並列に行なうことによって複数のゴールを実行並列するのだが [Ueda 1988a], 以下の規則を守ることで結合の決定性を保証している．

1. ゴール g から呼ばれた複数の節のガード部 (ヘッドを含む) の実行は並行的だが，ゴール g を具体化することはない．
2. ゴール g はガードの実行が成功 (下記の 4 参照) した節のうちのひとつを選ぶ．
3. ゴール g が選んだ節のボディ部は g を具体化できる．選ばなかった節のボディやガードは g を具体化しな

[2] この節の内容は，3.4 節を除き，上田和紀との共著論文 [Ueda and Chikayama 1990] を改訂したものである．

い.

4. ゴールが成功するとは，ゴールがある節を選び，選んだ節のボディ中のゴールすべてが成功することをいう.

つまり，ゴールは節の選択の前には複数の節を試しても良いが，結合は行なわない．選択した後には結合を行なえるが，その時にはもう節はひとつに決まっているわけである.

並行論理型言語のもうひとつの重要な特徴は，同期の取り方にある．同期はユニフィケーションによる情報の流れを制限することによって行なう．Concurrent Prolog では read-only 指定を用いる．PARLOG ではモード宣言によって，入力引数のユニフィケーションを一方向ユニフィケーションとテストの命令列にコンパイルしてしまう．だが，こうした言語では上記の規則 (1) を守るために特別な仕組みを導入する必要がある.

GHC の考え方は単純である．GHC では規則 (1) そのものを同期機構に用いる．つまり，ガード中から直接・間接に行なわれるユニフィケーションで呼び出し側を具体化するようなものは，すべて呼び出し側を具体化せずに済むようになるまで待たされる．つまり，GHC では結合の決定性と同期という両概念を統合したわけである.

核言語は，応用，実装，理論などプロジェクトの多岐に渡る側面の研究開発にあたる人々に，共通の枠組を提供しなければならない．GHC を核言語の基礎として採用するには，以下の条件を満たしていることが必要だった.

1. 十分な表現力があること.

2. 効率的な実装が (必要なら適当にサブセット化するなどして) 可能であること.

3. プログラマが理解し利用するために，十分簡潔であること．同時に，理論的な扱いの面からも十分簡潔であること.

他の言語で記述した並行アルゴリズムは，ほとんどどれでも GHC でうまく書けることはすぐにわかった．が，それだけでは不十分だった．我々のプロジェクトでは当初普通の論理型言語を用いていたので，その得意とする探索問題のプログラムをどう書けるかも重要だった．そこで，探索問題のさまざまな解法を開発した [Ueda 1987, Tamaki 1987, Okumura and Matsumoto 1987].

実装については，プロトタイプを作ってみた結果，GHC は少なくとも逐次計算機上には比較的簡単に実装できることがわかった [Ueda and Chikayama 1985].

3.2.1 Flat GHC

簡潔さの面では，プロセス解釈に向いた言語機能の簡潔な説明ができるように GHC の性質の研究を続けてきた．現在のところの我々の解釈によると，GHC のプロセスは結合という形で表される情報を観測・生成する，複数のボディゴールからなる抽象的存在である．個々のボディゴールの振舞いはガード付の節群で規定され，それは書き換え規則と見なせる.

元々の GHC には，ガードゴールがこのプロセス解釈にそぐわないという問題点があった．実際上も，たとえガードゴールをうまく実装できたとしても，それによる表現力強化のメリットより，実装の手間の問題の方が大きそうだった.

こうした考察から，GHC のサブセット Flat GHC を考えるようになった．Flat GHC のガードゴールは，節を選択する補助的な条件である．補助的な条件としてのガードゴールの満たすべき性質は，決定的 (成功するかどうかが引数の値だけに依存) で，結合を行なわないことであるという結論に達した．この制限によって，GHC の操作的意味論 [Ueda 1990] やプログラム変換規則 [Ueda and Furukawa 1988] などの理論的な扱いが容易になった.

Flat GHC の基本的な考え方をまとめると，次のようになる．プログラムはゴールの書き換え規則とみなせるようなガード付の節の集合である．節のガードは書き換え規則適用の前にどのような情報を観測すべきかを，ボディは元のゴールと置き換えるゴールのマルチ集合を与える．ボディのゴールは，言語で振舞いを定義する $t_1 = t_2$ のようなユニフィケーション・ゴール，または，ユーザが振舞いを定義するそれ以外のゴールである．ユニフィケーション・ボディ・ゴールは t_1 と t_2 をユニファイすることによって情報を生成する．それ以外のゴールは，いずれ行なうべき残りの仕事を表す.

3.2.2 GHC の性質

Flat GHC の意味論は，計算的・論理的の両面から理解できる．計算的な意味論の一例は上述のプロセス解釈である．Maher に通信と動機の論理としての意味づけを与えた [Maher 1987]．これによれば，プロセス間で授受する情報は項の間の等式制約とみなせる.

Concurrent Prolog とは異なり，PARLOG と同様，結合はユニフィケーション以外のゴールの節選択時にアトミックに発行されるわけではなく，選択の後にユニフィケーション・ボディ・ゴールによって順時発行され，他のゴールと並列に行なってもよい．だから，GHC の節選択は Concurrent Prolog よりも小さな操作である．さらに GHC では，ユニフィケーション・ボディ・ゴールの情報生成もアトミックではない．情報は少しずつ，通信遅延があるようなやりかたで送られても良い．この (Flat) GHC のような鷹揚な計算モデルでも，協調動作する並行プロセスは十分表現でき，実装の自由度を大きく残せることがわかった.

もうひとつ重要なことは，GHC はプロセスの正しい振舞いのための制御はするが，効率的な実行のための制御はしないということである．両概念を明確に区別するため，効率性については KL1 に譲ったのである．ボディ・ゴー

ルの逐次 AND 機能によって並列実行を抑制できる PARLOG とは，ここが異なる．情報の流れに基づく同期だけで，正しい並行プログラムが記述できることを示せたのは重要だと考えている．

Flat GHC に関する理論面からの重要な課題として，CCS [Milner 1989] や理論的 CSP [Hoare 1985] などの並行実行の理論モデルとの関係づけがある．並行論理型言語は CCS や CSP とは異なり，非同期通信を基本とし動的に再構成できるプロセス群を記述できるのだが，良く似た数学的手法で形式化できるのである．まだ完全に満足できる形式的意味記述には達していないのだが，Flat GHC は理論面からも十分簡潔であることは間違いようだ．しかも，そのままで実際のプログラミングにも使えるものになっているのである．

3.3 実用的並列言語 KL1

上述の通り，我々は並列推論システムの核言語の基礎として並行論理型言語 Flat GHC を設計した．しかし，並列計算機を考えるそもそもの動機である効率的実行を問題とする場合，この言語の表現能力では不足である．

Flat GHC プログラムは計算を構成するアトミックな操作をどこで (どのプロセサで) 行なうかについては何も指示しないので，利用できるプロセサ上に計算を分散する方法はいろいろある．Flat GHC ではアトミックな操作の実行順を半順序としてしか規定しないので，それを満たす全順序もいろいろある．実際の分散や順序が，最適からかけ離れたものにならないことを保証するには，ある程度は実行の物理的な詳細を指定できなくてはならない．

そこで，並行プログラム言語 Flat GHC に基づいて並列プログラム言語を設計し，プログラムをどのように実行するかをある程度詳細に指定できるようにした．本節ではこの言語 KL1 を概説する．

3.3.1 計算のマッピング

Flat GHC のプログラムは，正当性の保証に必要なアトミックな操作の順序だけしか指定しないので，並列処理の可能性は暗黙に表現されるだけである．限られた数のプロセサしかなく，通信コストも無視できない実際のハードウェアに上では，この可能性を忠実に引き出そうとするだけでは最適な効率は得られない．効率のためには，アトミックな操作をいつどこで行なうかの制御が必要である．この制御をマッピングと呼ぶ．

マッピングは逐次システムでは暗黙に指定することが多い．ある問題を解くのにふたつの方法があったとしよう．逐次システムでは，まず効率は良いが信頼度の低い方法を試し，それがうまくいかない場合だけ効率で劣るが信頼度の高い方法を試す，というのが良い方法だろう．これは並列システムにおいては最良とは限らない．第一の方法がプロセサのような計算資源のすべてを必要とはしないのなら，第二の方法もそれと並列に行なうべきである．この計算は第一の方法の結果如何によって，必要かも知れないし不要かも知れない．こういう計算を見込み計算という [Burton 1985]．効率を考えると，第二の方法による計算は第一の方法の実行に必要な資源を横取りしてはならない．これには第一の方法を第二の方法より優先させればよい．この意味で，元の逐次的なやり方で二つの方法を逐次的に実行したのは，正しい結果を得るためではなく，効率のための優先度を暗黙に指定するためだったのである．

もっと高度なマッピングが必要な場合もある．たとえばある問題にふたつの解法があり，少なくともひとつは答を効率的に見つけられることがわかっているが，どちらがそうかは前もってはわからないとしよう．この場合は，ふたつの方法に同程度の計算資源を割くべきである．このように，資源管理は並列計算アルゴリズムの重要な一翼を担うのである．

逐次計算機システム，そして従来の逐次システムの拡張としての並列計算機システムでは，マッピングは主にオペレーティング・システムの仕事だった．応用プログラムがだいたいにおいて逐次的で，マッピング戦略を逐次実行によって暗黙的に指定している限りは，この方法でも問題ない．だが，マッピング操作を頻繁に指定する並列システムにおいては，マッピング操作のたびにオペレーティング・システムを呼出すオーバヘッドは耐え難いものになる．

3.3.2 KL1 のマッピング機能

この問題を解決するために，KL1 には以下のような機能を採り入れ，効率的な実装を目指した．

荘園 荘園はゴールのグループである．このグループは実行の開始，中断，再開，放棄といった実行制御の単位となる．例外処理や資源消費管理機構にもこの荘園構造を用いる．荘園は，荘園内での実行を制御するメッセージを送るために荘園の外部から内部に向かう制御ストリームと，荘園内での出来事を報告する逆向きの報告ストリームという二本の通信路を持つ．荘園構造は Clark と Gregory の提唱したメタコール機構 [Clark and Gregory 1984] を拡張したものになっている．

優先度 KL1 での優先度制御の単位は，ボディ・ゴールである．各ゴールは整数値の優先度を持つ．各荘園は属するゴールに許される最高・最低の優先度を持っており，各ゴールの優先度はそれとの相対値で指定する．言語は多くの優先度レベルを提供し，それを実装ごとに物理的な優先度に翻訳する．

プロセサ指定 各ボディ・ゴールには，実行するプロセサ (あるいは，プロセサのグループ) を指定するプロセサ指定を付けられる．

この単純な機構は，より進んだマッピング戦略の研究の基礎になる．実際，種々のプログラムに対する自動負荷分散戦略が開発されてきており，比較的応用範囲の広い

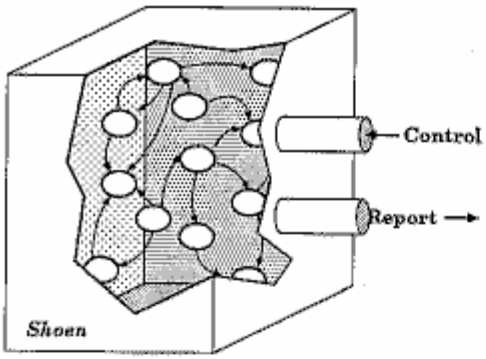

図 2: 荘園構造

手法はライブラリとして提供している [Furuichi *et al.* 1990].

KL1 言語の大きな特徴は，こうした優先度やプロセサの指定を並行実行の制御と分離したことにある．こうした指定はプラグマと呼ばれる．プラグマは言語処理系に対するガイドラインに過ぎず，処理系は必ずしもこれに従うとは限らない．同じことは荘園による制御機構についてもいえる．たとえば実行の放棄を指示しても，すぐに放棄するとは限らない．これでも良いとすると分散処理系の実装がずっと簡単になる．

多くの並列プログラム言語では，並列実行の指定は他の言語機構，ことに並行実行制御の機構と一体化している．これでは，効率をあげるために計算のマッピングの仕方を変えるだけでもプログラムを大幅に改定する必要が生じやすく，バグを導入する結果になりやすい．

プラグマは KL1 プログラム中に指定するが，他の言御構造とはシンタクス上明確に区別されている．プラグマはプログラムの性能を大幅に変えることはあっても，正当性を変えることはない[3]．プログラム開発労力の半分以上が適切なマッピングの設計にあることも少なくないので，正当性に影響を与えずにマッピング指定を変更できるのは大きな利点である．

3.3.3 逐次言語に伍して

並列アルゴリズムを比較する基準としては何が適当だろう．ある並列アルゴリズムが逐次実行時間 $c(n)$ (n は問題の大きさ) と平均並列動作可能性 $p(n)$ を持っていたとする．理想的な並列計算機上では，このアルゴリズムによる総実行時間は $c(n)/p(n)$ になる．つまり，理想並列計算機上では，逐次実行時間が大きくても並列度がもっと大きければ，良いアルゴリズムだということになる．

しかし，物理的に利用可能な並列度より，(実行中に変化する) アルゴリズムの並列動作可能性の方が大きくなることがあるとしたら，そうはいかない．実際には物理的な並列度は限られているので，逐次時間で既存の逐次アルゴリズムよりも悪いような並列アルゴリズムは，どんな大きな $p(n)$ を持っていようと，問題の大きさ n が大きくなると逐次計算機上の逐次アルゴリズムに負けてしまうのである．つまり，本当に役に立つ並列言語は，どんなアルゴリズムでも逐次言語と同じ逐次実行時間をような記述ができなくてはならないのである．

Pure Lisp や pure Prolog のようなピュアな言語では，破壊的代入という概念がないため，ある種の効率的アルゴリズムをそのまま表現することができない．これを克服するためには，処理系でハードウェアのメモリの持つ破壊的代入機能を生かせるような最適化手法が必要である．GHC もピュアな言語なので，本来同じ問題がある．こうしたピュアな言語で効率的アルゴリズムを記述するには，従来の言語の配列の持つ効率性を真似できなくてはならない．

このために，KL1 では論理変数の単一代入性を崩すことなく，一定の手間で配列要素の更新ができるようなプリミティブを導入した．そのプリミティブは以下のような形式で使える．

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

配列 Vect，インデクス値 Index，新しい要素の値 NewElem を指定すると，この述語は Elem を元の Index 番目の要素の値と結合し，NewVect に元の Vect とほとんど同じだが Index 番目の要素だけを NewElem に置き換えたような配列を返す．

他のゴールが元の配列 Vect を参照しているかも知れないので，単純に実装すると，NewVect のために新しい配列を割り付け，ひとつを除くすべての要素をコピーすることになろう．だが，この set_vector_element 以外のゴールは Vect を参照しないとわかれば，破壊的に更新しても問題ない．実際の KL1 処理系では，簡素化した効率的な参照カウント方式 [Chikayama and Kimura 1987] を用いてこのような場合を検出し，NewVect を一定の手間で作ることができる．

単一参照の配列を使えば，ランダム・アクセス・メモリをシ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，計算の複雑さを変えずに KL1 に書き換えられる．もちろん，データ構造に単一参照しか許さないのでは，並列実行の可能性をかなり低減することになる．しかし，計算の複雑さを保つ必要があるのは，物理的に使える並列性を使い切った後なのである．

3.4 さらに高いレベルの言語

核言語 KL1 では手続き言語よりも高いレベルのプログラム記述ができるが，その記述レベルは Lisp と同程度な

[3]正確には，優先度指定を使うと，ひとりでには止まらないような枝を持つプログラムを，いつかは必ず停止させるようにするためにも使える．

ので，知識情報処理領域の応用プログラム記述には低過ぎる場合もある．この節では KL1 の上に，さらに高いレベルの言語機構を提供する研究について述べる．

3.4.1 マクロ展開

ESP で提供しているもの [Kondoh and Chikayama 1988] と良く似た，強力なマクロ展開機構を設計・実装した．このマクロは単にマクロ呼び出しをその場に展開するだけではなく，プログラム中に引数，ゴール，節の各レベルに項を挿入するような機能を持っている．この機能を用いて以下のようなことができる．

- 単純な置き換え
- 条件コンパイル
- 関数的記法 (算術式を含む)
- 暗黙の引数

Flat GHC プログラムのゴールは，一回のリダクションでサブゴールに展開されるまでという，ごく短い寿命のものである．長い寿命を持つプロセスを実現するためには，同じ述語をほとんど同じ引数で再帰的に呼ぶようなプログラミングスタイルをとる．このプログラミングスタイルはオペレーティングシステムでも応用プログラムでも，いたるところで使われている．このスタイルをとると，プロセスの状態や他のプロセスとの通信経路 (共有変数) は，再帰呼び出しの引数に渡さなくてはならない．これによりモジュール性は高くなるが，いつもそうした引数を書かなくてはならないのは繁雑で，プログラムを理解するのにも改訂する際にも邪魔になる．暗黙の引数の機構はこうしたプロセスをより簡素に記述するのに便利である．

KL1 のマクロ展開機構は強力なので，それを用いて実現できる機能は単なる構文上の便宜という程度を越えたものになっている．しかし，プログラマは KL1 のどんなプログラミングスタイルをとろうと自由である．これは有利な場合もあるが，プログラムの理解や保守のためには，利用する言語機能を制限することが有益であることも多い．そこで，KL1 にコンパイルするような，より高いレベルの言語を設計することにした．それらについては次節以降に述べる．

3.4.2 A'UM

もっとも良く使われる KL1 のプログラミングスタイルは，メッセージストリームを介して通信し合うプロセスとして記述する方法である [Shapiro and Takeuchi 1983]．ストリームは漸時具体化されていくような二進セルで表現されたリスト構造で実現する．プロセスは再帰呼び出しで実現する．A'UM は，ストリームやプロセスの実現機構を書き下すのではなく，より直接的にこのようなプログラムを記述できるように設計したプログラム言語である．

A'UM のプロトタイプは，KL1 への翻訳系として実装した．オブジェクト指向を徹底させた結果，A'UM では言語のいかなる要素も，たとえば整数値までもがプロセスのように振舞う．実装にあたっては，こうしたものまでも KL1 では実際にプロセスとして実装するしかなかった．そこで，オブジェクト指向の徹底を捨てるか，実装方式を変えて並列推論システムとは別に考えるかの選択を迫られたのである．A'UM では後者の道をとることにし，より直接的な実装の研究が進行中である [Konishi *et al.* 1992]．プロトタイプ処理系は，ネットワーク結合したワークステーション上ですでに動作している．前者の道は，同様な目的を持ったもうひとつの言語 AYA で追求することとなった．これについては次節に述べる．

3.4.3 AYA

言語 AYA の設計は，A'UM について並列推論システム上での実際的な効率より，純粋なオブジェクト指向の追求を優先することを決めてから始まった [Susaki and Chikayama 1991]．

AYA の設計目的は A'UM の当初の設計動機と同じで，KL1 でオブジェクト指向のプログラミングスタイルをとるプログラムを，簡潔に記述できるようにすることにある．AYA の設計に当たっては，オブジェクト指向言語としての統一性よりも，実際的な効率や記述の自由度に重きをおいた．すべてがオブジェクトというわけではない．たとえば整数値は "add" というメッセージには答えない．設計は主にボトムアップに行なった．言語の機能の多くは KL1 でのプログラム経験に基づいて決められた．

AYA のプロセスは，メッセージを受けとるストリームを複数持つことができる．このため，メッセージストリームひとつがオブジェクトひとつに対応していると考えることはできない．ストリーム以外の通信パターン，たとえば非同期的な割り込みなども記述できる．

AYA の特徴的のひとつに，プロセスのマクロなコンテキストに対応するシーンという概念がある．プロセスは数多くのシーンを持つことがあり，外部からのメッセージにどう反応するかは，現在どのシーンにいるかによって異なる．

AYA も実装作業中で，KL1 へのトランスレータのプロトタイプはすでに動作している．

4 オペレーティングシステム: PIMOS

上述の通り，大規模並列計算機システムの性能をフルに引き出すためには，大規模並列プログラムの制御に最適化したオペレーティングシステムは，不可欠である．また，並列ソフトウェアの研究に実際的に多用されるシステムは，ユーザフレンドリで堅牢でなくてはならない．並列推論マシンのオペレーティングシステム PIMOS はこの要求に答えるべく設計し，核言語で実装した．この節では PIMOS の設計を概観する．

4.1 先行する研究

Shapiro は [Shapiro 1986] で，オペレーティングシステム全体を並行論理型言語で記述することの可能性と利点を指摘した．この指針に基づき，いろいろな面での改良を重ねて，いくつかの実験的なシステムが作られた．Logix system [Hircsh *et al.* 1987] や the Parlog Programming System (PPS) [Foster 1987] などがその例である．

PIMOS は PPS との類似点が多い．これは，ひとつには実装言語 (KL1 と PARLOG) の類似性に原因があり，また両グループが頻繁にアイデアを交換しあってきたためでもある．

PIMOS と上記の他のシステムとの大きな違いは，元となった言語処理系とシステムの使われ方にある．PIMOS は並列ハードウェアの上で効率的に実行でき，応用ソフトウェアの研究開発に実際に使っていくように設計したが，他のシステムは既存のシステム上の実験的なシステムである．言い替えれば，PIMOS は並行論理型言語でオペレーティングシステムを構築するという新しい方式を確立するという目的においては他のシステムと同じだが，さらに応用ソフトウェア開発のための快適で効率的な環境を提供するという目的も持っている．これは設計上のトレードオフにさまざまな影響を与えた．

4.2 目標

PIMOS の設計は，以下の目標を持って行なった．

堅牢性: PIMOS はスタンドアローンの並列計算機システム上で動作するものなので，すでに確立したシステムの上に構築する場合よりも高い堅牢性が必要である．

内部の並列性: 前述の通り PIMOS の究極の目的は並列推論ハードウェアの持つ能力をフルに引き出すことにある．そうしたオペレーティングシステム自体の中で必要な計算も，並列に行なうべきである．そうしないと，オペレーティングシステムがシステム全体のボトルネックになってしまう．

高い局所性: 目的とするアーキテクチャは疎結合計算機なので，プロセサ間の通信のコストは，プロセサひとつの中での通信コストよりはるかに大きい．したがって，プロセサ間の通信量はできるだけ抑制しなければならない．

柔軟性: ハードウェアの持つパラメタは変化していくと考えられ，システムはそのパラメタに合わせてチューンしやすい柔軟性を持つ必要がある．オペレーティングシステムのパラメタをチューニングする程度では不十分な場合，かなり本質的な再設計が必要になることも考えられる．したがって，改良が簡単にできるシステムにすることが望ましい．

4.3 資源管理

資源の管理はオペレーティングシステムのもっとも基本的で重要な役割である．本節では PIMOS の資源管理機構の設計について述べる．[4]

4.3.1 どんな資源を管理するか

従来のシステムでは，メモリ管理とプロセス管理はオペレーティングシステムの主要な仕事だった．他の記号処理言語と同様，KL1 はゴミ集めを含む自動メモリ管理機構を持っている．このため，基本的なメモリ管理は PIMOS ではなく，言語処理系の役目になっている．KL1 は暗黙の並列性とデータフロー同期機構を持っているので，コンテキスト切替やスケジューリング機能は言語が提供している．そこで，PIMOS は低レベルの細粒度プロセスの管理を行なう必要はなく，プロセスのグループという粒度の大きいものを核言語の荘園機能を用いて制御する．

一方，入出力装置のような資源については，PIMOS が全面的に責任を負っている．最低レベルでは，入出力装置は物理的な機器を制御する核言語のプリミティブとして提供されている．核言語はリアクティブシステムを記述することができるので，こうした機器は核言語レベルでは普通のプロセスのように見える．しかし，その提供する機能は，応用プログラムにとってはレベルが低過ぎる．他のオペレーティングシステムと同様，PIMOS はこうした機器を仮想化し，応用プログラムからは高いレベルの機能を持つ仮想機器を制御できるようにしている．

こうした仮想機器は，実際にはユーザタスクからの高いレベルの要求を，物理的な機器が理解できる低いレベルの要求に変換するようなプロセスである．ユーザタスクはそうしたプロセスにつながったストリームに要求メッセージを流す．したがって，機器の管理はそれにつながった通信ストリームの管理になる．保護機構は，そうしたストリームに流れるメッセージを監視して，機器への不当な要求をはねつけるようなフィルタプロセスを挿入することによって行なう．

上述の通り，PIMOS でのプロセス管理は荘園機構を用いておこなう．PIMOS は荘園をタスクとして，より高いレベルの資源管理機構を持つように仮想化する．ユーザプログラムからのタスクの制御は，そこにつながっているストリームからしか行なえない．メッセージをフィルタするプロセスを挿入する保護機構を，ここでも用いている．

4.3.2 資源の階層的管理

従来の多くのオペレーティングシステムでは，肝要な管理情報は通常単一のプロセスとして実装するカーネルに集中化してしまう．この集中化方式は管理情報の整合性を保つのに有利である．

だが，大規模並列システムにおいては，このような管理

[4] より詳細な記述は [Yashiro *et al.* 1992] にある．

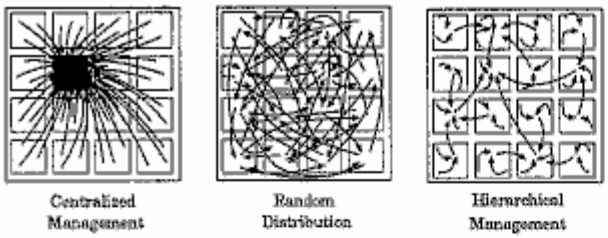

図 3: 管理の分散

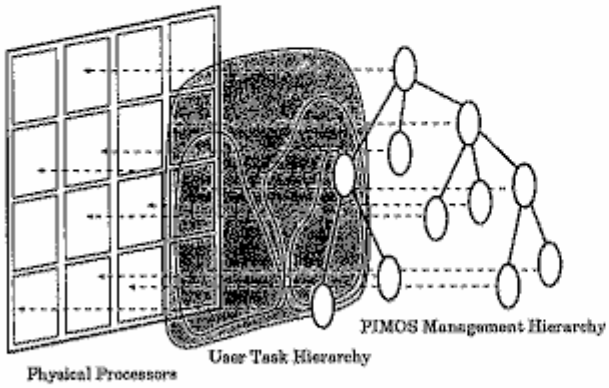

図 4: タスク管理の階層構造

情報の集中には問題がある．カーネルのオーバヘッドがたった 1% だったとしても，百台のプロセサを持つシステムではカーネルの実行速度がシステムのボトルネックになってしまう．しかも，管理に関する要求すべてがカーネルプロセスの動くプロセサに集中するため，通信メカニズムのホットスポットが生じてしまう．大規模並列計算機システムのオペレーティングシステムでは，管理の仕事も分散しなくてはならないのである．

ハッシングなどの手法で管理をランダムに分散させればボトルネックは解消するが，オペレーティングシステムのサービスの要求はどこでサービスを提供しているかと無関係に行なわれるので，通信が多くなるという問題が生じる．

ボトルネックと頻繁な通信を同時に避けるためには，情報の局所性を保つように管理を分散させなくてはならない．PIMOS はこのために階層的な資源管理方針をとった．ユーザタスクとオペレーティングシステムが用意する資源は，階層的な構造をなす．計算のマッピングを応用プログラムに任せる設計方針に従い，資源管理のための PIMOS のプロセスはサービス要求のあったところに作り，ユーザタスクの構造に対応するような階層的構造をなす．これを資源木と呼ぶ．この資源木が PIMOS のカーネルである．

資源管理情報は一切集中化しないし，資源の割り付けに全順序をつけることもしない．資源木のノードである管理プロセスは，自分の親と子供を知っているだけである．新たな資源の割り付けは階層のひとつのレベルで局所的に行ない，上のレベルにも下のレベルにも報告しない．必要なら，管理情報の統計だけは資源木の中で交換するが，システム全体の状態を正確にとらえているプロセスはどこにもない．システム全体の状態は木構造をたどればわかるが，そのコストが高いし，システム中の並行動作する活動のため，情報を集め終った時にはすでにその情報は古いものになっている．このルースな管理方針でもまったく問題なく運営できることがわかった．

4.3.3 サーバ

PIMOS はすべてのサービスを仮想機器に対応するサーバを用いて提供する．サーバは普通のタスクとして実現するので，カーネルは小さくなり，新たなサービスの付加は容易である．

サービス（たとえばディスプレイ上にウインドウを開くこと）を必要とする応用プログラム（クライアント）は，サービスの名前を用いてカーネルに要求する．カーネルは表の中にその名前のサービスを見つけ，サーバのタスクとクライアントのタスクをストリームで接続する．この際，クライアントタスクの中に保護のためのフィルタプロセスを挿入する．いったん接続ができたら，カーネルはストリームを流れるメッセージを見たりしない．サーバを保護するのはカーネルではなく，挿入したフィルタの仕事である．サービスの必要がなくなったら，通常クライアントプロセスは通信ストリームを閉じる．残るカーネルの仕事は，クライアントが異常終了した際に，それをサーバに知らせることである．

4.4 ファイルシステム

実験機 Multi-PSI [Takeda *et al.* 1990] 上の PIMOS の初期の版では，すべての外部入出力は入出力フロントエンドプロセサ PSI [Nakashima 1987] に頼っていた．この方針は応用研究のためのソフトウェア環境を早期に形成するのには有益だった．しかし，ディスクのような大容量外部記憶装置について，入出力フロントエンドとの低い通信スループットと，並列ハードウェアの高い処理能力のアンバランスが，PIM [Taki 1992] では問題になってきた．

そこで，PIM のプロセサにはもっと直接的にディスクを接続し，より高いスループットと短いディレイを得られるようにした．ハードウェア開発を容易にするために，SCSI (small computer standard interface) を用いて市販のディスクとインタフェースできるようにした．SCSI ひとつのスループットは低いが，PIM は SCSI を数多く持つことができ，総合的には十分なスループットを得ることができる．

このインタフェースが提供するのは低レベルのディスクへのブロック入出力だけなので，応用プログラムに高いレベルのインタフェースを提供するためにファイルシス

テムを設計することにした．ファイルシステムの設計にあたっては，以下の方針をとった．

分散キャッシュ: プロセサ間通信の頻度を減らすために，各プロセサはそれぞれ，ファイル中のデータのキャッシュを持つ．キャッシュ機構には“UNIX セマンティクス”を持たせた．つまり，あるプロセスがファイルに書き込むと，そのデータは即座に他のプロセスから使えるようにした．これは遅れが許されるような分散ファイルシステム [Levy and Silverschatz 1989] よりきつい条件であるが，PIMOS のように多くのプロセスが協力してひとつの問題を解くのが普通であるシステムには必須である．分散的に整合性を保つキャッシュ機構を設計した．これはスヌービーキャッシュ [Archibald and Bare 1986] と似たものだが，通信遅れを許す点で異なっている．

堅牢性: ハードウェア，オペレーティングシステム，そしてファイルシステム自身というシステムの部品すべては実験的なものであり，バグによって損傷する可能性がある．このため，快適なソフトウェア開発環境を提供するためには，十分なバックアップ機構が必要になる．このため，ファイルシステムにとって肝要な情報をロギングし，ログ情報に基づいて迅速に回復する機構を設計した．

より詳細な記述は [Itoh *et al.* 1992] を参照されたい．

4.5 ソフトウェア開発ツール

並列ソフトウェアの開発は逐次ソフトウェアの開発とは異なる側面を数多く持っている．PIMOS は並列ソフトウェアの開発をサポートするさまざまなツールを提供している．本節ではそれらについて述べる．

4.5.1 プログラムコードの管理

核言語では実行可能プログラムはモジュール型のデータオブジェクトとして用意されており，特権を持つソフトウェアからは言語のプリミティブを介して操作できる．実行プログラムの表現形式はハードウェアモデルごとに異なるが，PIMOS が違いを隠蔽するプログラム操作の共通インタフェースを提供している．

実行可能プログラムはデータベースに格納されており，それはサーバタスクとして実現する仮想デバイスになっている． 仕様の論理的健全性を保つためには，修正という概念を導入するのは，普通のデータだけでなくデータとして扱われるプログラムについても望ましくない．プログラムモジュールの更新は既存の，システム中のどこかで並列に動作中かも知れないプログラムの修正を意味するのではない． 単にプログラムデータベース中に保持するモジュール名と実行可能プログラムの対応関係を更新するだけである．そのプログラムを実行中の既存プロセスはこの更新の影響を受けない．ただし，更新したモジュールが名前により参照され，データベースを探しにいった場合には新しい版が見つかる．このように同じプログラムの複数の版がシステム中に混在するのである．これは意味論を明確にするだけでなく，分散実装を容易にする．

4.5.2 デバッグ用トレーサ

プログラムデバッグにもっとも良く使うツールは，プログラマがプログラム実行の詳細を調べられるようにするトレーサである．PIMOS もこのデバッグ目的のプログラムトレーサを提供している．

高レベル言語でのプログラム実行は，入れ子になったサブルーチン呼び出しのような階層構造をなす． 逐次言語のサブルーチンの場合，サブルーチンに対応する部分構造は，「あるサブルーチンの実行中」といった時間間隔に直接対応する．特定の部分構造をトレースするか否かは，その間トレースをオンにしたりオフにしたりすることで実現できる．並行言語では複数の部分構造の実行が並行的に行なわれるので，このような直接の対応はない．プロセスの数が限られたものなら，各プロセスごとにウインドウを用意し，ウインドウごとにトレースをスイッチするのも良いだろう．しかし，KL1 プログラムの場合，プロセス数が数百万になるのも普通で，このような方法ではうまく行かない．PIMOS が提供するトレーサでも複数のウインドウを使ってトレース情報を提示できるが，それは副次的手段に過ぎない．

トレースの制御，トレース情報の獲得，トレース対象プログラムの制御には，核言語の荘園構造を用いる．荘園中のゴールにはトレース対象のマークをつけることができる．言語処理系がそのようなゴールからサブゴールへのリダクションを見つけると，新たに作ったサブゴールを荘園の報告ストリームからメッセージとして報告する．このストリームを監視しているトレーサは，情報をユーザに提示し，このゴールをどうするか，つまり，単に実行するか，またトレースマークをつけて実行するかを尋ねる．実行順序を制御するために，ゴールの実行をしばらく差し止めることもできる．

トレーサは KL1 処理系が提供するデッドロック検出機構とのインタフェースも持っている．

4.5.3 性能チューニング

前述の通り，核言語 KL1 の長所は，プロセサ・時間の両軸での計算のマッピングを，プログラムの正しさを損なわずに変更できる点にある．効率的な計算を実現するマッピングを探すことは，並列推論システム上の応用ソフトウェア研究の重要課題である．

プログラムを解析してマッピングがどうなるかを推定するのは容易ではない． 多くの場合 ，実際にプログラムを走らせ，統計情報を集めてみると，見過ごしていたプログラムのいろいろな側面が明らかになるものである．この

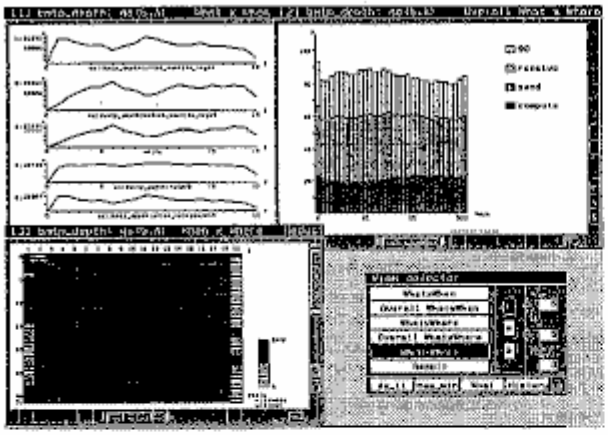

図 5: ParaGraph の出力例

ような実験を助けるために, PIMOS では負荷分散アルゴリズムの評価を助けるツールを提供している.

並列プログラムのプロファイル情報は, 三つの軸, いつ, どこで, 何を, を持っている. 逐次実行では, 「どこで」は一定していて, 実行順は完全に指定されているから「いつ」も重要ではない. このため「何」(プログラムのどの部分) にどのぐらいの時間がかかったかだけがわかるような, 単純なプロファイリングツールでも十分である. しかし, 並列実行の場合, 三つの軸いずれもが重要である. 核言語処理系は「何を」(プログラムのどの部分を, 低レベルでは, 普通の計算, プロセサ間通信, ガーベジコレクションの内どれを), 「どこで」(どのプロセサで) 「いつ」実行したかという統計情報を提供する機能を持つ.

何百ものプロセサから集まる大量の生データを理解するのは, 人間には容易ではないので, データを解析しグラフィックにユーザに提示するプロファイリングツール ParaGraph を用意した (Figure 5). このシステムは, 解析を容易にするために, いくつもの違った視点からの表示を提供している. ParaGraph については, [Aikawa 1992 *et al.*] により詳細な記述がある.

4.5.4 仮想マシン

ユーザプログラムと PIMOS との間の通信は, すべて荘園の制御・報告ストリームを通じて初期化するので, プログラムを荘園内で走らせ, そのインタフェースストリームを監視すれば, ユーザプログラムで PIMOS をエミュレートすることができる.

同じ方法を使って, 並列計算機システム全体, つまり仮想マシンのエミュレータを書けば, PIMOS を自分自身の下でデバッグすることができる. 仮想マシンは PIMOS のひとつのタスクに過ぎないのだから, PIMOS の保護機構がデバッグ中の版のバグが本当の PIMOS に浸透するのを防いでくれる. また, プロファイリングシステム ParaGraph を PIMOS の性能チューニングに使うこともできる. この仮想マシン機構は PIMOS のカーネルのデバッグとチューニングに大いに役立った.

5 経験

PIMOS の最初の版は 1988 年に Multi-PSI [Takeda *et al.* 1990] 上に実装された [Chikayama *et al.* 1988]. その後, 多くの応用分野での実験的ソフトウェアの研究開発経験を通じ, さまざまな拡張や改善を施してきた. 応用ソフトウェアでの利用経験は他に報告されている (たとえば [Nitta *et al.* 1991] を参照されたい) ので, ここでは主として核言語 KL1 を用いた PIMOS 自身の開発経験について報告する.

5.1 自動同期機構

KL1 の自動同期機構により, PIMOS をアーキテクチャの異なるハードウェアシステムに移植することが容易になった.

PIMOS の最初の版の開発は, 並列推論マシンの実験機 Multi-PSI の開発と並行して行なわれた. このため, 初期の開発段階では実際に核言語を走らせられる並列システムはなく, 開発は逐次的な処理系上で行った. この処理系ではゴールのスケジューリングは固定されている. この時点ではエミュレータの固定スケジューリングに隠された重篤な同期の問題がある可能性を完全には否定し切れなかった. なにしろ, 大規模ソフトウェアを KL1 で書いた最初の経験だったのである.

KL1 処理系の準備が整うと, PIMOS を Multi-PSI に移植した. 実際の並列マシン上のスケジューリングはエミュレータとは全然異なるにも関わらず, 同期の問題は (どく高いレベルの設計上の問題いくつかを除き) ほとんどまったく出なかった. こうなるはずだとは思っていたものの, 実際に経験してみて, 自動的なデータフロー同期を行なう言語でシステムを記述するメリットに自信を深めた.

1991 年には, 並列推論マシンの最初のモデル PIM/m と, その上の KL1 処理系に, ソフトウェアを実装する準備が整った. この新たなプラットフォーム上に移植するための低レベル入出力機構の改訂を行なった後, PIMOS はほとんど即座にこのシステム上で何の問題もなく動作し始めた. このシステム上の KL1 処理系のスケジューリング方針は Multi-PSI と同じだったので, これは驚くには当たらない.

同じ年, システムを商用並列プロセサ上の PIM のエミュレータ上に移植した. このエミュレータは, 複数のプロセサが共有メモリバスを持つクラスタを疎結合した構造のモデルのための核言語処理系をデバッグするために作ったものである. このエミュレータのスケジューリング方針は Multi-PSI や PIM/m とはまったく異なり, クラスタ内のプロセサについてはゴールの分散は処理系が自動的に行なう. 期待通り, 同時に意外にも, PIMOS は何の

問題もなく動作する一方，言語処理系の問題をいくつか明らかにした．

現在 (1992 年 2 月)，核言語処理系と PIMOS を PIM の他のモデルに移植する作業が進行中である．今や PIMOS をこうしたモデルに移植する際に根本的な問題が出てくることはないと確信できる．

5.2 細粒度並行性

アルゴリズム設計者は計算を逐次的な過程とみなしやすく，多くのプロセスを協調させてひとつの問題を解かせるためには余計な努力が必要である，というのは事実である．これは往々にして並列処理に不利な，並列計算の設計は人間にとって不自然であることを示す事実と見做されてきた．しかし，核言語の持つ暗黙の並行性は，面白い現象を招いた．

多くのアルゴリズムは実際逐次処理，または低い並列性を念頭に設計されている．このアルゴリズムを核言語で記述すると，核言語の暗黙の細粒度並行性のため，プログラムが設計者が思っていた以上の並行性を示すことが多い．設計者が客観的にプログラムを眺めると，別の暗黙の並行性を発見できる．こうして見つかった並行性が，効率向上のための物理的並列性を高める良い材料となることもある．その場合は，この並行性を引き出すマッピングプラグマをプログラムに追加し，より高い並列性を持ったより効率的なプログラムにすることができる．もし言語が大粒度の並列性しか持っていなかったら，こうは行かなかっただろう．

5.3 記述力

PIMOS の開発を通して，KL1 の並行・並列両面での記述力が十分であることは証明された．

リアクティブなシステムを記述できるため，外部入出力聞きを制御する言語プリミティブを整合性良く持たせることができた．論理を超越するような要素を取り込むことなく，外部機器を普通のプロセスのようにモデル化できたのである．これによって仮想マシンの素直な実現が可能になり，開発効率は大きく向上した．

核言語の荘園構造と優先度制御機構は，システム中のさまざまな活動の実行を制御するのに十分な機能を持っていた．たとえば，ユーザプログラムが無限ループに入り込んでしまった場合は，以下の手順でそのようなプログラムを中断できる．

- デバイスハンドラはユーザのプロセスよりも優先度が高いので，キーボードからの割り込みをセンスできる．
- コマンドシェルはユーザタスクだが，その下で走らせるジョブを自分より低い優先度にしておくので，この割り込みをセンスできる．
- 荘園構造を用いて，シェルは無限ループに入ったタスクを止めることができる．

5.4 プログラムの容易さ

最初に応用ソフトウェア開発にシステムを使い始めたころ，多くのプログラマは核言語に落ち着かないものを感じたようだ．この問題の最大の原因は，プログラミングスタイルが自由過ぎる点にあったようである．

核言語そのものでは論理変数は複数の入出力モードを持っても良い．同じプロセスが状況に応じて，同じ共有変数を読んでも書いても良い．これは言語としては許されているのだが，競争状態になり，特定のスケジューリングにおいてだけ出てくるような問題を起こす．トレースしたりデバッグ用の情報を報告するようにプログラムを書き換えたりすると，スケジューリングが変わってしまい問題が隠れてしまうことがあるので，このようなバグを直すのは難しい．論理変数の入出力モードが決まっているようなプログラミングスタイルが，しだいに確立されていった．これは言語のサブセット化の方向を示唆するものである (6 節参照)．

もうひとつの問題は，数多くの並行プロセスをどのように組織するかだった．多くのスタイルが試され，オブジェクト指向プログラミングスタイル [Shapiro and Takeuchi 1983] が事実上の標準として受け入れられていった．このオブジェクト指向スタイルの上に多くのプログラミングイディオムが確立されてきた [Chikayama 1991]．これは，より高いレベルの言語の設計の方向を示唆するものである (3.4 節参照)．

自動的なデータフロー同期によって，低レベルでの同期の問題は一掃され，プログラマはもっと高いレベルの問題に集中できるようになった．プログラミングスタイルが確立され，ソフトウェア開発環境が経験の蓄積に基づいて拡張されてきたおかげで，核言語による並列ソフトウェア記述は逐次プログラムを他の記号処理言語，たとえば Lisp などで記述するのに比べて，特に難しいものではなくなった．

残る最大の問題は，効率的実行のための計算マッピングアルゴリズムの設計である．言語設計上正当性と効率性に関わることがらを分けたこと，視覚的な性能解析ツールを用意したことで，マッピングアルゴリズムの実験はかなり容易になったが，依然マッピングアルゴリズムの設計は容易な仕事ではない．この方向のさらなる研究が必要だろう．

6 将来の研究開発

並列推論マシン，KL1 言語処理系，PIMOS からなる現在の並列推論システムの問題として，システムが専用のハードウェアでしか動作しないということがある．このシステムでは KL1 プログラムを非常に効率的に実行できるのだが，専用ハードウェアが必要であることは，この

環境を全世界の研究者で共有するのに深刻な障害である．Unix 上で動作するポータブルな核言語の処理系はあり，ソフトウェア開発の初期段階には使われてきたが，これは抽象マシンのインタプリタになっており，性能的には限られたものであるので，本格的な実験的研究には不適当である．

この問題を解決するために，もっと簡単で効率の良い処理系ができるように，言語のサブセットを作る研究を行ない，うまくいきそうな見通しを得た [Ueda and Morita 1990]．これとは別に，KL1 を C にコンパイルする実装の研究も行ない，まずまずの性能を非常に高いポータビリティで実現する目処も立った [Chikayama 1992]．こうした成果を見ると，並列ソフトウェア研究に有効な核言語処理系を商用ハードウェアの上に実装することは可能である．そのような実装に加え，PIMOS，ことにそのソフトウェア開発環境も商用ハードウェアの上に移植し，大規模並列知識情報処理システムの研究開発の基礎として提供することが望ましい．

7 おわりに

日本の FGCS プロジェクトにおける，並列推論システムの基本ソフトウェアの研究開発を概観した．

システムの目的は，大規模並列計算機システムのためのソフトウェア技術の基礎を確立することにある．研究開発にあたっては，まずプログラム言語を設計し，それから応用ソフトに向かう上向きの研究開発と，ハードウェアアーキテクチャに向かう下向きの研究開発を同時に行なうミドルアウトのアプローチを取った．

1988 年から，並列推論マシンの実験機 Multi-PSI や，並列推論マシン PIM のひとつのモデルの上で動いているシステムを，応用ソフトウェアの研究開発に利用してきた．我々の経験から，核言語は並列処理システムのオペレーティングシステムや種々の応用ソフトウェアに十分な記述力を持つことがわかった．正当性と効率性に関わることがらを分離する言語の機能と，オペレーティングシステムが提供するプログラミング環境によって，並列ソフトウェアの実践的研究は従来の環境に比べてはるかに容易になった．

計算のマッピングに関する研究は，今後とも継続していく必要がある．また，商用ハードウェア上に効率的で快適な環境を開発していくことも，残された重要な仕事である．

謝辞

KL1 と PIMOS の設計と実装は，ここに掲げきれない多くの研究者の共同作業によるものである．この稿の初期の版に有用なコメントをしてくれた上田和紀君に感謝する．

参考文献

[Aikawa 1992 *et al.*] S. Aikawa, K. Mayumi, H. Kubo, F. Matsuzawa and T. Chikayama. ParaGraph: A Graphical Tuning Tool for Multiprocessor Systems. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1992*, ICOT, 1992.

[Archibald and Bare 1986] J. Archibald and J. L. Bare. Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. In *ACM Trans. on Computer System*, Vol. 4, No. 4 (1986), pp. 273–298.

[Burton 1985] F. W. Burton. Speculative Computation, Parallelism and Functional Programming. In *IEEE Trans. Computers*, Vol. C-34, No. 12 (1985), pp. 1190–1193.

[Chikayama 1984] T. Chikayama. Unique Features of ESP. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1984*, ICOT, 1984, pp. 292–298.

[Chikayama 1991] T. Chikayama. For KL1 Programming without Tears. In *Proc. KL1 Programming Workshop '91*, ICOT, 1991, pp. 8–14. *in Japanese.*

[Chikayama 1992] T. Chikayama. A Portable and Reasonably Efficient Implementation of KL1. To appear as an ICOT Tech. Report, ICOT, 1992.

[Chikayama and Kimura 1987] T. Chikayama and Y. Kimura. Multiple Reference Management in Flat GHC. In *Proc. Fourth Int. Conf. on Logic Programming*, MIT Press, 1987, pp. 276–293.

[Chikayama *et al.* 1988] T. Chikayama, H. Sato and T. Miyazaki. Overview of the Parallel Inference Machine Operating System (PIMOS). In *Proc. Int. Conf. on Fifth Generation Computer Systems 1988*, ICOT, 1988, pp. 230–251.

[Clark and Gregory 1981] K. L. Clark and S. Gregory. A Relational Language for Parallel Programming. In *Proc. ACM Conf. on Functional Programming Languages and Computer Architecture*, ACM, 1981, pp. 171–178.

[Clark and Gregory 1983] K. L. Clark and S. Gregory. PARLOG: A Parallel Logic Programming Language. Research Report DOC 83/5, Dept. of Computing, Imperial College of Science and Technology, 1983.

[Clark and Gregory 1984] K. L. Clark and S. Gregory. Notes on Systems Programming in PARLOG. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1984*, ICOT, 1984, pp. 299–306.

[van Emden and de Lucena Filho 1982] M. H. van Emden and G. J. de Lucena Filho. Predicate Logic as a Language for Parallel Programming. In *Logic Programming*, K. L. Clark and S. -Å. Tärnlund (eds.), Academic Press, 1982, pp. 189–198.

[Foster 1987] I. Foster. Logic Operating Systems: Design Issues. In *Proc. Fourth Int. Conf. on Logic Programming*, J.-L. Lassez (ed.), MIT Press, Vol. 2, 1987, pp. 910–926.

[Furuichi *et al.* 1990] M. Furuichi, K. Taki, N. Ichiyoshi. A Multi-Level Load Balancing Scheme for OR-Parallel Exhaustive Search Programs on the Multi-PSI. In *Proc. Second ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 1990, pp. 50–59.

[Goto *et al.* 1988] A. Goto, M. Sato, K. Nakajima, K. Taki, and A. Matsumoto. Overview of the Parallel Inference Machine Architecture (PIM). In *Proc. Int. Conf. on Fifth Generation Computer Systems 1988*, ICOT, 1988, pp. 208–229.

[Hircsh *et al.* 1987] M. Hircsh, W. Silverman and Ehud Shapiro. Computation Control and Protection in the Logic System. In *Concurrent Prolog: Collected Papers*, Ehud Shapiro (ed.), MIT Press, Vol. 2, 1984, pp. 28–45.

[Hoare 1985] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[Inamura and Onishi 1990] Y. Inamura and S. Onishi. A Detection Algorithm of Perpetual Suspension in KL1. In *Proc. Seventh Int. Conf. on Logic Programming*, MIT Press, 1990, pp. 18–30.

[Itoh *et al.* 1992] F. Itoh, T. Chikayama, T. Mori, M. Sato, T. Kato and T. Sato. The Design of the PIMOS File System. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1992*, ICOT, 1992.

[Kondoh and Chikayama 1988] S. Kondoh and T. Chikayama. Macro Processing in Prolog. In *Proc. Fifth Int. Conf. and Symp. of Logic Programming*, 1988, pp. 466–480.

[Konishi *et al.* 1992] K. Konishi, T. Maruyama, A. Konagaya, K. Yoshida, T. Chikayama. Implementing Streams on Parallel Machines with Distributed Memory. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1992*, ICOT, 1992.

[Levy and Silverschatz 1989] E. Levy and Z. Silberschatz. Distributed File Systems: Concepts and Examples. Tech. Report TR-89-04, Dept. of Computer Science, The University of Texas at Austin, 1989.

[Maher 1987] M. J. Maher. Logic Semantics for a Class of Committed-Choice Programs. In *Proc. Fourth Int. Conf. on Logic Programming*, MIT Press, 1987, pp. 858–876.

[Milner 1989] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[Miyazaki *et al.* 1985] T. Miyazaki, A. Takeuchi and T. Chikayama. A Sequential Implementation of Concurrent Prolog Based on the Shallow Binding Scheme. In *Proc. 1985 Symp. on Logic Programming*, IEEE, 1985, pp. 110–118.

[Nakashima 1987] H. Nakashima and K. Nakajima. Hardware Architecture of the Sequential Inference Machine PSI-II. In *Proc. 1987 Symp. on Logic Programming*, IEEE, 1987.

[Nitta *et al.* 1991] K. Nitta, K. Taki and N. Ichiyoshi. Experimental Parallel Inference Software. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1992*, ICOT, 1992.

[Okumura and Matsumoto 1987] A. Okumura and Y. Matsumoto. Parallel Programming with Layered Streams. In *Proc. 1987 Symp. on Logic Programming*, IEEE, 1987, pp. 224–231.

[Shapiro 1983] E. Y. Shapiro. A Subset of Concurrent Prolog and Its Interpreter. Tech. Report TR-003, ICOT, 1983.

[Shapiro 1986] E. Y. Shapiro. Systems Programming in Concurrent Prolog. In *Logic Programming and its Applications*, M. van Canegham and D. H. D. Warren (eds.), 1986, Ablex Publishing Co., 1986, pp. 50–74.

[Shapiro and Takeuchi 1983] E. Shapiro and A. Takeuchi. Object-oriented Programming in Concurrent Prolog. In *New Generation Computing*, Vol. 1, No. 1 (1983).

[Susaki and Chikayama 1991] K. Susaki and T. Chikayama. A Process-Oriented Language AYA upon KL1. In *Proc. KL1 Programming Workshop '91*, ICOT, 1991, pp. 117–125. *in Japanese.*

[Takeda *et al.* 1990] Y. Takeda, H. Nakashima, K. Masuda, T. Chikayama and K. Taki. A Load Balancing Mechanism for Large Scale Multiprocessor Systems and its Implementation. In *New Generation Computing*, Vol. 7, No. 2 (1990), pp. 179–195.

[Taki 1992] K. Taki. Parallel Inference Machine PIM. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1992*, ICOT, 1992.

[Taki 1992] K. Taki. Parallel Inference Machine PIM. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1992*, ICOT, 1992.

[Tamaki 1987] H. Tamaki. Stream-Based Compilation of Ground I/O Prolog into Committed-choice Languages. In *Proc. Fourth Int. Conf. on Logic Programming*, MIT Press, 1987, pp. 376–393.

[Ueda 1986] K. Ueda. Guarded Horn Clauses. In *Logic Programming '85*, E. Wada (ed.), Lecture Notes in Computer Science 221, Springer-Verlag, 1986, pp. 168–179.

[Ueda 1987] K. Ueda. Making Exhaustive Search Programs Deterministic. In *New Generation Computing*, Vol. 5, No. 1 (1987), pp. 29–44.

[Ueda 1988a] K. Ueda. Guarded Horn Clauses: A Parallel Logic Programming Language with the Concept of a Guard. In *Programming of Future Generation Computers*, M. Nivat. and K. Fuchi (eds.), North-Holland, 1988, pp. 441–456.

[Ueda 1988b] K. Ueda. Theory and Practice of Concurrent Systems. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1988*, ICOT, 1988, pp. 165–166.

[Ueda 1990] K. Ueda. Designing a Concurrent Programming Language. In *Proc. InfoJapan'90*, Information Processing Society of Japan, 1990, pp. 87–94.

[Ueda and Chikayama 1985] K. Ueda and T. Chikayama. Concurrent Prolog Compiler on Top of Prolog. In *Proc. 1985 Symp. on Logic Programming*, IEEE, 1985, pp. 119–126.

[Ueda and Chikayama 1990] K. Ueda and T. Chikayama. Design of the Kernel Language for the Parallel Inference Machine. In *The Computer Journal*, Vol. 33, No. 6 (1990) pp. 494–500.

[Ueda and Furukawa 1988] K. Ueda and K. Furukawa. Transformation Rules for GHC Programs. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1988*, ICOT, 1988, pp. 582–591.

[Ueda and Morita 1990] K. Ueda and M. Morita. A New Implementation Technique for Flat GHC. In *Proc. Seventh Int. Conf. on Logic Programming*, MIT Press, 1990, pp. 3–17. A revised, extended version to appear in *New Generation Computing.*

[Yashiro *et al.* 1992] H. Yashiro, T. Fujise, T. Chikayama, M. Matsuo, A. Hori and K. Wada. Resource Management Mechanism of PIMOS. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1992*, ICOT, 1992.

[Yoshida and Chikayama 1990] K. Yoshida and T. Chikayama. A'UM: A Stream-Based Object-Oriented Language. In *New Generation Computing*, Vol. 7, No. 2 (1990), pp. 127–157.