TM-1038

推論型言語の並列処理方式

市吉　伸行

April, 1991

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03)3456-3191 ~5
Telex ICOT J32964

Institute for New Generation Computer Technology

論理型言語の並列処理方式 *

市吉 伸行 †

1 はじめに

論理型言語の登場以来，その逐次処理方式は目覚ましい進歩を遂げてきた．Warren Abstract Machine (WAM) と呼ばれる時間・メモリ効率の良い処理方式が標準的逐次処理方式として普及し，また，静的解析などによる様々な最適化あるいは専用マシンによって，逐次実行性能をさらに向上させる研究も数多くなされてきている [63, 59]．しかし，多大な計算量を必要とする応用問題も多くあり，大幅な性能向上への要求は際限がない．逐次実行の数倍から数百倍以上の性能向上を実現し得るものとして並列実行が注目され，最近 6，7 年の間に並列処理方式の研究開発が多くの大学や研究機関で活発に行なわれて来ている．特にここ数年，並列計算機が広く普及しはじめ，本格的な並列処理系が稼働し始めている．

通常の並列プログラム開発では，逐次プログラムと比べると，同期，排他制御，プロセス間通信，負荷分散などの問題が新たに加わるためプログラムの設計，デバッグが難しい．また現状では，いくつもの異なるアーキテクチャの並列計算機があり，それぞれに依存した通信プリミティブや共有変数宣言を手続き型言語に追加した並列言語でプログラムを書かなければならない．そのため，せっかく開発した並列ソフトウェアのポータビリティを悪くしている．

論理型プログラムは並列実行した場合でも，プログラムの意味 (宣言的意味) を処理系が保証するので，並列化に伴う同期等に関する低レベルのバグが入り込まず，また，よりポータブルな並列プログラムが開発できる．したがって，論理型言語のメリットが並列処理の世界でよりよく発揮できると期待される．

では，現在の処理系技術水準は上記の理想にどれだけ近づいているであろうか？ 本稿では，論理型言語の並列処理方式の現状をなるべく広く紹介したいと思う．論理プログラミングおよび論理型言語逐次処理方式の基礎知識を仮定するが，それらについては文献 [61, 63] などを読んで頂きたい．

---

*Parallel Implementation Schemes of Logic Programming Languages by Nobuyuki ICHIYOSHI (Institute for New Generation Computer Technology)

†(財) 新世代コンピュータ技術開発機構

2 論理型言語の並列性

2.1 論理プログラムの並列性

Kowalski は，導出ステップを手続き呼出しと対応させることによって，反駁手続きを再帰的プログラムの実行と見做せることを指摘し，論理型言語 Prolog の理論的基礎を築いた．AND 関係にあるゴールを左から右に順に実行し，OR 関係にあるホーン節を上から下に順に試行するという Prolog の計算過程は，論理プログラムの一つの逐次実行モデルであり，1 つずつ順に処理せずに並列に反駁手続きを進めても正しい答を出すことができる．

論理プログラムの中のどの並列性を取り出すかによって，並列実行モデルが分類される．

2.2 並列実行モデル

(1) AND 並列 /OR 並列実行

Conery [11] は，論理プログラムの持つ色々な並列性を指摘し，また，反駁手続きの持つ AND 並列性と OR 並列性を同時に実現するメッセージ通信モデルである AND/OR プロセスモデルを記述した．AND/OR プロセスモデルは論理プログラムの並列実行方式の最初の具体的提案という意義があったが，AND/OR 木のノード (小粒度) がメッセージ通信しながら計算を進めて行くという方式だったため，並列化オーバヘッドが大きく，試作システムに止まった．

その後，Prolog の高い逐次処理性能を犠牲にしない並列実行を設計目標とした処理方式が活発に研究されるようになった．普通に書いた Prolog プログラムをそのまま並列処理系で走らせて，プロセッサ台数分に近い速度向上が得られる — したがって，ユーザは速度向上以外には並列実行を意識しない — というのがその理想像である．

とりわけ，OR 並列実行のための方式と AND 並列実行のための方式が最もよく研究され，大規模なプログラムを実行できる処理系が開発された．3，4節でそれらについて解説する．

(2) 並行論理型言語

上記のような，ホーン節に関して完全な証明体系である反駁手続きの (平たく言うと Prolog の言語仕様

を保った) 並列化に対して, 完全性を犠牲にして別の方向の並列性に発展した並行論理型言語 (concurrent logic language) と呼ばれるグループがある. Concurrent Prolog [44], Parlog [9], GHC [49], KL1 [50], Strand [15] などがそうである.

並行論理型言語の実行モデルでは, OR ノードの下の枝を全解探索しようとする代わりに, 入力引数のテストによって 1 つの枝を非可逆的に選択する (コミットする).[1] 並行論理型言語の処理方式については 5節において解説する. 文献 [60] では, 並行論理型言語向けの並列マシンのアーキテクチャについて詳しく述べているので, 参考にして頂きたい.

(3) その他の並列実行モデル

3, 4, 5節で取り上げなかったその他の並列実行モデルについて 6節で触れる.

3 OR 並列処理方式

3.1 OR 並列実行と多重環境

逐次 Prolog は反駁木[2]を深さ優先探索していたが, OR ノードの下の複数の分岐を並列に探索するのが OR 並列実行である. 抽象的手続きとしてはこのように単純だが, 実装上は次のような問題がある.

例えば,

```
:- p(X), q(X).
p(a). p(b).
q(b). q(c). q(d).
```

というプログラムを考えてみよう. 反駁木のルートからゴール p(X) の実行が終了した時点までの部分は図-1 のようになる. ここで, 分岐 C1 と C2 は p/1 を定義する 2 つの節に対応している.

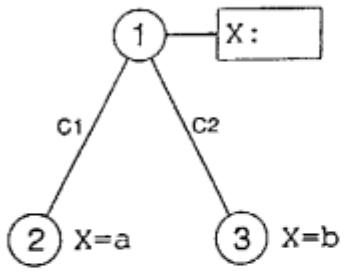

図 1: 多重環境

通常の逐次処理系による実行では以下のような順で処理が進む. ノード 1 において変数セル X を割当て, p/1 の最初の C1 を試行する. 節 C1 のヘッドユニフィケーションの結果として, 変数セル X にアトム a が書き込まれる. その後, ゴール q(X) が失敗して, ノード 1 にバックトラックする際に, 変数セル X を未束縛状態に戻す (undo する). 次に, p/1 のもう一つの節 C2 を試行する. 節 C2 のヘッドユニフィケーションでは, 変数セル X にアトム b が書き込まれ, ゴール q(X) の呼出しが成功する.

ここで, 変数束縛の方式をそのままにして分岐 C1 と C2 とを並列に試行すると, 変数セル X への書き込みが衝突し, 正しい結果が得られない. このように, OR 並列実行においては, 同時に存在し得る複数の OR 分岐の変数束縛環境 (多重環境という) を, どのように正しく実現するかが問題となる.

ひとつの素朴な解決方法は, 異なる OR 分岐が同一の変数セルを共有しないようにすることである. そのために, 親ノードの下に子ノードを作る際に, 親ノードの変数セルを, 子ノードから指すのでなく, 子ノードにコピーしてしまう. 変数を含んでいる可能性のある構造体もコピーしなければならないので, OR ノードに付随したゴール列の大半をコピーすることになる.[3] しかし一般に, コピーの手間は非常に大きく, この方式は効率が悪い.

多重環境を正しく扱え, かつ効率のよい方式が模索され, これまでに幾通りもの方式が提案されてきた. 各種の考え方や諸処理のトレードオフなどの問題について興味深い点を多く含んでいるので, それらを紹介し, 最後に比較を行なう. 3.2.6節の方式を除いて, 共有アドレス空間型並列マシン向けの方式である.

3.2 多重環境管理の諸方式

反駁木のうち現時点で展開されている部分を以下では OR 木と呼ぶことにする. 逐次実行ではローカルスタックが OR 木を表現しており, 木と言っても分岐のない線形な形をしている. OR 並列実行では, OR 木が実際に枝分かれした木の形をしている.

3.2.1 Deep binding 方式

これは変数束縛を束縛リスト (Lisp でいう連想リスト) で表現するという一番素朴な方法である. この方式では, 各ノードが連想リストへのポインタを持ち, 新しい変数束縛に対応して連想対を追加し, それを子ノードへ継承する. 図-1 の例では, ノード 2 が変数 X に値 a を束縛する際に, 変数セルに値を直接書き込ま

[1] Prolog および並行論理型言語の OR ノードにおける非決定性を, それぞれ don't-know nondeterminism (OR ノードの下のどの枝が正しいか分からないので全てを試みる), don't-care nondeterminism(テストに成功したどの枝を選んでも構わない) と呼んで区別することがある.

[2] Lloyd [33] では SLD 木.

[3] 人間が紙の上で反駁手続きを模倣する時はこのような方法によっている.

ず, < X, a > という連想対を連想リストに追加するようにする. ノード3でも同様である. 逐次処理方式と比べると, 各ノードにおいて, そのノードで行なった束縛の数に比例するメモリ領域が余分に必要であるだけである. しかしながら, 変数値を参照するためには, 自ノードから出発して, その変数の値を示す連想対が現れるまで (未束縛の場合は変数の属するノードまで) 連想リストを手繰らねばならず, OR 木が深くなるにつれて変数アクセス時間が長くなるという欠点がある.

3.2.2 タイムスタンプ方式

OR 木において変数は異なる OR 分岐によって異なる値を持つ. タイムスタンプ方式 [48] では, 変数は, それら全ての変数値を表わすようなデータ構造 (仮に変数値リストと呼ぶ[4]) を指すポインタとして表現される. ある変数に値を束縛する時には, 束縛値と共にどの OR ノードでの束縛かをあらわすためにタイムスタンプを付加して変数値リストに加える. OR ノードからの変数値参照においては, 変数値リストの中からタイムスタンプ情報を用いてそのノードの祖先ノードを探し, それに対応する値を得る (そのような祖先ノードがなければ未束縛).

この方式は, 祖先ノードの判定を可能にするタイムスタンプの表現や判定アルゴリズムが複雑になること, 部分木の探索が終了した際の不要になった変数値の解放が難しいこと, など多くの困難を抱えている.

3.2.3 ハッシュウィンドウ方式

Deep binding 方式では変数と変数値の対応を線形リストで表わしていたが, 別のデータ構造を使えば速い変数値アクセスができると考えられる. 例えば, 変数アドレスをキーにしてハッシュ表に変数値を登録すれば (平均) 定数時間で変数値アクセスができよう. ただし, OR ノードによって同じ変数に対する変数値は一般に異なるので, 各ノードでハッシュ表を持つ必要がある. これをハッシュウィンドウ方式という. OR ノード生成時のハッシュ表の初期化では変数値をハッシュ表に登録せず, 実際の変数アクセスの度に, 既に自分のハッシュ表に登録されているかを調べて, もし値がなければ祖先ノードのハッシュ表を順に調べ, 見つかったところで自分のハッシュ表に登録する. このような lazy な変数値登録によって, アクセスしない変数の登録を省くことができる [7].[5]

PEPSys 処理系 [55] ではハッシュウィンドウ方式の変形を採用しているが, ベンチマークによれば, 大半の変数値アクセスは 1 つないし 2 つのハッシュウィンドウを手繰ることで済むという [6].

3.2.4 バージョンベクタ方式

タイムスタンプ方式では変数のアクセスを定数時間に抑えられなかったが, プロセッサ数が決められた有限の数であることを利用して変数アクセスを定数時間で行なうための一つの方法がバージョンベクタ (versions vector) 方式 [23] である.

バージョンベクタ方式では, プロセッサ数が p であるとき, 長さ p のベクタ (バージョンベクタ) を指すポインタとして変数を表現し, プロセッサ P_i から見た変数の値がベクタの第 i 要素となるように管理する. これにより変数へのアクセスは, バージョンベクタの先頭アドレスの読出し, オフセット (= プロセッサ番号) を足し込んだ先の読出し, という単純な命令列で済むようになる.

しかし一方, プロセッサが一つの部分木の探索を終えて, 別の OR 分岐の探索に移るという処理 (プロセススイッチ) については, deep binding 方式とタイムスタンプ方式では単にカレントな OR ノードをスイッチするだけの処理でよかったのに対して, バージョンベクタ方式では, プロセッサの実行環境であるバージョンベクタの更新が必要となる. すなわち, (1) バックトラックして旧 OR 分岐の変数束縛を巻き戻 (undo) し, (2) 新 OR 分岐における変数束縛環境を設定 (install) しなければならない. バックトラックは旧ノードと新ノードの共通の祖先ノードまで行なえばよく (差分的プロセススイッチ), プロセススイッチの手間は, OR 木における新旧ノード間の距離 (詳しくはその間の変数束縛数) にほぼ比例する.

変数束縛の巻き戻しには, 逐次処理系と同じトレールスタックを用いればよいが, 変数環境の設定には変数の値が必要なので, トレールの際に変数アドレスと共に束縛値も登録するようにする. このように拡張したトレールスタックは deep binding 方式における束縛リストと同じものであるが, バージョンベクタ方式ではこの束縛の記録を変数アクセスの目的には使わないところが異なる.

3.2.5 束縛アレイ方式

束縛アレイ (binding array) 方式 [54, 52] は変数アクセスを定数時間で行なう別の方式である. この方式では, 反駁木のルートから各々の枝分かれに沿って順に (OR 分岐においては分岐先毎に独立に) 変数に

---

[4] [48] では束縛リストと呼んでいるが, 一般用語としてのそれと混同しやすいのでここでは避けた.

[5] ただし, lazy な登録においては, 変数へのアクセス時に先祖 OR ノードに付属するハッシュ表を順に辿らねばならず, 最悪ケースの処理オーダは deep binding 方式と同じになる.

番号を振る. 例えば,

```
:- r(X).
r(X) :- s1(X,Y), s2(X,Z), t(Y,Z).
r(X) :- u(X,W), v(W).
```

の実行においては図-2 のように変数番号が振られる. また, 各プロセッサには束縛アレイと呼ばれる配列を割り付けておき, 変数番号のエントリに変数値を対応付ける (図-3). OR 木は全プロセッサが共有するが, 束縛アレイは各プロセッサに固有である. 変数値へのアクセスは, 変数セルからの変数番号の読出し (例えば, 変数 X から変数番号 1), 束縛アレイの対応するエントリへのアクセスという単純な処理になる.

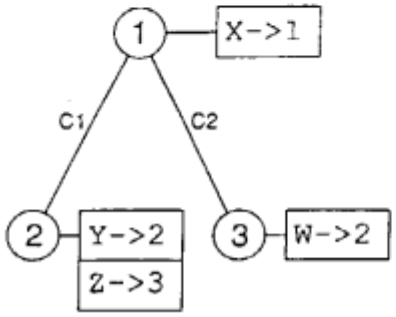

図 2: 変数の番号付け

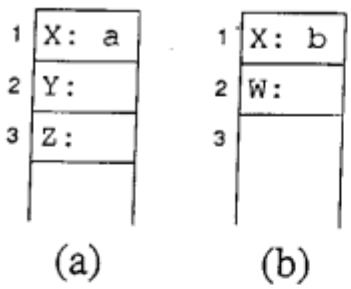

図 3: 束縛アレイ

この方式は, 変数値アクセスの簡単化の他にも以下のような長所を持っている.

(1) 物理的な変数アドレスと論理的な変数番号の切り離し

逐次処理系では変数同士のユニフィケーションにおいて, 新しい変数が古い変数を指すようにしていた. これは, 変数割付け領域をスタック管理した時に, dangling reference が生じないことを保証するためであった. ところが, 並列処理系でプロセッサ毎に複数の変数スタック領域を割り付けるようにすると, アドレスの単純な比較では変数の新旧関係が決められなくなる. 束縛アレイ方式では変数番号の比較で新旧関係が分かる.

(2) メモリアクセスの局所性

バージョンベクタ方式のバージョンベクタは全プロセッサに共有されるので, 変数参照 / 束縛の度に共有データへの読み書きが発生することになる. これに比べて束縛アレイ方式は, 固有データ (束縛アレイ) は頻繁に読み書きし, 共有データ (共有されている OR ノード) は時々読むか, 稀に更新する (主に OR 分岐の生成消滅に伴う情報管理のため) という優れたメモリアクセス特性を持っている.

3.2.6 コピー方式 / 再計算方式

これまでの方式は, スタックやその他のデータをできるだけプロセッサが共有しようという点で共有メモリ並列マシン向きであった. これに対し分散メモリ並列マシンでは, 他プロセッサ上のメモリへのアクセスのコストが, 局所メモリへのアクセスと比べて, 通常 1~2 桁程度の大きさなので, できるだけ各プロセッサが局所的な処理をすることが望まれる.

そこで, プロセッサ毎にスタック等を独立に持ち, プロセッサ内では逐次処理系と同じような処理をすることが考えられる. プロセッサは暇になると仕事を持っているプロセッサに新たな仕事を要求する. 要求を受け取ったプロセッサは未実行の OR 分岐を与えるが, その際に実行環境としてルートからその OR 分岐までのスタック情報を要求側のプロセッサにコピーする. コピー方式を採用すると, スタック共有による多重環境管理の煩わしさからも解放される.

株分け方式 [30] はコピー方式の実験的処理系で, プロセッサの担当する仕事の粒度を大きくするために, 未実行 OR 分岐を残す OR ノードのうちルートに最も近いものを選び, その下の OR 分岐の幾つかを分け与える (株分け) ようにしている. 株分けする側のプロセッサは株分けする OR ノードの下の OR 分岐の中の一つを実行中であり, そこでなされた変数束縛は株分け時に輸出してはならず, 未束縛変数に戻してからコピーしなければならない. そのための変数束縛の有効性判定のために, 変数の束縛時にどのレベルの OR ノードによる束縛かという情報を付加する. これもタイムスタンプの一種だが, 単なる自然数でよいのでオーバヘッドは小さい.

実行環境をコピーする代わりにルートからの実行トレースを教え, 実行トレースを受け取ったプロセッサがそれに従って実行を再現する再計算方式も提案されている [10, 45]. 実行トレースとは各 OR ノードで何番目の候補節を選んだかという情報であり, スタックのコピーよりも送るデータ量が小さくてすむと期待される. この方式の一つの大きな問題点は, 副作用

のあるプログラムの逐次セマンティックスを保存するのが困難なことである (例えば, 実行再現中に read 述語や write 述語のある場合を考えてみよ).

3.2.7 諸方式の比較

OR 分岐の並列実行に伴う多重環境の管理について各種方式を紹介してきた. 素朴な deep binding 方式はプロセススイッチが定数時間でできるが, 変数値アクセスが定数コストに抑えられなかった. バージョンベクタ, 束縛アレイ方式, コピー方式 / 再計算方式では, 変数値アクセスが定数時間でできたが, プロセススイッチが定数コストで抑えられなかった.[6] 一方, 3.1節初めに述べたナイーブな方式では, 変数値アクセスおよびプロセススイッチ (「カレントな OR ノード」を変えるだけ) が定数時間だが, OR ノードの生成が定数時間でない.[7]

Gupta と Jayaraman [19] は, 環境生成 (OR ノードを作ること), 変数アクセス, プロセススイッチ, の三者を同時に定数時間に抑えられるような OR 並列処理方式が存在しないことを証明した. そこで, これらのトレードオフをどのように選ぶかによって処理系が性格付けられることになる. このうち, どのような変数アクセスと環境生成が起きるかはプログラム (と与える質問) によって決まるが, プロセススイッチはプロセッサ数やスケジューリング戦略に依存する. したがって, 前二者を定数時間に抑えておいて, スケジューリングによってプロセススイッチのオーバヘッドを全体として低く抑える工夫をするのが好ましいだろうと [19] は述べている.

3.3 Aurora 処理系

処理系の具体例として Aurora [34] を紹介する. Aurora は, 米国の Argonne 国立研究所, 英国の Manchester 大学, スウェーデンの SICS 研究所が中心となった非公式的な Gigalips プロジェクトの中で共同で研究開発された OR 並列処理系である.

3.3.1 処理系の特徴

Aurora 処理系は SRI モデル [53] という方式を採用している. SRI モデルは変数束縛に束縛アレイを用いた OR 並列実行モデルで, 変数束縛方式以外にメモリ管理, スケジューリング規則などを記述している. Aurora の特徴を記す.

(1) ワーカ

SRI モデルではプロセッサを抽象化してワーカと呼ぶ. ワーカはエンジンとスケジューラという 2 つの「ペルソナ」(顔) を持っており, OR 分岐の試行 (本来の仕事) をしている間はエンジンであり, 一つの OR 分岐の試行が終わり新たな仕事を探す間はスケジューラとなる. 自分の受け持つ OR 分岐以前にまで遡るバックトラック (public backtracking) はその OR 分岐の試行終了を意味し, エンジンはスケジューラになり変わり, ある OR 分岐にスイッチするとスケジューラがエンジンになり変わって実行に取り掛かる.

(2) サボテンスタック (cactus stack)

逐次実行と違って, OR 並列実行では OR 木が線形でないので, 線形スタックを分岐のあるサボテンスタックに拡張している. 各プロセッサは実行中の OR 分岐のルートから始まり, 現在実行中のノードまでの線形な部分を自分のスタック領域に展開する (OR 木のルートを下にすると丁度サボテンのように, いくつもの OR 分岐が途中から生えて上に伸びている).

(3) スケジューリング

SRI モデルでは仕事の粒度をできるだけ大きくするために株分け方式と同様に, スケジューリング規則として, ルートからの各経路上で一番ルートに近い未実行 OR 分岐のみをプロセススイッチの対象としている. (逆に言うと, 祖先ノードに未実行分岐が残っているような OR ノードの分岐はプロセススイッチの対象としない.)

(4) OR 分岐の中断 / 放棄

副作用のある述語 (入出力やデータベース更新) やカットなど非論理的述語は正しい順序で実行しないと, Prolog の逐次実行と同じプログラムの意味を保てない. そのため, Aurora 処理系では OR 分岐の中断機能を導入している. すなわち, ワーカは実行中の OR 分岐を中断して, 他の実行可能な OR 分岐にスイッチすることができるただし, サボテンスタックの管理はこのために複雑になる.

例えば, 副作用のある述語の実行は, 逐次実行において自分より先に試行される部分 (反駁木において自分より左にある部分) の実行が終了するまで中断し, 自分が OR 木において最左分岐になった後に実行すれば, 逐次実行と同じ順を保つことができる.

カットも他の OR 分岐の実行に影響を与え, しかも他のカットによって枝刈りされる可能性があるので, 実行のタイミングに気をつけなければならない. 最左分岐になるのを待ってカットを行なえば安全だが, カットの時期が遅れることによって見込み計算[8] の量が増えるので, 安全な部分的枝刈りを早期に行なう方

[6] 逐次処理方式の WAM でも, プロセススイッチにあたるバックトラックが定数時間処理でない.

[7] ここでの変数値アクセスとは, 正確にはデレファレンス一段分のことである.

[8] Speculative computation. 計算結果が不要となる可能性のある計算. カットによって枝刈りされるかも知れない計算を進めるのは, 見込み計算をしていることになる.

法が工夫されている [24].

3.3.2 性能その他

Aurora処理系は共有バス型並列計算機 Symmetry 上に実装され, ベースとなった SICStus Prolog と比べて逐次実行において 20% 程度のオーバヘッドがあり, クイーン問題など全解探索型のプログラムでプロセッサ台数倍に近い速度向上を実現している. Aurora 処理系はこれまでに 4 通りのスケジューラが開発されたにも係わらず, ガーベジコレクタが実装されていないために, 本格的な応用プログラムを実行できないという問題がある.

なお, SICS ではコピー方式を用いた Muse (*M*ulti-*s*equential Prolog engines) [1] 処理系も開発されている. Muse では実行環境コピーの際に OR 木ルートからコピーする代わりに差分だけ処理するようにしている. またその際に, 仕事を与える側のプロセッサはバックトラックを模倣して, 与えようとする OR ノードにおける変数束縛を再現する. このため通常実行時に, 株分け方式のようなタイムスタンプが不要となっている.

Muse も Symmetry に実装され, 機能の違いがあるので Aurora と直接的比較はできないものの, コピー方式であるために, 逐次実行におけるオーバヘッドは 6% 程度と, Aurora よりも低く抑えられている.

4 AND 並列処理方式

4.1 AND 並列実行の諸モデル

OR 並列実行は反駁木の並列探索と思えばよかったが, ゴール間の AND 関係はホーン節間の OR 関係と違って, データ依存性があるので, AND ゴールを単に並列に実行することはできない. 例えば, AND 関係にあるゴールを左から順に実行するという逐次 Prolog の規則を弱めて, 複数のゴールを単に並列に実行すると, 各ゴールの実行結果である変数束縛は整合性を持たない. AND 並列実行ではゴール間の変数共有を考慮した実行モデルが必要であり, 以下のようなモデルが提案されてきている.

(1) AND/OR プロセスモデル

まず, 変数束縛の矛盾が生じる可能性のあるゴールを並列に実行しないことで上記の問題を避けることが考えられる. そのためには, 変数を共有するゴール間に適当な実行順序を導入すればよい. 例えば,

```
p(X,W) :- a(X,Y),b(X,Z),c(Y),d(Y,Z,W).
```

という節において, ゴール a(X,Y) とゴール b(X,Z) のの実行後にそれぞれ変数 Y と変数 Z の値が決まっている (基底項である) としよう. この時, c(Y) を a(X,Y) の実行終了後に, また d(Y,Z,W) を a(X,Y) と b(X,Z) の実行終了後にそれぞれ実行を開始すれば変数束縛の矛盾は生じない. (ゴール c(Y) と d(Y,Z,W) は変数 Y を共有しているが, (仮定により) 実行開始時に値が決まっているので, 変数束縛の矛盾は生じない.)

Conery の AND/OR プロセスモデルの AND 並列実行に関する部分は, この方式に従っており, 適当な実行順序を予め静的に決めておき, 節のボディ部実行開始時にゴール間を矢線で結ぶことによってこの半順序を表わす. 自分に入る矢線のない (必要とするデータが揃っている) ゴールは実行可能であり, それらが複数あれば並列に実行する. 実行が終わったゴールはそれから出ている矢線を消す (自分の供給すべきデータを供給した), というようにして実行が進む.

「適当な実行順序」を決めるためには, モード解析またはユーザによるモード指定によって, ゴールの入出力モードを知り, ゴール間のデータ依存関係を求める必要がある. 依存関係が決められない場合は, 適当な順 (例えば, 左から右) に逐次に実行すればよいが, 並列性は低くなる.

Lin と Kumar は AND-OR プロセスモデルの実行時オーバヘッドの小さい実装を提案している (4.2節).

(2) ストリーム AND 並列

もっと小粒度の並列性を取り出すために, データ依存関係における下流ゴールを上流ゴールと並列に実行し, 必要なデータが上流ゴールによってまだ用意されていなければ下流ゴールが中断してデータを待つというパイプライン的実行方式も考えられる. これをストリーム AND 並列実行と呼ぶ (4.3節). 並行論理型言語はストリーム AND 並列性を持つが, これについては次の 5節で別に取り上げる.

(3) 制限 AND 並列

AND/OR プロセスモデルやストリーム AND 並列モデルにおける実行時オーバヘッドを避けるために, 静的解析により, ボディ部を逐次に実行されるブロックに分け, 各ブロックに属するゴール間には互いにデータ依存性のないことを保証してそれらを並列に実行する, という方式が考えられたが, かなり精密な大域的な静的解析ができないと余り並列度が取り出せなかった. そこで DeGroot は, 静的解析にデータ依存性に関する若干の実行時検査を組み合わせるというアイデアを提案し, これを制限 AND 並列 (Restricted AND-Parallelism (RAP)) と名付けた [14].[9]制限 AND 並列方式が取り出せる並列性は AND/OR プロセス方式よりも小さいが, その分, 実行時オーバヘッドも小

[9]独立 AND 並列 (Independent AND-Parallelism (IAP)) とも呼ぶ.

さい.

RAP が生まれた背景には, 並列度が高ければ高いほど良いのではない, という意識の変化もある. 並列実行の目的が高速化にあるのなら, プロセッサ数以上の並列度をしかも高いオーバヘッドを伴いながら取り出すのは無意味だ, という認識である. 制限 AND 並列処理方式については最も詳しく述べる (4.4節).

(4) AND/OR 並列

これは, 共有変数のあるゴールに実行順序を入れるのではなく, 独立に並列実行する. 各ゴールは OR 関係にある変数束縛情報を生成し, AND 関係にあるゴールの変数束縛情報の組み合わせのうち, 整合性のあるものを全体の結果とする. これについては, AND/OR 並列実行モデル (6.1節) で取り上げる.

4.2 Lin-Kumar 方式

Lin と Kumar は, ゴール依存関係グラフをビットベクタによって表現し, グラフの操作をビット演算/検査によって実現することによって, AND/OR プロセスモデルの実行時オーバヘッドの小さい実現法を示し [32], また, 幾つかの最適化の効果を評価するベンチマークテストを行なっている [31].

4.3 ストリーム AND 並列実行方式

例えば, 下記のような生成テスト型のプログラムを考えてみよう.

```
:- generate(BigX), test(BigX).
```

逐次 Prolog では generate が大きな構造 BigX を作った後に, test が BigX の表層レベルのテストで失敗するかも知れない. もし generate が表層レベルを作ったすぐ後に test に関する実行を進めれば, 早く失敗が分かり, 表層レベル以下の無駄な計算をしないですむだろう. 逐次 Prolog の計算規則では, 常にゴール列の最左ゴールを導出の対象としているが, 上記のような実行はこの規則を緩めることに対応しており, これによりコルーチン機能が実現できる [37]. これはストリーム AND 並列モデルの逐次実行と言える.[10]

ストリーム AND 並列は, AND-OR プロセスモデルや制限 AND 並列よりも細かい並列度を抽出できるが, そのためのオーバヘッドも比較的大きい. ストリーム AND 並列の並列実行については研究が余り盛んとは言えないが, これまでに [38] や [46] などの事例がある. 技術的には, 前向き実行については並行論理型言語の処理方式, 後向き実行については制限 AND 並列処理方式と共通する部分が多い.

[10] なお, コルーチン機能のない Prolog で無駄な計算をしないためには, インクリメンタルにテストしながら構造を生成する述語を定義することになろう. そのようなプログラムは中断/再開のオーバヘッドがないためにコルーチン的実行よりも効率は良いが, プログラムの宣言的意味の明快性は減じよう.

4.4 制限 AND 並列処理方式

4.4.1 独立性検査

次の例を見てみよう.

```
f(X,Y,Z) :- g(X,Y), h(X,Z).
```

f のボディ部のゴール g(X,Y) と h(X,Z) が独立に並列実行できるためには, 共有変数 X が変数を含んでいてはならない (基底項でなくてはならない). そうでないと, データの受け渡しがあるかも知れないからである. また, Y と Z とがユニファイされている可能性があるので, この 2 つが異なる変数であることも必要である. このことを,

```
(ground(X), indep(Y,Z) ->
  g(X,Y) & h(X,Z); g(X,Y), h(X,Z))
```

と表わす.[11] "," も "&" も論理的に AND の意味だが, 前者が逐次実行を示すのに対し, 後者は独立並列実行を示す. ground(X) は X が基底項, すなわち, 未束縛変数を含まない項, であることのチェックであり, indep(X,Y) は X と Y とが未束縛変数を共有しないことのチェックである.

基底性テストと独立性テストは完全である必要はなく, テストが成功した時に必ず基底性ないし独立性が成り立っていればよい. テストが完全に近いほど並列性を上げられるが, 逐次実行の効率を落とさないようにしたいという初期の動機からすれば, これらのテストは軽い処理であるべきである. また, 静的解析によって入力引数の基底性や変数同士の独立性が保証されれば実行時のテストが不要になる.

4.4.2 後向き実行

前節では, 制限 AND 並列方式の前向き実行について述べたが, 本節では後向き実行 (バックトラック処理) について説明する. 次のゴール列の並列実行を考えてみよう.

```
p(X), (q(X) & r(X,Y) & s(X,Z)), t(Y,Z)
```

5 つのゴールを左から順に P, Q, R, S, T と名付けよう. ここで, Q, R, S の並列実行中に R が失敗したとする. 制限 AND 並列実行の性質から, R は Q, S とは全く無関係なので, それらの実行結果に依

[11] これは, &-Prolog [25] の記法である.

らずにこの並列実行全体が失敗する. したがって, R プロセスと並列に走っている Q プロセスと S プロセスを放棄し, P の持つ (かも知れない) チョイス点に戻る. 別のケースとして, 3 つのゴールの並列実行終了後, T が失敗したとしよう. この場合, 最新のチョイス点をまず Q, R, S から探せばよいが, 逐次実行と同じ順で解を見つけるために, 右から左に向かって探す. もし, チョイス点が S の中にあれば, そこまで戻って再試行すればよく, また S になく, R にあれば, チョイス点に戻る過程で, 逐次実行の場合と同様に, S の実行を巻き戻し, チョイス点以降の R を S と並列に再実行する.

ここで注意すべきは, R の失敗時に, Q にバックトラック点があるなしに係わらず, P までバックトラックすることである. このために R の失敗の理由に無関係な Q にバックトラックして, 再び R で失敗するのと比べて無駄な試行錯誤が減っている. このようなバックトラックを依存性に基づくバックトラック (dependency-directed backtracking) と言う. しかし逆に, R が失敗するかも知れないのに S の実行を開始するのは, 逐次実行にない見込み計算をしていることになる.

4.4.3 その他

RAP は DeGroot の後, 主に Hermenegildo によって発展をみた. 制限 AND 並列実行のための抽象機械 RAP-WAM の設計 [27], 副作用の扱い [36], 静的解析手法 [35], などの一連の仕事によって, 完成度を高め, Prolog プログラムの意味を保ったまま, 逐次処理の効率を落とさずに, 並列実行によって速度向上させるという目標をほぼ達成したと言えるだろう [26]. また, 最近では, 制限 AND 条件を緩めて並列性を上げる試みもされている [26].

5 並行論理型言語の処理方式

並行論理型 (concurrent logic language) の節は, ホーン節の右辺が 2 つに区切られたガード付きホーン節 (guarded Horn clause) と呼ばれる次の形をしている.

$$H \ :- \ G_1, \ \ldots \ , \ G_m \mid B_1, \ \ldots \ , \ B_n.$$

ここで, "|" をコミットバー, その左側をガード部, 右側をボディ部と言う. ガード部では入力引数の観測のみが許されており, 述語呼出しの際にはその述語を定義する節のうち, ガード部のテストが成功 (ヘッドとマッチし, ガード部ゴールが成功) したものの 1 つが非決定的に選ばれ, そのボディ部が並列に実行される. ガード部テストの際に, テストの対象となる引数が未定義変数であるとそのテストは入力値が定まるまで中断する.[12]

並行論理型プログラムの典型的なゴールは, 外界を観測して (入力引数のテスト) 意思決定を行ない (コミット), 外界に作用を及ぼし (共有変数への書き込み), 更新された状態で自分を再帰的に呼び出すということを繰り返す. 多くの並行論理型プログラムは, プロセス群が互いに通信し合いながら, 問題を解いてゆくという構造で記述されている.

並行論理型言語の言語仕様は Prolog と異なるので, 処理方式は Prolog 逐次処理方式の並列実行への拡張という形をしていない. したがって本節では, 逐次実行方式も含めて並行論理型言語処理方式を論ずることとする.

並行論理型言語のフラット (flat) なサブセットとは, 限られたシステム組込み述語のみの出現をガード部に許すもので, ガードがネストしないので処理方式を単純化・効率化できる. 並行論理型言語処理系の多くはフラットな言語を対象としているので, 以下ではフラットな言語の処理方式のみについて述べる.

5.1 プロセス指向処理方式

これまでに開発された, または開発中の並行論理型言語の並列処理系として, Flat Concurrent Prolog (FCP) 処理系 [47], マルチ PSI/PIM 上の KL1 処理系 [39, 18](以下単に, KL1 処理系と呼ぶ), Flat Parlog 処理系 [16], Strand 処理系 [15] などがある. (フルセットの言語の処理系としては, Parlog 処理系 [12] がある.)

これらの処理系の実行モデルでは, 実行可能なゴールの集合であるゴールプールがあり, プロセッサはその中からゴールを取り出し, 定義節のガード部を順に試行する. ガード部テストに成功した定義節があれば, そのボディ部を展開する. すなわち, ボディユニフィケーションを実行し, ボディゴールをゴールプールに入れる. ガード部テストは中断することがあり, 中断したゴールは必要なデータが揃ってからゴールプールに戻され, 再びスケジューリングの対象となる. ゴールプールからのゴールの取り出しはコンテキストスイッチに相当するが, ボディ部展開において, ボディゴールの内の 1 つはゴールプールに入れずにすぐ実行することで, その回数を減らすのが普通である. これは, Prolog 処理系の末尾再帰呼出し最適化に対応する.

上田 [51] に倣ってこの処理方式をプロセス指向処理方式と呼ぼう. 代表的な処理系の一つとして KL1 処

[12] これは GHC 言語 [49] の仕様である. 細部の違いはあるが, 他の並行論理型言語でも基本的に同様である.

理系における実装技術を具体的に見てみよう.

5.1.1 KL1 処理系

KL1 は GHC のフラットなサブセットである Flat GHC (FGHC) を, システム記述や並列推論マシン上での実行のために, タスク制御機能, 優先度・負荷分散指定機能などで拡張した言語である [50]. KL1 処理系は分散メモリ並列マシン向けに開発された処理系で, メッシュ状ネットワークのマルチ PSI 上で稼働しており, 幾種類かのアーキテクチャを持つ並列推論マシン PIM 向けの処理系も開発中である [18]. 例えば, PIM/p は共有バス型クラスタをハイパキューブで接続したアーキテクチャをしており, その上の KL1 処理系は, 分散メモリ並列マシン用処理系と共有メモリ並列マシン用処理系を組み合わせたものになっている [58].

KL1 処理系に採用された主な実装技法を紹介する.

(1) MRB 方式

並行論理型言語の典型的なプログラミング技法としてプロセス間のストリーム通信がある. 1 回の通信にコンスセル 1 つが割り付けられ, それが参照された後にゴミとしてたまって行くとすると, ガーベジコレクションが頻発することになろう.

多重参照ビット (MRB) [8] は, この問題のひとつの解決として考案された. この方式では, ポインタに MRB と呼ばれる 1 ビット情報を付加し, MRB=OFF の時に単一参照であることが保証されるように管理する. そして, あるプロセスがデータを読んで, 以後そのデータが不要になった際に, それが単一参照ならば, 以後そのデータは参照されないことが分かるので, ゴミとして回収する. 例えば, 1 対 1 のストリーム通信において送信側で割り付けられるコンスセルは受信側のみによる単一参照なので, 受信時に回収される.

また, プロセスがある配列の 1 エントリを書き換えて新しい配列を作る場合, 古い配列が単一参照であって, 以後使われないのなら, 新しい配列を古い配列の領域に割り付けることができ, 1 エントリを破壊的に書き換える処理で済んでしまう. 配列要素の定数時間更新がこれによって実現できたことになる. これは配列を扱うプログラムの計算オーダに関係するのでゴミの即時回収以上に重要な効果と言える.

(2) スケジューリング

KL1 処理系では, ボディ部をできるだけ左から右の順で実行しようとする. そのためにプロセッサ毎にゴールを LIFO キュー (ゴールスタックと呼ぶ) で管理する. これは, ボディ部内でのデータの流れは普通左から右に向かっており, そのようなゴール列を左から順に実行するとデータ待ちによるゴールの中断が起き難いからである. また, 探索の枝を異なる優先度でスケジュールできるように細かい優先度管理 (現処理系で 4096 レベル) を行なっている.

(3) 負荷分散

PIM/p 上処理系ではクラスタ内の暇なプロセッサが忙しいプロセッサのゴールスタックからゴールを奪うこと (タスクスチール) で, 自動負荷分散を行なうようにしている.

プロセッサ (PIM/p ではクラスタ) をまたがる負荷分散は, 自動的には行なわず, プログラマの指定に従う. このために, ボディゴール毎にそれを実行するプロセッサ (PIM/p ではクラスタ) を指定する (プロセッサ / クラスタ番号は実行時に決まればよい) 機能が用意されている.

(4) その他

この他に, プロセッサ間即時ガーベジコレクション方式 [28] やタスクの分散終了検出方式 [42] などで新しい技法が採用されている. また, レジスタ数の少ないマシンでは, ゴール呼出し時に全ての引数をレジスタ上に展開することができないので, ゴール引数を lazy にフェッチするメモリベース方式が採用されている [62].

5.2 メッセージ指向処理方式

プロセス指向処理方式は, 安定した本格的処理系が実装され, 一定水準の性能を達成し,[13] また, 中規模以上のプログラムを十分に動かせることを示した. しかし, この処理方式には並行論理型言語らしい中断の多い通信型のプログラムの性能が出しにくいという欠点がある. そこで通信型プログラムの性能を上げるようなメッセージ指向処理方式が提案された [51].

プロセス指向処理方式は, ゴールの親子関係の連鎖を, last-call 最適化 (末尾再帰呼出し最適化) によってコンテキストスイッチなしのスレッドとし, それをスケジュール単位にしていた. これに対し, メッセージ指向処理方式では, メッセージ通信による因果関係の連鎖を, last-send 最適化によってコンテキストスイッチなしのスレッドとし, それをスケジュール単位にしようとする.

並行論理型言語ではメッセージ通信はプリミティブではなく, ユニフィケーションを用いて実現されているので, プログラムの大域的解析によってメッセージ通信を同定する必要がある. このために, 精密なモード解析手法が開発された. これは, 引数単位の入出力

[13] おおよそ, append で 130 KLIPS, プロセス中断の頻繁なプログラムで 20 KLIPS, 平均的なプログラムで 40 KLIPS 程度である.

という粗いものでなく，正規表現でモードを表わしており，不完全メッセージも扱える．整合的なモードが定まるプログラムとして，モード付きFGHCというFGHCプログラムのサブセットが定義される．

メッセージ指向処理方式では，メッセージ通信を通常のユニフィケーションと区別できるので，送信側がコンスセルを組み立てることをせず，レジスタにメッセージを乗せて受信側を呼び出すという最適化も行なえる．また，プロセス指向処理方式がプロセススイッチを極力減らしてスループットを上げるために，メッセージを多量に生産した後にまとめて消費するようにスケジュールする (スループット指向) のに対し，メッセージ指向処理方式ではメッセージを生成すると直ちに消費するようにスケジュールされる (レスポンス指向) ので，後者ではアクティブなデータ量が少なくて済むという特徴もある．静的解析の副産物として，実行時のMRB管理オーバヘッドなしにMRB情報を使った最適化も可能となっている．

汎用Unixマシン上のハンドコンパイルによる初期評価によると，2進木のノードをKL1プロセスとして実現した場合，プロセス指向方式と比べて3倍程度の高速化が得られ，再帰呼出しを用いたCプログラムの処理時間と比べて2〜3倍程度にまで接近している．

最近設計された分散制約型言語Janus [43] においても静的情報を用いてコンパイラがメモリ領域の回収 / 再利用命令を出せるようにしている．ただしJanusではプログラマにストリーム通信を明示させることで，複雑な解析を省いている．実行時オーバヘッドにつながるような並行論理型言語の自由度を制限するアプローチはStrandにも共通している最近の傾向である．

5.3 SIMD型処理方式

Nilsson [40] とBarklund [5] は独立に，SIMD型並列マシンへのKL1処理方式を記述している．前者では各KL1ゴールが，また後者ではKL1プログラムを表現する条件グラフの各ノードが，同期並列に実行される．ある意味で，KL1の並列計算モデルのイメージをそのまま表現していると言える．しかし，素朴に実装すると，一旦中断したゴールに (仮想的) 要素プロセッサが割当てられたままになるので，プロセッサ有効稼働率が低くなるという問題があろう．

6 その他の並列実行モデル

本節では3, 4, 5節で取り上げなかった並列実行モデルについて簡単に触れる．

6.1 AND/OR並列実行

制限AND並列処理系は，一般に分割統治型のプログラムから高い並列度が取り出せるが，探索プログラムからは低い並列度しか取り出せない．一方，OR並列処理系は逆の性質を持つ．もし，一つの処理系でANDとOR両方の並列性を取り出せれば，より広い範囲のプログラムが並列実行によって高速化できると期待される．

AND/OR並列実行は，AND並列にOR並列がネストしたものとも，その逆とも見ることができる．前者のように見た場合，一つの節CのAND関係にあるゴールたち$G_1, \ldots, G_n$が並列に実行され，その際，その各々のゴールG_iの実行においてOR関係にある候補節たちが並列に試行される．そして，各ゴールのOR関係にある結果を整合性を取って組み合わせたものが，最初の節Cの実行結果となる．しかしながらこのやり方には，整合的組み合わせの処理のオーバヘッドがあるばかりでなく，逐次実行なら変数束縛が進んでいるために狭まっていた各の個々のゴール実行の探索空間が拡大してしまうという重大な問題がある．

Gupta と Jayaraman [20] は，独立AND関係にあるゴールのみを並列に実行し，各ゴールそれぞれをOR並列実行し，結果である解集合の直積を作る，という方式を提案している．このようなAND/OR並列実行の利点がよく発揮されるのは次のような例である．

```
get_pair(K1,K2,pair(R1,R2)) :-
  retrieve(K1,R1),
  retrieve(K2,R2),
  compatible(R1,R2).
```

というようなプログラムで，`retrieve(K1,R1)`および`retrieve(K2,R2)`がそれぞれm個，n個の解を持つ場合，OR並列実行でも2つの`retrieve`の独立AND並列実行でも，`retrieve(K1,R1)`のm回の成功それぞれに対して，`retrieve(K2,R2)`がn回呼び出される．それに対して，AND/OR並列実行では`retrieve(K2,R2)`は1回だけ呼び出され，そのn通りの解が`retrieve(K1,R1)`のm通りの解と組み合わされる．

AND並列実行される各ゴールから解が次々と返され，その解系列たちをjoinしていくというREDUCE-ORモデル [29] 等の方式も提案されている．

6.2 Andorraモデル

Andorraモデル [21] はWarrenの提唱する並列実行モデルで，基本的アイデアはRong YangがP-Prolog [56] で示している．Andorraモデルでは，AND関係にあるゴールを並列に実行するが，それぞれのゴールは非決定性のない限りにおいて前向き実行を続ける

ことができる. そして, 実行中に候補節を一意に選択する情報が不足するとゴールは中断する. 並列実行中の全てのゴールが中断すると, 実行中のゴールの最初のものをOR展開する. すなわち, 適用可能な複数の候補節に対応するOR分岐を作り, それぞれの枝でAND並列実行を再開する. したがって, OR木の分岐ノード間をAND並列実行で結んだ形となる.

組み合わせ的な制約問題解決では, 決定的な処理を先に進めることで, 問題に関する情報が増え, その結果, AND関係にあるゴールの非決定性 (候補の数) が減って, 探索空間が狭められることが多い. この情報は入力データに依存して様々な方向に流れ得るので, データフローに基づいて動的に実行スケジュールができるAndorraモデルは, そのような問題に適していると言える. そのような例として, 簡単な暗号解読プログラムの例が [57] に記されている.

Andorraモデルの変形がその後, 幾つか提案されている [3, 22]. 問題点は, 効率のよい処理方式がまだ見つかっていないことである.

6.3 探索並列

探索並列 (search parallelism) は Conery [11] の指摘した並列性の一つで, 非常に多数の候補節があるときに, それらを幾つかの集合に分割してゴールとのマッチングを並列に行なうというものである. 通常のプログラムでは節インデキシング手法によって効率良く候補節探索ができるので, この方向の並列処理はあまり研究されていない.

単位節 (unit clause) の集合は関係データベースと見ることができる. データベースへの問い合わせとしての論理プログラムの (並列) 処理については, 演繹データベースやデータベースの並列処理の分野の研究を参照して頂きたい.

6.4 並列ユニフィケーション

導出ステップではゴールと節のヘッドとの間でユニフィケーションが行なわれる. 高速なユニフィケーションは高速な処理系を作る上で不可欠であり, 並列処理による高速化は当然考えられることである. ところが, 実際の論理プログラムにおいてはユニフィケーションは粒度が小さ過ぎて並列化に向かない. 例えば, WAMでは節を解析することで大半のユニフィケーションを単なるデータのロード / ストアや定数との比較に落とすことに成功しており, それらについては並列化しても高速化は望めない.

このような訳で, 並列ユニフィケーションによる処理系高速化は余り研究されてこなかったが, 最近, データ並列型論理プログラムが研究されており [4], 新しい論理型プログラム並列実行パラダイムに発展するか注目される.

6.5 通信プリミティブを追加したモデル

これまでに述べた並列処理方式は, 論理型言語そのものの持つ自然な並列性に基づいたものだった. これに対し, 逐次論理型言語に通信プリミティブを追加して, 複数の逐次プロセスが通信 / 同期し合うようなプログラムが書けるようにするアプローチがある.

Delta-Prolog [41, 13] は, Prologにランデブー型の送信 / 受信プリミティブを付加した言語である. これらのプリミティブはバックトラックが起きると送受信の対で巻き戻しされる. このために1つのプロセスにおけるバックトラックが送受信プリミティブを通じて他のプロセスに伝播する. この分散バックトラック (distributed backtracking) は後向き実行を複雑にし, 効率の良い実装を困難にしている.

Shared Prolog [2] では, 並列に走るProlog逐次プロセスが黒板 (共有ファクトデータベース) を介して通信し合う. Linda [17] の一つのインスタンスProlog-Lindaと考えることができる.

7 おわりに

以上, 論理型言語の並列処理方式について述べてきた. 並列処理方式の種類の多いことに今更のように驚くが, 論理型プログラムが複数の側面での並列性を持つことや, 高い並列度の抽出と処理効率オーバヘッド低減の間に様々なトレードオフがあり得ることが理由であろう.

Prologの並列実行方式では, 比較対象となる逐次処理系においてWAMという効率のよい処理方式が確立していたために並列実行によるオーバヘッドの低減が目標とされ, それはOR並列方式においても制限AND並列方式においても満足すべきレベルに近づいたと言えよう.

並行論理型言語の処理方式は, プロセス指向の第一世代の処理系が一定水準の性能を達成して, 中大規模のプログラム開発を可能にした. 通信の多いプログラムの性能を上げるメッセージ指向の処理方式も提案されており, プログラムの詳細な大域解析のできるコンパイラが開発されると, まとまった処理系にまで発展するであろう.

しかし, まだ未解決の課題も多い. Prologの並列実行方式について言えば, 副作用やカットが処理に逐次性を強制してしまう問題がある. これらはPrologにおけるメタ論理的あるいは非論理的な要素であり, 並列実行に際して処理系の側からも問題点としてクローズアップされて来た訳である. 本稿では十分に取り上

げられなかったが, 効率の良いメモリ管理も課題として残されている. 並列実行においては, メモリ領域のスタック管理が複雑になったり, ヒープ管理をしなければならなくなる. また, そのために不要になったメモリ領域が効率良く回収できず, ガーベジコレクション (GC) 性能の処理系の全体性能に与える影響が大きくなる. メモリ管理の通常実行部の性能向上につれて, この問題はより深刻になろう.

小規模共有メモリ型並列計算機での方式の実証がほぼ成功した後, より大規模な並列計算機での実装も次なる課題となって行くであろう. この分野の更なる発展を願って本稿を終えたい.

参考文献

[1] Ali, K. A. M. and Karlsson, R.: The Muse or-parallel Prolog model and its performace, In *Proceedings of NACLP'90*, pp. 757–776 (1990).

[2] Ambriola, V., Ciancarini, P., and Danelutto, M.: Design and distributed implementation of the parallel logic language Shared Prolog, In *PPoPP'90*, pp. 40–49 (1990).

[3] Baghat, R. and Gregory, S.: Pandora: Nondeterministic parallel logic programming, In *Proceedings of ICLP'89*, pp. 471–486 (1989).

[4] Barklund, J.: *Parallel Unification* PhD thesis, UPMAIL, Computing Science Department, Upsala University, (1990).

[5] Barklund, J., Hager, N., and Wafin, M.: KL1 in Condition Graphs on a Connection Machine, In *Proceedings of FGCS'88*, pp. 1041–1050 (1988).

[6] Baron, U., "Chassin de Kergommeaux, J., Hailperin, M., et al.: The parallel ECRC Prolog system PEPSys: An overview and evaluation results, In *Proceedings of FGCS'88*, pp. 841–850 (1988).

[7] Borgwardt, P.: Parallel Prolog using stack segments on shared-memory multiprocessors, In *Proceedings of SLP'84*, pp. 2–11 (1984).

[8] Chikayama, T. and Kimura, Y.: Multiple reference management in Flat GHC, In *Proceedings of ICLP'87*, pp. 276–293 (1987).

[9] Clark, K. L. and Gregory, S.: PARLOG: Parallel programming in logic, *ACM Transactions on Programming Languages and Systems* Vol. 8, No. 1, pp. 1–49 (1986).

[10] Clocksin, W.: Principles of the DelPhi parallel inference machine, *Computer Journal* Vol. 30, No. 5, pp. 386–392 (1987).

[11] Conery, J. S. and Kibler, D. F.: Parallel interpretation of logic programs, In *Proceedings of Functional Programming Languages and Computer Architectures*, pp. 163–170 (1981).

[12] Crammond, J.: The abstract machine and implementation of parallel Parlog, Research report, Department of Computing, Imperial College (1990).

[13] Cunha, J. C., Ferreira, M. C., and Pereira, L. M.: Programming in Delta Prolog, In *Proceedings of ICLP'89*, pp. 487–502 (1989).

[14] DeGroot, D.: Restricted AND-parallelism, In *Proceedings of FGCS'84*, pp. 471–478 (1984).

[15] Foster, I. and Taylor, S.: Strand: A practical parallel programming tool, In *Proceedings of NACLP'89*, pp. 497–512 (1989).

[16] Foster, I. T. and Taylor, S.: Flat Parlog: A basis for comparison, *International Journal of Parallel Programming* Vol. 16, No. 2, (1988).

[17] Gelernter, D.: Generative communication in linda, *ACM Transactions on Programming Languages and Systems* Vol. 7, No. 1 (1985).

[18] Goto, A., Sato, M., Nakajima, K., Taki, K., and Matsumoto, A.: Overview of the parallel inference machine (PIM) architecture, In *Proceedings of FGCS'88*, pp. 208–229 (1988).

[19] Gupta, G. and Jayaraman, B.: On criteria for or-parallel execution models of logic programs, In *Proceedings of NACLP'90*, pp. 737–756 (1990).

[20] Gupta, G. and Jayaraman, B.: Optimizing and-or parallel implementations, In *Proceedings of NACLP'90*, pp. 605–623 (1990).

[21] Haridi, S. and Brand, P.: ANDORRA Prolog – an integration of Prolog and committed choice languages, In *Proceedings of FGCS'88*, pp. 745–754 (1988).

[22] Haridi, S. and Janson, S.: Kernel Andorra Prolog and its computation model, In *Proceedings of ICLP'90*, pp. 31–46 (1990).

[23] Hausman, B. et al.: Or-parallel Prolog made efficient on shared memory multiprocessors, In *Proceedings of SLP'87* (1987).

[24] Hausmann, B. and Ciepielewski, A.: Cut and side-effects in or-parallel prolog, In *Proceedings of FGCS'88*, pp. 831–840 (1988).

[25] Hermenegildo, M. V.: &-Prolog and its performance: Exploiting independent and-parallelism, In *Proceedings of ICLP'90*, pp. 253–268 (1990).

[26] Hermenegildo, M.: Non-strict independent and-parallelism, In *Proceedings of ICLP'90*, pp. 237–252 (1990).

[27] Hermenegildo, M. V.: An abstract machine for restricted AND-parallel execution of logic programs, In *Proceedings of ICLP'86*, pp. 25–54 (1986).

[28] Ichiyoshi, N., Rokusawa, K., Nakajima, K., and Inamura, Y.: A new external reference management and distributed unification for KL1, In *Proceedings of FGCS'88*, pp. 904–913 (1988).

[29] Kalé, L. V.: The REDUCE-OR process model for parallel evaluation of logic programs, In *Proceedings of ICLP'87*, pp. 616–632 (1987).

[30] Kumon, K., Masuzawa, H., Itashiki, A., Satoh, K., and Sohma, Y.: Kabu-Wake: A new parallel inference method and its evaluation, In *Proceedings of CompCon Spring '86*, pp. 168–172 (1986).

[31] Lin, Y.-J. and Kumar, V.: Performance of and-parallel execution of logic programs on a shared memory multiprocessor, In *Proceedings of FGCS'88*, pp. 851–860 (1988).

[32] Lin, Y.-J., Kumar, V., and Leung, C.: An intelligent backtracking algorithm for parallel execution of logic programs, In *Proceedings of ICLP'86*, pp. 55–68 (1986).

[33] Lloyd, J. W.: *Foundations of Logic Programming* Springer-Verlag, Berlin, second, extended edition (1987).

[34] Lusk, E., Warren, D. H. D., Haridi, S., et al.: The Aurora OR-parallel Prolog system, In *Proceedings of FGCS'88*, pp. 819–830 (1988).

[35] Muthukumar, K. and Hermenegildo, M. V.: The CDG, UDG, and MEL methods for automatic compile-time parallelization of logic programs for independent and-parallelism, In *Proceedings of ICLP'90*, pp. 222–236 (1990).

[36] Muthukumar, K. and Hermenigildo, M.: Complete and efficient methods for supporting side-effects in independent/restricted and-parallelism, In *Proceedings of ICLP'89*, pp. 80–97 (1989).

[37] Naish, L.: Automating control for logic programs, *Journal of Logic Programming* Vol. 2, No. 3, pp. 167–183, (1985).

[38] Naish, L.: Parallelizing NU-Prolog, In *Proceedings of ICLP'88*, pp. 1546–1565 (1988).

[39] Nakajima, K., Inamura, Y., Ichiyoshi, N., Rokusawa, K., and Chikayama, T.: Distributed implementation of KL1 on the Multi-PSI/V2, In *Proceedings of ICLP'89*, pp. 436–451 (1989).

[40] Nilsson, M. and Tanaka, H.: Massively parallel implementation of Flat GHC on the Connection Machine, In *Proceedings of FGCS'88*, pp. 1031–1040 (1988).

[41] Pereira, L. M. and Nasr, R.: Delta-Prolog: A distributed logic programming language, In *Proceedings of FGCS'84*, pp. 283–291 (1984).

[42] Rokusawa, K., Ichiyoshi, N., Chikayama, T., and Nakashima, H.: An efficient termination detection and abortion algorithm for distributed processing systems, In *Proceedings of ICPP'88, Vol. I Architecture*, pp. 18–22, (1988).

[43] Saraswat, V. A., Kahn, K., and Levy, J.: Janus: A step towards distributed constraint programming, In *Proceedings of NACLP'90*, pp. 431–446 (1990).

[44] Shapiro, E. Y.: A subset of Concurrent Prolog and its interpreter, In Shapiro, E. Y., editor, *Concurrent Prolog: Collected Papers, Vol. 1*, chapter 2, pp. 27–83 The MIT Press (1987).

[45] Shapiro, E. Y.: Or-parallel Prolog in Flat Concurrent Prolog, *J. Logic Programming* Vol. 6, No. 3, pp. 243–267 (1989).

[46] Somogyi, Z., Ramamohanarao, K., and Vaghani, J.: A backtracking algorithm for the stream AND-parallel execution of logic programs, In *Proceedings of ICLP'88*, pp. 1142–1159 (1988).

[47] Taylor, S., Safra, S., and Shapiro, E.: A parallel implementation of Flat Concurrent Prolog, *International Journal of Parallel Programming* Vol. 15, No. 3, pp. 245–275 (1987).

[48] Tinker, P. and Lindstrom, G.: A performance-oriented design for or-parallel logic programming, In *Proceedings of ICLP'87*, pp. 601 615 (1987).

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

[50] Ueda, K. and Chikayama, T.: Design of the kernel language for the parallel inference machine, *Computer Journal* Vol. 33, No. 6, pp. 494–500 (1990).

[51] Ueda, K. and Morita, M.: A new implementation technique for Flat GHC, In *Proceedings of ICLP'90*, pp. 3–17 (1990).

[52] Warren, D. H. D.: Or-parallel execution models of Prolog, In *Proceedings of TAPSOFT'87*, pp. 243–259 (1987).

[53] Warren, D. H. D.: The SRI model for OR-parallel execution of Prolog – abstract design and implementation issues, In *Proceedings of SLP'87*, pp. 92–102 (1987).

[54] Warren, D. S.: Efficient Prolog memory management for flexible control strategies, In *Proceedings of SLP'84*, pp. 198–202 (1984).

[55] Westphal, H., Robert, P., "Chassin de Kergommeaux, J., and Syre, J.-C.: The PEPSys model: Combining backtracking, AND- and OR- parallelism, In *Proceedings of SLP'87*, pp. 436–448 (1987).

[56] Yang, R.: *A Parallel Logic Programming Language and its Implementation* PhD thesis, Keio University (1986).

[57] Yang, R.: Solving simple substitution ciphers in Andorra-I, In *Proceedings of ICLP'89*, pp. 113–128 (1989).

[58] 今井明, 後藤厚宏, 堂前慶之: 共有メモリマルチプロセッサにおける KL1 言語の並列実行方式 — 負荷分散とユニフィケーション —, 情報処理学会研究報告 *CPSY 90-48* (1990).

[59] 金田悠紀, 松田秀雄: 逐次型推論マシンのアーキテクチャ, 情報処理 Vol. 32, No. 4 (1991).

[60] 後藤厚宏: 並列型推論マシンのアーキテクチャ, 情報処理 Vol. 32, No. 4 (1991).

[61] 田中英彦: 論理型言語指向の推論マシンの位置付けと開発の現状, 情報処理 Vol. 32, No. 4 (1991).

[62] 平野喜芳, 後藤厚宏: 並列論理型言語 KL1 のコンパイル方式の改良, 並列処理シンポジウム *JSPP'90* 論文集, pp. 281–288 (1990).

[63] 横田実: 論理型言語の逐次処理方式, 情報処理 Vol. 32, No. 4 (1991).