ICOT Technical Report: TR-569

TR-569

Proceeding of KL1 Programming Workshop '90

瀧　和男，古川康一

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 Programming Workshop '90

Workshop on
Concurrent Programming and Parallel Processing

May 14 – 16, 1990
ICOT

序

第五世代コンピュータプロジェクトが始まったのは，1982年であるが，その計画作りは，その3年前の1979年から始められた．このプロジェクトの大きな特徴は，それ以前のコンピュータプロジェクトと異なり，ソフトウエア，あるいはプログラミング言語を重視した点である．そして，我々が選んだソフトウエアのパラダイムが，論理プログラミングであり，その言語の一つがPrologであった．Prologは，1970年代の初頭にフランスのマルセイユで生まれたプログラミング言語である事は，周知の事実であるが，本プロジェクトが始まる直前の1982年3月に，筑波において，Prologコンファレンスが開催された．当初，30名ぐらいの出席者を想定していたが，約80名に及ぶ参加者があり，Prologに関する活発な技術討論が展開された．現在，Prologは，解説書も出回り処理系も流布しているが，当時は多分そこに参加した研究者以外には，国内でPrologを知る人は，ほとんどいなかったのではないかと思われる．

今日の，GHC/KL1の状況は，丁度，当時のPrologの置かれた状況そのもののような気がする．GHC/KL1の前身であるRelational Languageは，Prologに遅れること約10年，1980年代初頭にロンドン，インペリアルカレッジで生まれた．それから約十年して，KL1プログラミングワークショップが開催される運びとなった．

KL1は，Prologと同様，論理プログラミング言語の一種であるが，その性質は全く異なる．それは，知識処理用の記述言語というよりも，並列処理の汎用高水準言語と捉える事ができるであろう．KL1プログラミング技術の開発は，始まったばかりである．他の言語の例を見るまでもないが，一つの言語が本格的に普及するまでには，10年から15年の年月を要するであろう．KL1は，その揺籃期が既に経過し，これから本格的な成育期に入るところである．本ワークショップが，KL1プログラミングのこれからの発展を大いに加速する事を期待したい．

KL1プログラミングワークショップ
プログラム委員長　　古川 康一

本ワークショップの目指すもの

KL1でプログラムを書こうとしている人たちの抱えている課題は、ほとんどICOTの研究課題そのものである。

ICOTの目標は、御存じの通り、将来の大規模知識情報処理のための基礎技術を研究開発することである。研究開発の枠組は、簡単化していうならば、図1に示すように、マシンを賢くするための「知識処理技術」の開発と、マシンを早くするための「並列処理技術」の開発からなる。そしてこの両者を研究するための基礎であるとともに、それらを統一的に取り扱うための枠組がロジックプログラミングである。この考え方は、ICOTの設立当初から変わらず受け継がれている。

従って、ロジックプログラミング言語は、プロジェクトの中で重要な位置を占め続けてきた。プロジェクトの最初から使用されてきた逐次実行のProlog、そしてPSI上のESPは、一種の推論メカニズム或いは知識記述パラダイムを提供することから、知識処理寄りの記述言語であったといえるだろうし、それだけ知識処理技術の研究者に多くの恩恵を与えてきたともいえよう。一方、プロジェクトの中期になって、並列処理研究の活発化とともに育ってきたGHC/KL1は、バックトラックを伴う自動探索の機能を捨てる代わりに、並列処理のために極めて重要となる諸機能を取り込んできた。すなわち図1の枠組の中では、並列処理寄りの記述言語といえる。

これは並列処理研究の、予想を越える(一部の人にとっては予想通りの)難しさのために、必然的に生まれた結果ともいえるものであり、並列処理技術の研究者にとっては大きな恩恵を与え始めている。反対に、知識処理研究にとってのGHC/KL1は、知識記述パラダイムの観点からは、より無色透明であるがゆえに、利用者にとって必ずしも満足のゆかないものともいえよう。この意味においては、もう少し色つきの、知識記述パラダイムを提供する並列論理型言語の開発が待たれる。

しかしながら、知識処理プログラムの本格的なものは、しばしば多量の計算を必要とするはずである。プロトタイピングには逐次型のESPなりを使うにしても、計算時間が掛かり過ぎる局面では、百歩譲って、並列型のKL1で記述していただきたいし、また実用規模のデータでテストしようとする限り、並列化は避けて通れない場合が、数多く現れてくるだろう。

実際、マルチPSI上のKL1の性能は、単一プロセッサ当たりでPSI-IIのESPの3分の1であり、64プロセッサ持ってきたとしても、最大でPSI-IIの性能の21倍、並列

化のオーバーヘッドを考えると10倍少しということになる。苦労して並列化して、PSI-IIの10倍早いだけでは、努力の甲斐がないといえるかも知れない。しかしながら、PIMになると、プロセッサ1台の性能でマルチPSIの3から4倍、プロセッサ数で最大8倍だから、PSI-IIに比べて240倍から320倍以上の性能が期待できる。このくらい高性能になると、さすがに有難みがでてくるであろう。

かくして、「知識処理を並列に書く」という難事業に突入することとなる。この難しさについては、実感のない人も多かろうが、理屈は簡単である。知識処理はもともとどうやって解いたら良いか分からない要素を多分に持っている。それに対して、どうやれば効率良く実行できるか未だ研究途上の並列処理を抱き合わせにするのだから、難しいことが重なって、しばしば訳が分からなくなる。

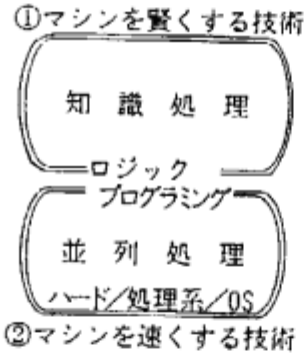


図1: 第五世代コンピュータ研究開発の枠組

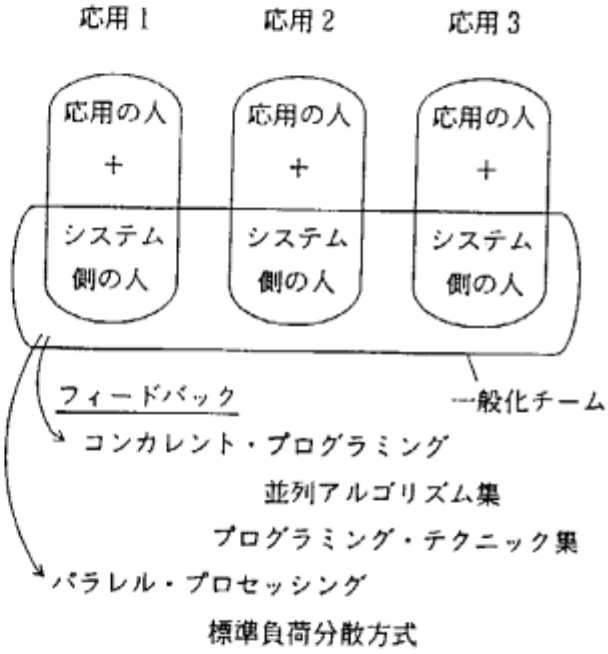


図2: 並列ソフトウェア研究開発のためのチーム編成方法

そこで並列化を始めようという人には、次のようなアドバイスをしている。少なくとも解こうとする問題が良く分析されていること、できれば逐次実行の場合の良いアルゴリズムが確定しているような問題を取り上げること。このような問題を並列化する場合は、プログラムが論理的に持っている並列実行可能性を実際に並列マシンにどう割り付けて並列実行させたら効率が良いか、即ち負荷分散の方針をどうすべきかについて、並列処理の専門家のアドバイスを受け易い。問題またはアルゴリズムがよくモデル化されているので、専門外の人でも理解し易いわけである。

問題なのは、使っていたアルゴリズムが本質的に逐次性を持っている場合である。これはアルゴリズムを変えない限り、並列実行の可能性が低い。このような場合は、仮

に並列言語に書き換えて並列マシンに割り付けたとしても、たくさんのプロセッサはかわるがわる順番に動くだけで、決して並列処理にはならない。

そこで、もっと並列性が出るようにアルゴリズムを変えようとする。これはしばしば、問題の解法自体を変更していることに当たる。そうするとこんどは、得られる解の品質が変わったり、不必要な見込み計算がふえる場合がある。このような場合には特に、負荷分散方式を少し手直しするだけで、解の品質や見込み計算量が大きく影響を受け易いわけで、並列処理の専門家はきわめて手を下しにくくなる。即ち負荷分散を調整して並列性は向上したが、解は悪くなったり計算量が増えて実行時間は伸びたりということが起こり得る。この場合、負荷分散の調整が悪かったのか、アルゴリズムの変更の仕方が悪かったのか、よく注意しないと区別がつかないことになる。

こうなると即ち、知識処理の難しさと並列処理の難しさが混じり合って、訳が分からなくなった状態といえる。もう少しきちんというと、問題解法のアルゴリズムと、負荷分散のアルゴリズムの相互依存度が増し、各々を独立に扱えなくなった状態ともいえる。

このような困った状態は、逐次性の高いアルゴリズムに並列性を入れようとする場合のほかに、データの集中管理を必要とするアルゴリズムを分散アルゴリズムに焼き直そうとする場合 (例えば黒板モデルの並列化など) にも、しばしば発生する。

このようなときに重要なのは、知識処理の専門家と並列処理の専門家の単なる協力ではない。知識処理の専門家であり解こうとする問題領域のことをよく知った人が、並列処理についてもよく勉強しその勘どころをわきまえた上で、アルゴリズムの修正に取り組まねばならない。或いは、並列処理の専門家が、問題領域のことを十分勉強して、という方向のアプローチもあろう。しかしながら、これらはそう簡単なことではない。

そこで、並列化にあたって問題解法と負荷分散方法が依存関係を持ってしまうような問題を扱う時の、研究開発の進め方として、図2に示すような形態を奨励しており、実践も始めている。即ち、開発すべき応用問題毎に、問題領域のプログラム開発を手掛けてきた専門家 (応用の人) と、並列処理システム開発の専門家 (システム側の人) のジョイントチームを作り、専門技術の相互の勉強を促進しつつ開発に取り組む。これらは研究開発の縦糸である。このようなジョイントチームを応用問題毎に置き、一方で、個々の研究開発の中で生まれた技術上の工夫、即ち負荷分散方式やプログラミングテクニックや並列アルゴリズム上の工夫を相互に伝え合い、さらに一般化してゆく一般化チーム (研究開発の横糸) を設置する。そして、たとえば標準的な負荷分散方式をOS でサポートし記述言語にも取り込むようフィードバックをかけたり、あるいはプログラミングテクニック集を作って普及させたりといった活動をおこなう。

初期のKL1 プログラム開発で、このようなジョイント研究開発チームの形態をとり、それなりに成功した例としては、マルチPSI のデモプログラムとしてFGCS'88 の時

から使われている、詰め碁、および自然言語構文解析プログラムPAXの開発があげられる。それぞれ囲碁プログラム開発の経験者、自然言語処理プログラム開発の経験者と、KL1言語処理系やPIMOSやマルチPSI開発の経験者のジョイントによる。また横糸に当たる一般化チームとしては、同様にICOTの言語処理系やOSやハード開発者から組織されるKL1プログラミングTGと呼ばれるグループがその役割を担ってきた。このような活動は、今後さらに活発化させてゆこうと計画しており、また各社の応用開発グループとのジョイントチームも検討するべき時期と考えている。

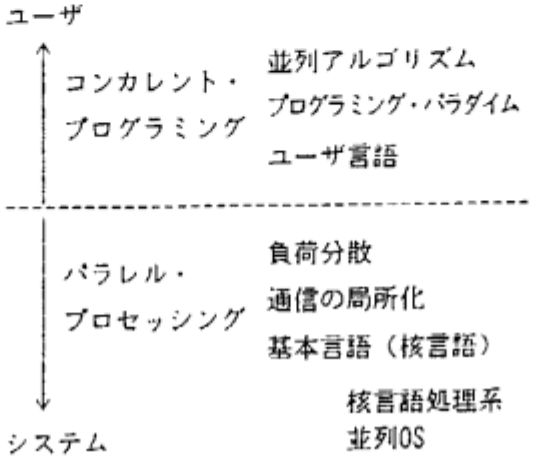


図 3: 並列ソフトウェアの研究テーマ＝本ワークショップのテーマ

今回のワークショップは、先に述べた横糸チームの行うべき活動を拡大したものとお考えいただきたい。即ち情報交換をすべきテーマとして、図3に示すようなものがあげられる。

まずユーザのプログラミングに関わりの大きい技術、即ち問題の解法を考え並列プログラム化する辺りまでの技術として、各種アルゴリズムの並列化、プログラミング技法や問題のモデル化技法を含むプログラミングパラダイム、ユーザ言語あるいは特定問題領域向き記述言語、などに関する技術交流がある。これらをコンカレント・プログラミングの技術と呼ぼう。この範疇で扱う並列は、プログラムが論理的に持つ並列性(並列実行の可能性といってもよい)までである。

もう一方は、ユーザプログラムが論理的に持っている並列実行可能性を実際の並列マシンにどのようにマッピングし、効率良く並列実行させるかに関わる技術である。これをパラレル・プロセッシングの技術と呼ぼう。この中には、負荷の均等分散、通信の局所化、言語やOSへの反映などが含まれる。これらは本来は、ユーザが心配しなくても、システムが勝手に、最良の状態にコントロールしてくれると有難い話であるが、実はまだどうすればうまくコントロールできるか分かっていないことの方が多く、問題対応でプログラマが工夫しなければいけない状態である。

これらの個々の技術について、もう少し説明が必要だし伝えるべきことが蓄積されつつあるが、それはワークショップの中で明らかにされるだろうし、私自身もまた別稿にまとめたい。

他かに本ワークショップで議論すべき項目として、並列プログラムの評価方法、プログラム開発環境やデバッグ方式、現行の OS や言語処理系への改善要求なども重要であり、またどこでどのような並列プログラミングの活動が行われているか、そしてどのような苦労や工夫が重ねられているかを互いに知り合うことも、意義深いことと考える次第である。

最後に蛇足ながら、本書の表紙のロゴマークは、本ワークショップのキーワードともいうべき Concurrent Programming と Parallel Processing の頭文字を CP^3 と略して図案化したものである。

知識処理と並列処理は、第五世代マシン実現のための、開発途上の、二大基礎技術である。KL1 でいま並列プログラムを書こうとしている多くの人々は、これら二大技術の接点で、または二大技術の交錯する混沌の中で、自らの研究航路を切り開いてゆく勇者たちに違いない。

KL1 プログラミングワークショップ
プログラム委員会幹事　　瀧 和男

プログラム委員会

プログラム委員長
古川 康一

幹事
瀧 和男

委員

市吉 伸行	上田 和紀	木村 宏一
久保 幸弘	後藤 厚宏	谷口 尚
近山 隆	寺崎 智	西ヶ谷 茂
新田 克己	David Hawley	星田 昌紀
六沢 一昭		

KL1 Programming Workshop '90

プログラム

日時: 平成 2 年 5 月 14 日 (月) 9:30 ～ 17:30 (チュートリアル)
5 月 15 日 (火) 9:50 ～ 21:00
5 月 16 日 (水) 10:00 ～ 17:35

会場: (財) 新世代コンピュータ技術開発機構 アネックス会議室
(チュートリアル－入門・初級編のみ 212 会議室)
〒 108 東京都港区三田 1 丁目 4 番 28 号 三田国際ビル
電話: 03-456-3193 (ICOT 21F), 03-769-1298 (ICOT アネックス)

主催: (財) 新世代コンピュータ技術開発機構

5月14日(月) チュートリアル(KL1講習会)

Track 1

9:30~12:30 入門編
13:30~17:30 初級編

Track 2

9:30~12:30 中級編
13:30~17:30 ハッカー編

5月15日(火)

1 9:50~10:10 『開会』 古川康一, 瀧和男

2 10:10~12:10 『午前の部』 座長: 瀧和男

12:10~13:30 昼食休憩

3 13:30~14:50 『午後の部(その1)』 座長: 上田和紀

14:50~15:20 休憩

4 15:20~17:30 『午後の部 (その 2)』 座長: 近山隆

休憩 (10 分)

4.6 『KL1 をこうしてほしい』
コメントのある人全員

17:30~18:00 休憩

5 18:00~21:00 『夜の部』

18:00~18:30 食事と懇談
18:30~20:00 パネル討論
『私は KL1 プログラミングを楽しんでいるか』
— KL1 言語 / 処理系は使い物になるか
— 私は並列化のここに困っている
座長: 瀧, パネリスト: 渡辺 (日立), 長谷川, 高橋 (三菱), 古関 (日電), 近山
20:00~21:00 個別コンサルティング (悩みの相談コーナー)

5 月 16 日 (水)

6 10:00~12:00 『午前の部』 座長: 市吉伸行

12:00~13:30 昼食休憩

14:50~15:20 休憩

休憩 (10 分)

9 17:30~17:35 『閉会の辞』 渕一博

疎結合並列計算機上での OR 並列問題に適した 動的負荷分散方式とその評価

古市 昌一
三菱電機株式会社
情報電子研究所

瀧 和男，市吉 伸行
(財) 新世代コンピュータ技術開発機構

概要

疎結合並列計算機上での OR 並列型全解探索問題に適した動的負荷分散方式とその評価について述べる．負荷分散で一番重要なのは，負荷の均等化である．密結合並列計算機上ではこれまでに幾つかの動的負荷均等化方式が成功しているが，疎結合並列計算機上ではプロセッサ間の通信がオーバヘッドとなるために，動的な負荷の均等化は難しい．ここでは，OR 並列型全解探索問題一般に適用可能な動的負荷分散方式を提案する．本方式は，動的に検出した暇なプロセッサに対して仕事を割り付けるものである．また，プロセッサをグループ化してプロセッサグループに対する負荷分散とグループ内での負荷分散を行なう事により，階層的に負荷の均等化を行ない，更に階層を増やす事も可能なため，プロセッサの台数拡張性が高い．本方式を ICOT で開発した並列推論マシン・マルチ PSI/V2 上に実現し，詰込みパズルの全解探索問題に適用して評価を行なったところ，32 台プロセッサで 28.4 倍，64 台で 50 倍の台数効果が得られた．

1 はじめに

並列計算機の性能を最大限に引き出すためには，負荷の均等化が最も重要である．負荷の均等化は，与えられたプログラムを互いに独立なサブタスクに分割し，各要素プロセッサ (PE) の負荷が均等となるようにそれらを割り付けることで行なわれる．負荷の均等化方式は，様々な分野でこれまでに多くの研究がなされており [1, 2, 4, 7, 11, 14]，特に密結合並列計算機においては，幾つかの動的負荷均等化方式が成功している [7]．しかし疎結合並列計算機では，プロセッサ間通信のコストが高いために動的な負荷の均等化がより難しい．

数値演算を主体とする応用プログラムにおいては，タスクの粒度やタスク間の依存関係，及び通信のパターンが実行前に予測可能な場合が多く，負荷の均等化を静的に行なう事も可能である．しかし，多くのプログラムではこれらが不可能な場合が一般的であり，動的な負荷の均等化が不可欠である．疎結合並列計算機における動的負荷均等化方式の成功例はこれまで幾つか紹介されているが [5, 6]，個々のプログラム毎に作られており，また台数の拡張性はあまり考慮されていない．

本論文は，OR 並列型探索問題一般に適用可能な動的負荷均等化方式 (動的負荷分散方式) について述べたものである．本方式はプロセッサをグループ化して，グループのレベルとプロセッサのレベルで動的に負荷の均等化を行なうもので，多階層構造のために台数拡張性がある．本方式を疎結合並列計算機マルチ PSI/V2 上で詰込みパズルの全解探索問題に適用したところ，ほぼ線形的な台数効果が得られた．

以下，第 2 章では要求駆動型動的負荷分散方式について述べ，第 3 章では本論文で提案するマルチレベル動的負荷分散方式について述べる．第 4 章では性能の測定結果とその評価を行ない，第 5 章では負荷分散を行なう際に要求されるタスクの粒度に関する一般的議論を行ない，測定結果の解析を行なうとともに，本方式を他の応用プログラムに適用する方法について述べる．最後に，第 6 章にてまとめを行なう．

2 要求駆動型動的負荷分散方式

本章では，簡単な動的負荷分散方式について述べる．負荷分散は，プログラムを互いに独立なサブタスクに分割（サブタスクの生成）し，各 PE の負荷が均等化するようにサブタスクを PE に割り付ける（サブタスクの割り付け）ことによって行われる．

2.1 サブタスクの生成

サブタスクの生成は，特定の PE (マスター PE と呼ぶ) 上で行ない，その様子を図 1 に示す．ここで，大きな三角形は OR 木で表わされる探索空間を示す．縦棒で塗り潰された小さな三角形はサブタスクの生成を行なうサブタスクジェネレータを示し，探索の深さがあるレベル (負荷分散 (開始) レベル) に達するまでマスター PE 上で実行される．円は生成されたサブタスクを示し，その大きさが粒度を示す．従って，サブタスクの粒度は負荷分散レベルを深くするに従って小さくなる．ただし，

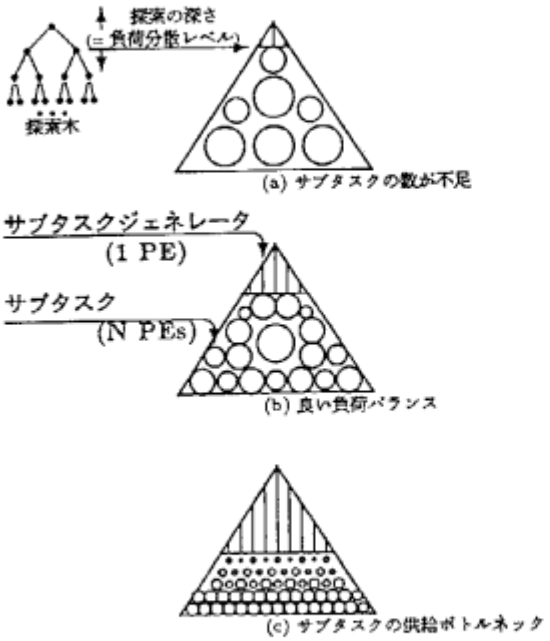


図 1: サブタスクの生成

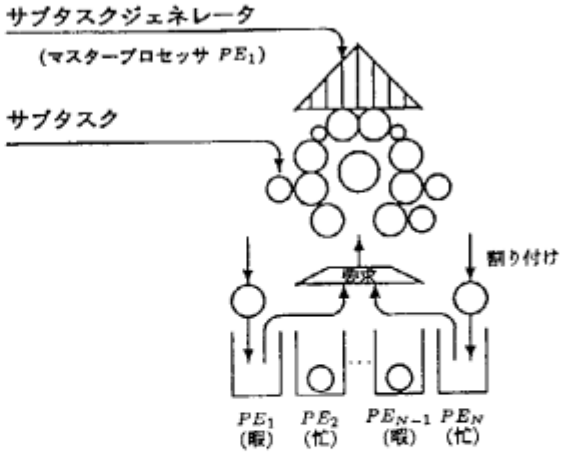


図 2: 要求駆動による動的負荷分散

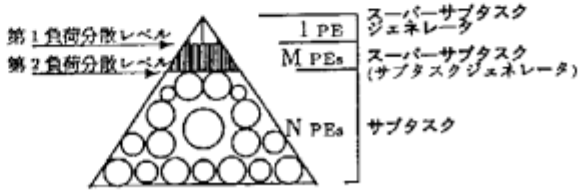


図 3: 二段階のサブタスク生成

粒度にはばらつきがあるものと仮定する．各サブタスクは，負荷を均等化するための戦略に従ってプロセッサに割り付けられる．

サブタスクの粒度には，互いに反する 2 つの条件が要求される．一方では，多くの PE を全て忙しく働かせるためにサブタスクの数は多くなければならない．他方，負荷分散の手間と比較して充分粒度を大きくしなければ，負荷分散オーバヘッドにより性能が抑えられる．例えば，図 1(a) の場合は負荷分散のオーバヘッドは小さいが，サブタスクの数が少なく粒度のばらつきにより負荷は均等化されない．また，図 1(c) の場合はサブタスクの数が多過ぎて，サブタスクの供給がボトルネックとなる．従って良い台数効果を得るためには，図 1(b) に示されるように，チューニングによって最適な負荷分散レベルを見つけねばならない．

2.2 サブタスクの割り付け

2.1のようにして生成したサブタスクは，各 PE の負荷が均等化するように暇な PE に割り付けられる．これは，ある PE が暇になるとマスター PE に対して新たなサブタスクの割り付けを要求するメッセージを送る事によって行なわれるので，要求駆動による動的負荷分散と呼ばれる．図 2はその方式を示したものであり，詳細は次の通りである．

1. 初期状態では全 PE は暇であり，マスター PE に要求メッセージを送る．
2. マスター PE は暇な PE にサブタスクを割り付ける．
3. 各 PE は，サブタスクの実行を終了すると新たなサブタスクを要求するメッセージを送る．

PE が暇になってからサブタスクが割り付けられるまでの間には遅延が生ずるが，これがサブタスクの平均実行時間と比較して無視できない場合には，サブタスクのダブルバッファリングを行なえば良い．

3 マルチレベル動的負荷分散方式

3.1 サブタスク生成ボトルネック

前章で示した要求駆動方式による動的負荷分散方式には，台数の拡張性がないという問題点がある．即ち，プロセッサの台数が増えるに従って単位時間当たりに解かれるサブタスクの数が，単位時間当たりに生成及び供給するサブタスクの数を越えた時，サブタスクの生成がボトルネックとなる．

3.2 マルチレベル動的負荷分散方式

このボトルネックを解消するためには，サブタスクの生成を行なうプロセッサの数を増やせばよい．そのために，サブタスクの生成を図 3に示すように二段階で行ない，負荷分散を図 4に示すように行なうマルチレベル動的負荷分散方式を提案する．

まず，スーパーサブタスクジェネレータはマスター PE に割り付けられ，探索が第 1 負荷分散レベルに達す

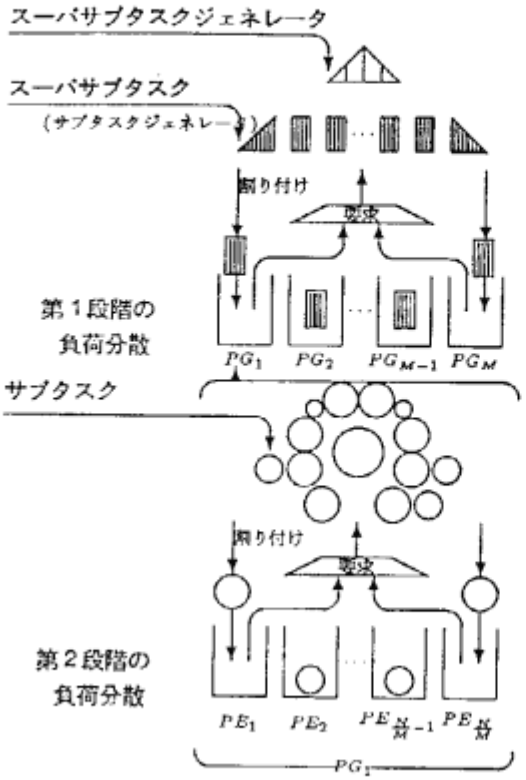


図 4: 二段階の動的負荷分散

るまでタスクをスーパーサブタスクに分割する．これを第一段階の負荷分散と呼ぶ．ここで，N 台の PE は M 個のプロセッサグループ (PG) にグループ化され，スーパーサブタスク (サブタスクジェネレータ) は PG に割り付ける．各 PG は固定台数の PE ($\frac{N}{M}$ 台) から構成される．また，PG 中の特定の PE はグループマスター PE と呼ぶ．第一段階の負荷分散では，スーパーサブタスクは要求駆動方式で検出した暇なグループマスター PE に割り付け，グループ間の負荷の均等化が行なわれる．

次に，M 個の PG に割り付けられたサブタスクジェネレータは，第 2 負荷分散レベルに遷するまでスーパーサブタスクをサブタスクに分割し，グループ内の PE に割り付けを行なう．これを第二段階の負荷分散と呼ぶ．第二段階の負荷分散では，要求駆動方式で検出した暇な PE にサブタスクを割り付け，グループ内の負荷の均等化が行なわれる．

なお，ここでは簡単のため 2 段階で負荷分散を行なった場合について説明したが，プロセッサ台数が更に多く二段階でサブタスクを生成してもボトルネックとなる場合には，更に多段階で行なえば良い．

3.3 グループマージ

マルチレベル動的負荷分散方式は台数拡張性があるが，スーパーサブタスクの数が充分多くない時には，グループ間の負荷の不均衡が生ずる場合がある．この問題を部分的に解決するために，我々はグループマージ手法を取り入れた (図 5)．

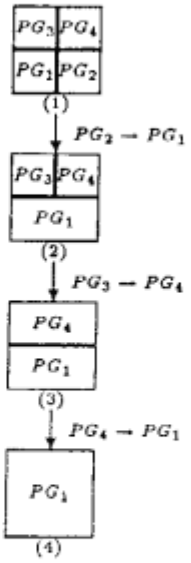


図 5: グループマージ手法

図 5(1) は N 台の PE が 4 つのプロセッサグループに分けられている事を表わす．ここでスーパーサブタスクが 5 個あったとすると，最初の 4 個は PG_1, PG_2, PG_3,PG_4 に割り付けらる．次に PG_1 が全て仕事を終えると，5 番目のスーパーサブタスクは PG_1 に割り付けられる．この時点で 4 つの PG は全て忙しい状態である．次に PG_2 が仕事を終えて暇になると，割り付けるスーパーサブタスクがないので PG_2 は PG_1 にマージされる (図 5(2))．同様にしてグループがマージされていき，最終的には全ての PG が PG_1 にマージされる (図 5(4))．

このグループマージ手法では，グループのマージが進んでグループを構成する PE 台数が多くなるにしたがって，再度サブタスク生成がボトルネックになるという問題点がある．我々の実験例では本手法はうまく働いているが，ボトルネックとならないような手法の研究を今後行なう必要があろう．

4 性能の測定とその評価

前章で示したマルチレベル動的負荷分散方式は，OR 並列型の全解探索問題である詰込みパズルに適用して，疎結合並列計算機マルチ PSI 上で性能評価を行なった．

4.1 問題の説明

詰込みパズルは，様々な形をしたピースを長方形のケースに詰め込むパズルである (図 6)．ここでは，ピースをケースに詰め込む全ての方法を求める全解探索問題として計測に用いた．なお，このパズルはピースが全

	サブタスクの数 (N)	総リダクション数 (R) (×1,000)	サブタスクの総リダクション数 (S) (×1,000)	ジェネレータの総リダクション数 (G) (×1,000)	$\frac{G}{R}$ (%)	サブタスクの平均リダクション数 (×1,000)
L1	13	8,269	8,267	1.7	0.0	636.0
L2	118	8,273	8,267	5.4	0.0	70.1
L3	485	8,289	8,253	35.9	0.4	17.0
L4	1,583	8,321	8,201	120.6	1.4	5.2
L5	5,625	8,446	8,049	397.5	4.8	1.4
L6	16,124	8,854	7,774	1,080.4	12.2	0.5
L7	38,105	-	-	-	-	-
L8	64,980	-	-	-	-	-
L9	14,560	-	-	-	-	-
L10	3,166	-	-	-	-	-

表 1: サブタスクの数と粒度

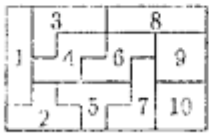


図 5: 詰込みパズル

て5つの四角からなる時にはペントミノとして知られている.

このパズルを解くには, まずケースの隅の方のある位置にピースを置くための置き方を求める. ピースを1つケースに詰める事が, 探索を1段深める事に相当する. ここで, 複数の置き方がある場合にはそれぞれの置き方に対する部分探索空間は互いに独立であり, OR木で表わされる. ピースを一つ置くと, 順次その隣の空いている位置に置くための置き方を求め, ケースが全てピースで埋まったらそれが解である. また, 途中で置けるピースがなくなったら, その探索枝は終了である. なお, 探索のレベルを深めるにつれてOR木の数は次第に増加する. ただし, 置けるピースがなくなった時にOR木は枝刈りされるため, ある探索レベルを境にしてOR木の数は急激に減少する.

4.2 マルチ PSI/V2 と並列言語 KL1

計測は, ICOT で開発した並列推論マシン実験機であるマルチ PSI/V2 [10] の上で行なった. マルチ PSI/V2 は, 並列論理型言語 KL1 [3] を高速に処理する疎結合並列計算機で, 最大 64 台の PE から構成される. 各 PE は逐次型推論マシン PSI-II の CPU であり, 8 × 8 の格子状の高速なネットワークで接続されている. 並列言語処理系はマイクロプログラムで記述されており, 1 プロセッサ当たりの append 実行性能は 130KRPS[1] である.

並列論理型言語 KL1 は Flat GHC をベースとしており, メタプログラム機構, プラグマ機構等が拡張されている. これらの機構は, OS の記述及び負荷分散の指定のために利用される. マルチ PSI の OS は PIMOS [3] で, 全て KL1 で記述されており, OS の基本機能を全て提供している. なお, 計測は OS 及び KL1 の計測機能を利用して行なった.

4.3 暇なプロセッサの検出方法

暇なプロセッサの検出は, KL1 が言語レベルで提供する優先度制御機能を利用した. 本機能は, プログラムの実行効率を高めるために提供されているものであり, 4,096 段階の優先度を用いて細かくプログラムの実行を制御できる. ここでは, 問題を解くためのサブタスクには高い優先度を与え, サブタスクの要求を行なうためのメッセージを送るタスクには低い優先度を与えた. 従って, プロセッサが暇になった時にのみ優先度の低いタスクが実行される. 優先度制御機能を利用した暇なプロセッサの検出方法は大変簡単であり, またインプリメンテーション上のオーバヘッドは大変小さく, また OS の機能を必要としないのを特徴とする.

4.4 サブタスクの粒度の計測

表 1 は, 負荷分散レベルを変えた時のサブタスクの数とその粒度に関して計測した結果である. 表中, L1~L10 は負荷分散レベルを示す. 各 Ln に対応するサブタスクの数 (N) は, 探索のレベルが n の時に生成されるサブタスクの数, 即ち OR ノードの数を示す. L2~L10 の総リダクション数 (R) は, 二段階の負荷分散 L1-Ln を行なった時の総リダクション数を示す. L1 の総リダクション数 (R) は, 1 レベルの負荷分散 L1 を行なった時の総リダクション数を示す. サブタスクの総リダクション数 (S) は, 生成されたサブタスクのリダクション数の総和である. ジェネレータの総リダクション

[1] Reduction Per Second: 一秒当たりに実行されるリダクション (推論ステップ) の数であり, 大雑把に言って 1 リダクションは手続き型言語の数十命令に相当する

数 (G) は，スーパーサブタスク及びサブタスクの生成に要するリダクション数の総和であり，$R-S$ によって計算される．$\frac{G}{R}$ は，総リダクション数 (R) 中サブタスクの生成に要するリダクション数の占める割合である．サブタスクの平均リダクション数は，生成されたサブタスクの平均粒度であり，$\frac{S}{N}$ によって計算される．

図 7 はサブタスクの粒度の分布を示すグラフである．X 軸はサブタスクの粒度 (リダクション数)，Y 軸はサブタスクの数を示す．

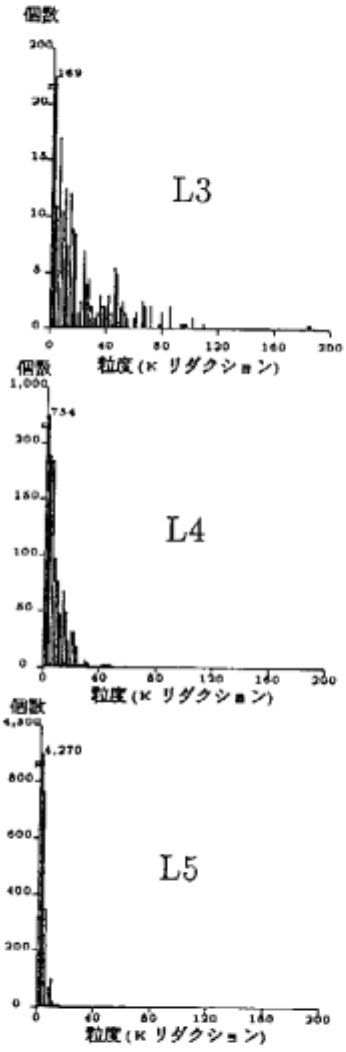


図 7: サブタスクの粒度分布

台数		1 台	8 台	16 台	32 台	64 台
実行時間 (秒)	L3	260.4	36.7	22.4	16.8	13.2
台数効果	L3	1	7.1	11.6	15.5	19.7
台数効果率 (%)	L3	100	88.8	72.5	48.4	30.8

表 2: 一段階負荷分散を行なった時の実行性能

台数	1 台	8 台	16 台	32 台	64 台
実行時間 (秒)					
L1-L3	259.2	33.7	18.8	10.4	7.8
L1-L4	261.2	34.2	17.5	9.2	5.3
L1-L5	262.4	37.8	18.7	9.8	5.8
L1-L6	278.4	46.9	26.5	17.8	14.2
L2-L3	260.3	34.5	19.7	10.9	6.6
L2-L4	264.8	34.4	17.6	9.4	5.3
L2-L5	265.1	37.3	18.8	9.7	5.3
L2-L6	273.7	50.8	26.7	15.5	11.2
台数効果					
L1-L3	1	7.7	13.8	24.9	33.2
L1-L4	1	7.6	14.9	28.4	49.3
L1-L5	1	6.9	14.0	26.8	45.2
L1-L6	1	5.9	10.5	15.6	19.6
L2-L3	1	7.5	13.2	23.9	39.4
L2-L4	1	7.7	15.0	28.2	50.0
L2-L5	1	7.0	14.1	27.3	50.0
L2-L6	1	5.4	10.3	17.7	24.4
台数効果率 (%)					
L1-L3	100	96.3	86.3	77.8	51.8
L1-L4	100	95.0	93.1	88.8	77.0
L1-L5	100	86.3	87.5	83.8	70.6
L1-L6	100	73.8	65.6	48.8	30.6
L2-L3	100	93.8	82.5	74.7	61.6
L2-L4	100	96.3	93.8	88.1	78.1
L2-L5	100	87.5	88.1	85.3	78.1
L2-L6	100	67.5	64.3	55.3	38.1

表 3: 二段階負荷分散を行なった時の実行性能

4.5 実行性能の計測

実行性能の計測は，64 台構成のマルチ PSI/V2 上で行なった．一段階の負荷分散を行なった時の性能は，プロセッサ台数を 1, 8, 16, 32, 64 台とした時のそれぞれについて計測し，一番性能が良かった負荷分散レベル L3 のデータを表 2に示す．二段階の負荷分散を行なった時の性能は，第一負荷分散レベルを L1, L2 に設定した時のそれぞれについて，第二負荷分散レベルを L3, L4, L5, L6 に設定した時の性能を測定し，その結果を表 3に示す．

なお，二段階の負荷分散はプロセッサグループ (PG) の大きさを 4 台のプロセッサ (PE) としたが，これは 8PE の性能を計測するためである．従って，64PE の時

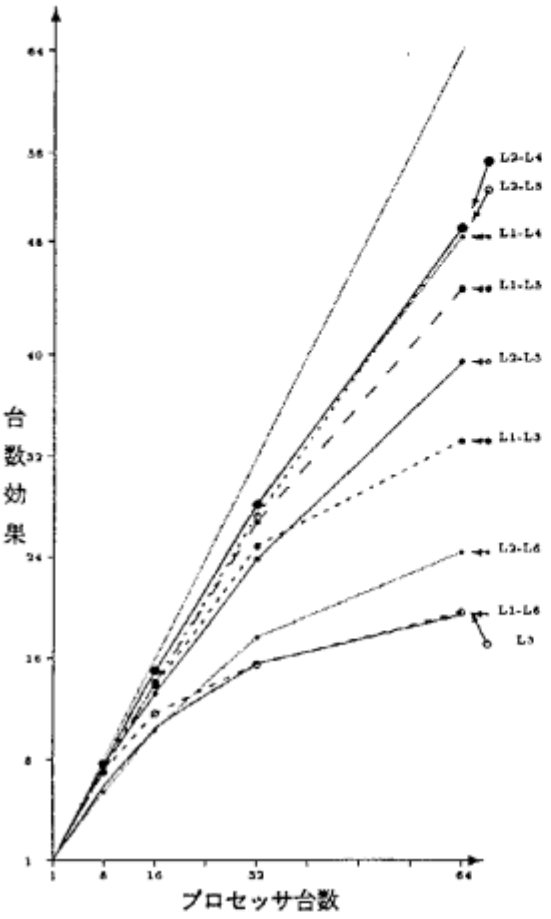


図 8: 台数効果

には 16PG から構成され，32PE の時には 8PG，16PE の時には 4PG，8PE の時には 2PG から構成される．また，1PE の時の性能は全てのサブタスクを同一 PG，及び同一 PE に割り付けた場合について計測した．表中の各項目は次の通りである．また，台数効果をあらわすグラフを図 8に示す．

実行時間 (T_N) はマルチ PSI/V2 のシステムクロックを用いて計測した．台数効果 (S_N) は 1 台で実行した時の実行時間 (T_1) と N 台で実行した時の実行時間 (T_N) との比で表わされ，$\frac{T_1}{T_N}$ にて計算される．台数効果率 (SR_N) は $\frac{S_N}{N} \times 100$ にて計算される．

4.6 計測結果

表 1中で，負荷分散レベル L1 では 13 個のサブタスクが生成され，L8 までその数は増えて，その後 OR ノードが枝刈りされるために数は減っている．L10 の時には 3,106 個であるが，これは解の数を示している．

総リダクション数 (R) は負荷分散レベルを深くするに従って増えている．ここで，1 つのサブタスクを分散するのに要するコストはほぼ一定であり，約 35 リダクションである．従って，サブタスクの数が増えるに従って総リダクション数も増える．

表 2からわかるように，一段階の負荷分散を行なった時にはプロセッサの台数を増やすにつれて実行時間は小さくなっているが，32PE 或いは 64 PE の時には台数効果率は 50% 以下になっており，頭打ちとなっている．これは，サブタスクの生成ボトルネック或いは負荷の不均衡が原因である．この事に関しては，次章にて詳しく述べる．

表 3からわかるように，二段階の負荷分散を行なった時には Li-Lj の全ての組で実行時間は小さくなっており，ほぼ線形に近い台数効果が得られている．64 台で一番性能が良いのは L2-L4 及び L2-L5 の組であり，50 倍の台数効果が得られている．一段階と二段階の負荷分散を比較すると，特に 32 台と 64 台の時の性能向上が著しい．なお，表中で四角で囲まれた性能は，それぞれのプロセッサ台数の時一番性能が良かったものと二番目に良かったものを示す．

5 考察

第 2章で述べたように，サブタスクの数と粒度は互いに相反する 2 つの要求条件を満たさねばならない．一方では負荷の均等化のためにサブタスクの数は充分多くなければならず，他方では生成がボトルネックとならないように粒度は充分大きくなければならない．本章では，これらの要求条件を式とグラフを用いて明確にし，計測結果の解析を行ない，更に新たな問題が与えられた時に本方式を適用するための指針を示す．

5.1 サブタスクの数に対する要求条件

ここでは，プロセッサの平均稼働率がある値以上になる事を保証するために必要なサブタスクの数の下限を求めてみる．ただし，以下に示すようなやや乱暴な仮定を置く．

与えられた OR 並列問題とプロセッサの台数は固定とする．また，暇なプロセッサがサブタスクを要求してからサブタスクが割り付けられるまでの時間は，各サブタスクの実行時間と比較すると無視できるもので，サブタスクの生成はボトルネックになっていないものとする．

ここでプロセッサの台数を N_{PE}, とし，N_{PE}, -1 台の各プロセッサは平均粒度と等しい大きさのサブタスクを K 個ずつ実行するものとする．残りの 1 台のプロセッサは，同じ粒度のサブタスクを K 個実行した後，最後に最大粒度のサブタスクを 1 個実行するものとする．この時，平均粒度のサブタスクの個数は次式であらわされる．

$$\text{サブタスクの数} = K \times N_{PE}, \qquad (1)$$

最後のサブタスクを実行中，N_{PE}, -1 台のプロセッサは暇にしているので，この時に平均稼働率は一番悪くな

る．

$$\text{平均稼働率} = \frac{K \times \text{平均粒度}}{K \times \text{平均粒度} + \text{最大粒度}} \quad (2)$$

この最悪ケースにおいて期待稼働率を達成する，即ち平均稼働率 $\geq$ 期待稼働率 を満たすためには，上の二式から導かれる次式のように，充分多くのサブタスクの数が必要である．

$$\text{サブタスクの数} \geq N_{PE_s} \times \frac{\text{期待稼働率}}{1 - \text{期待稼働率}} \times \frac{\text{最大粒度}}{\text{平均粒度}} \quad (3)$$

ここで，期待稼働率はユーザによって与えられるものであり，最大粒度と平均粒度はプログラムに依存する．例えば 64PE のシステムを考えた時，最大粒度が平均粒度の 10 倍大きいと仮定し，平均稼働率として 80% を期待したとすると，次の条件

$$\text{サブタスクの数} \geq 64 \times \frac{0.8}{1 - 0.8} \times 10 = 2,560 \quad (4)$$

を満たせば平均稼働率 80% が保証され，負荷分散レベルをチューニングしてサブタスクの数を決める際のガイドラインとなるものである．なお，式 (3) は最悪ケースの場合に期待稼働率を保証するものであり，一般的には更にサブタスクの数を少なくしても，期待する平均稼働率を達成できる場合が多い．

5.2 サブタスクの粒度に関する要求条件

次に，サブタスクの生成がボトルネックとならないために必要なサブタスクの粒度の下限を，次のような制約条件の下で求めてみる．

サブタスクの生成に要するコスト (生成コスト) とその分散に要するコスト (分散コスト) は，マスター PE のみにかかるものとし，マスター PE はサブタスクの生成と分散のみに専念し，他の PE がサブタスクを実行するものとする．この時，サブタスク生成がボトルネックになっていなければ，N_{PE_s} 台のプロセッサで $(N_{PE_s} - 1)$ 倍の台数効果が得られる．ここで，平均粒度が (生成コスト + 分散コスト) の $(N_{PE_s} - 1)$ 倍以上大きければ，サブタスクの生成はボトルネックとならない．

$$\text{平均粒度} \geq (\text{生成コスト} + \text{分散コスト}) \times (N_{PE_s} - 1) \quad (5)$$

この条件は，平均粒度のサブタスクが生成される時のものである．従って，例えば平均より小粒度のサブタスクが最初に連続的に生成されるような場合には，ボトルネックとなる場合もある．従って，平均粒度は条件式 (5) よりも大きいのが望ましい．

5.3 問題の規模と台数効果

条件式 (3) は，システム全体の稼働率をある値以上に保つために必要なサブタスクの数の下限を与えたものであり，同時に平均粒度の上限を与える式でもある．また，条件式 (5) はサブタスクの供給がボトルネックにならないために必要なサブタスクの平均粒度の下限を与える式であり，同時にサブタスクの数の上限を与えたものである．ここでは，サブタスクの数の下限と上限，或いは粒度の上限と下限と，問題の規模及び台数効果の関係について考察する．図 (9) は，これらの関係を大規模の問題 (a) 及び小規模の問題 (b) のそれぞれについて表したグラフである．

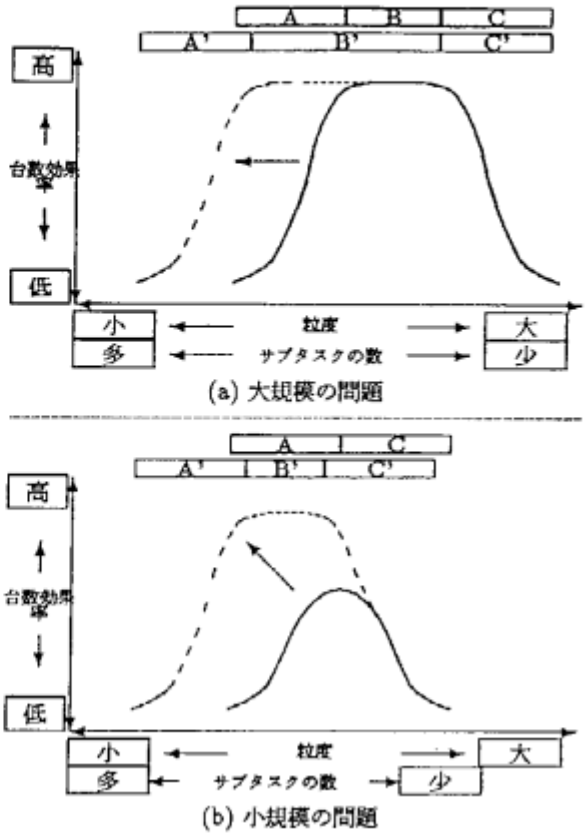


図 9: 粒度と台数効果

領域 A は，条件式 (5) を満たしていないためにサブタスクの生成がボトルネックとなり，グラフは左に傾いている．領域 C は，条件式 (3) を満たしていないために負荷の不均衡が生じて，グラフは右に傾いている．

ここで (b) のグラフを (a) に重ね合わせた時，異なる規模の問題のグラフはそれぞれ領域 A の部分では接近しているが，領域 C の部分では離れている．これは，同じ粒度であっても問題の規模が違うとサブタスクの数が違うからであり，次の関係式から明らかである．

$$\text{サブタスクの数} = \frac{\text{問題の規模}}{\text{平均粒度}} \quad (6)$$

大規模の問題の場合には，領域 A と C の間に広い平坦な領域 B があり，その部分では高い台数効果率を得ることができる．このことは，大規模の問題の方が小規模の問題よりも負荷の均等化が易しいことを示している．小規模の問題では，領域 A と C はオーバーラップしており，サブタスクの生成ボトルネックと負荷の不均衡の双方が原因となって良い台数効果を得るのは難しい．

ここで，図9を用いて我々のマルチレベル負荷分散を考察する．本方式は，領域Aにおけるサブタスク生成ボトルネックを解消したものであり，条件式(5)中のプロセッサ台数(N_{PE_s})を，プロセッサグループ中のプロセッサ台数($N_{PE_s in PG}$)に置き換えたものとみなすことができる．すなわち，ボトルネックを起こさないための粒度の下限を小さくする効果がある．これは，図中点線で表されたグラフのように大規模の問題では台数効果率のカーブを左方向に引き延ばし，小規模の問題ではカーブを左上方向に持ち上げる効果がある．

5.4 計測結果の考察

ここでは，詰込みパズルの解法プログラムに条件式(3)と(5)を適用して最適なサブタスクの数と粒度を求め，計測結果の考察を行なう．

まず，負荷の均等化に必要なサブタスクの数を条件式(3)から求める．ここで，最大粒度は図7中のL3のデータ(180)を用い，平均粒度は表1のL3のデータ(17.0)を用い，期待稼働率を80%と決め，プロセッサ台数は64とすると，式(3)から2,710個が求まる．表1でサブタスクの数をみると，L5とL6がこの要求条件を満たしており，L4では要求の60%程度のサブタスク数である．表3をみると，L1-L5とL2-L5については良い台数効果が得られているが，Ln-L6に関しては後に述べる原因で性能は良くない．またL4はこの条件を満たしてはいないが，負荷の不均衡は表面化せずに良い性能が得られている．これは，条件式(3)が最も厳しい場合の要求条件を与えているためである．

次に，サブタスクの生成がボトルネックとならないために必要な粒度を求める．ここで，サブタスクの生成コストは詰込みパズルの場合は約40リダクションで，マルチレベル動的負荷分散を適用した時に負荷分散に要するコストは約35リダクションである．また，プロセッサグループの大きさは4台とすると，必要な粒度は式(5)から225リダクションである．表1をみると，L1からL6の全ての負荷分散レベルでこの条件を満たしているが，Ln-L6は表3からわかるように良い性能は得られていない．

L6は条件(3)と(5)の両者を満たしているのに関わらず良い性能を得られていないが，この原因はグループマージの問題であると考えられる．即ち，プロセッサグループがマージされて次第に大きくなるにつれて，条件式(5)中のプロセッサ台数は多くなる．例えば，実行が開始された当初はプロセッサ台数が4台であっても，グループマージが行なわれて最終的に64台になった時には，要求される平均粒度は4,625リダクションとなってL6はこの条件を極端に満たさない．

以上の結果より，L3, L4, L5, L6が図9(b)の小規模の問題のグラフ中のどこに位置するかをみてみる．まず二段階の負荷分散の適用により，グラフは点線のカーブのように左上方向に持ち上げられている．L3は負荷の不均衡を生じているので領域C'に属すると考えられる．L4とL5は良い性能が得られており，領域B'に属すると考えられる．L6は，実行の初期ではB'に属するが，グループマージにともなって二段階の負荷分散が一段階の負荷分散と同等になってしまうため，最終的にA領域に属するものと考えられる．

5.5 最適な負荷分散レベルの決定方法

OR並列問題にマルチレベル動的負荷分散方式を適用する際には，最適なサブタスクの数と粒度を得るために負荷分散レベルをチューニングする必要があるが，ここではそのための指針を示す．

(a) 推測或いは実測によって問題の規模を求める．

(b) 期待稼働率を決め，条件式(3)より要求されるサブタスクの数を求める

(c) (a)と(b)からサブタスクの平均粒度を求める

(d) サブタスクの生成コストを実測或いは推測によって求め，サブタスクの分散コスト(固定)からプロセッサグループの大きさを条件式(5)から導く．

(e) 問題の規模が大きい場合には，条件式(5)中のプロセッサの数は条件式(3)中のものと同じか或いは大きくなるため，1レベルの負荷分散が効果的である．

(f) $2 \leq N_{PE_s}$(条件式(5)) $\leq N_{PE_s}$(条件式(3))を満たす時には，マルチレベルの負荷分散が効果的である．

(g) N_{PE_s}(条件式(5)) ≤ 2 の時，問題の規模が小さ過ぎるために，そのプロセッサ台数では良い台数効果は得られない．この場合は使用するプロセッサ台数を少なくしてサブタスクの数を減らし，1レベルの負荷分散を試みてみる．

以上の手順で本方式を適用することによって，多くの試行錯誤を繰り返すことなく，与えられた問題に対して最大の台数効果を得ることができる．

6 おわりに

以上，疎結合並列計算機マルチPSI/V2上での，OR並列型全解探索問題に適したマルチレベル動的負荷分散方式について述べた．本方式はプロセッサをグループ化し，グループレベルの負荷とプロセッサレベルの負荷の均等化を階層的に行なうものである．階層構造をなしているため，プロセッサの台数拡張性がある．ただし，グループマージ手法には改良の余地がある．

本方式は詰込みパズルの全解探索問題に適用し，メッシュ結合型の疎結合並列計算機であるマルチPSI上に実現して計測を行なったところ，次のようにほぼ線形に

近い台数効果が得られた: 8台で7.7倍, 16台で15倍, 32台で28.4倍, 64台で50倍.

またサブタスクの数と粒度についての定式化を行なったが, これは疎結合並列計算機上でのOR並列問題一般の負荷分散を考える際の指針となるものである. 本方式はOR並列問題に限らず, アルファベータ枝刈り問題等プロセッサ間通信があまり頻繁に起こらない木構造の探索型問題一般に適用可能である. 各種応用プログラムへの本方式の適用が, 今後の課題である.

7 謝辞

本研究の機会をいただいたICOTの内田俊一第4研究室室長, データの計測にあたって多くの測定用ツールを整備していただいた中島克人氏(現在 三菱電機(株)情報電子研究所), 及び様々な助言をいただいた各氏に感謝致します.

参考文献

[1] K. M. Baumgartner, and B. W. Wah. "GAMMON: A Load Balancing Strategy for Local Computer Systems with Multiaccess Networks". In *IEEE Transactions of Computers*, Vol. 38, No.8, pages 1,098–1,109, Aug. 1989.

[2] T. Chikayama. "Load balancing in a very large scale multi-processor system". In *Proceedings of Fourth Japanese-Swedish Workshop on Fifth Generation Computer Systems*. SICS, 1986.

[3] T. Chikayama, H. Sato, and T. Miyazaki. "Overview of the parallel inference machine operating system (PIMOS)". In *Proceedings of the International Conference on Fifth Generation Computer Systems 1988*, pages 230–251, 1988.

[4] T. C. K. Chou, and J. A. Abraham. "Load Balancing in Distributed Systems". In *IEEE Transactions of Computers*, Vol. SE-8, No.4, pages 401–412, Jul. 1982.

[5] E. W. Felten and S. W. Otto. "A highly parallel chess program". In *Proceedings of the International Conference on Fifth Generation Computer Systems 1988*, pages 1001–1009, Dec. 1988.

[6] C. Ferguson, and R. E. Korf. "Distributed Tree Search and its Application to Alpha-Beta Pruning". In *Proceedings of the Seventh National Conference on Artificial Intelligence 1988*, Vol. 1, pages 128–132, Aug. 1988

[7] A. George, M. T. Heath, J. Jiu, and E. Ng. "Solution of Sparse Positive Definite Systems on a Shared-Memory Multiprocessor". In *International Journal of Parallel Programming*, Vol. 15, No. 4, pages 309–325, 1986.

[8] S. Hiroguchi, and Y. Shigeki. "Optimal Number of Processors for Finding the Maximum Value on Multiprocessor Systems". In *Proceedings of The Twelfth Annual International Computer Software and Applications Conference 1988*, pages 308–315, Oct. 1988

[9] V. Kumar and V. N. Rao. "Parallel Depth-First Search, Part I: Implementation. In *International Journal of Parallel Programming*, 16(6), 479–499, 1988

[10] 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*, pages 436–451, 1989.

[11] L. M. Ni, and Kai Hwang. "Optimal Load Balancing Strategies for A Multiple Processor System". In *Proceedings of 10th International Conference of Parallel Processing 1981*, pages 352–357, Aug. 1981 .

[12] D. M. Nicol, and F. H. Willard. "Problem Size, Parallel Architecture, and Optimal Speedup". In *Journal of Parallel And Distributed Computing*, 6, pages 404-420, 1988

[13] S. Pulidas. "Imbedding Gradient Estimators in Load Balancing Algorithms". In *Proceedings of 8th International Conference on Distributed Computing Systems 1988*, pages 482–490, Jun. 1988

[14] A. N. Tantawi. "Optimal Static Load Balancing in Distributed Computer Systems". In *Journal of the Association for Computing Machinery*, Vol. 32. No. 2, pages 445–465, April 1985.

[15] E. Tick. "Compile-Time Granularity Analysis for Parallel Logic Programming Languages". In *Proceedings of the International Conference on Fifth Generation Computer Systems 1988*, ICOT, 1988.

[16] J. Schaeffer. "Distributed Game-Tree Searching". In *Journal of Parallel And Distributed Computing*, 6, pages 90-114, 1989

マルチ PSI 上の最短経路問題の実現と評価

和田 久美子, 市吉 伸行
(財) 新世代コンピュータ技術開発機構

概要

最短経路問題は重要なグラフ問題のひとつであり，今まで様々な逐次アルゴリズム及び並列アルゴリズムが開発されている．しかし，大規模汎用 MIMD マシン上で大規模なグラフを対象に実行する場合，プロセッサへの効率の良い負荷分散方法については，現在まであまり研究されていない．本稿では，最短経路問題を解く 1 つの分散アルゴリズムを提案し，疎結合マルチプロセッサであるマルチ PSI/V2 上での大規模な格子状グラフを用いた実行における負荷分散方式について考察する．実験した負荷分散方式は，グラフの局所性を多く保つ 2 次元単純マッピング方式，プロセッサ稼働率の高い 2 次元多重マッピング方式，そして理論的にはプロセッサ数の 3 分の 2 の台数効果を得ることのできる 1 次元単純マッピング方式である．それぞれのマッピング方式を用いた実行結果を示し，解析を行う．

1 はじめに

最短経路問題は，輸送やパイプライン網の建設，集積回路の設計などにおいて非常に重要な問題である．そのため，最短経路問題を高速に解くアルゴリズムの様々な研究がされてきた．逐次アルゴリズムでは，n 個の点と e の辺を持つ与えられた有向グラフに対して，$O(n^2)$ 時間で 1 点から全点への最短経路問題を解く Dijkstra のアルゴリズム [6] がある．Warshall-Floyd のアルゴリズム [12, 7] は，$O(n^3)$ 時間で全点から全点への最短経路問題を解く．Johnson は，e が n^2 よりずっと小さい時，データ構造を工夫することによって Dijkstra のアルゴリズムの実行時間が $O(e \log n)$ に改良できることを示した [9]．Fredman と Tarjan は，Fibonacci heap を用いて，$O(e + n \log n)$ 時間で 1 点から全点への最短経路問題を解くアルゴリズムを提案した [8]．並列 / 分散アルゴリズムも同様に研究されている [11]．Deo ら [5] は，8 プロセッサから成る密結合 MIMD マシン上で効率良く動作するよう，逐次アルゴリズムを並列化した．Chandy と Misra[3] は，分散最短経路アルゴリズムを与えた．最近では，Driscoll らが，n 個の点と $e \geq n \log n$ 個の辺を持つグラフに対して relaxed heap を用いれば，$P \geq e/(n \log n)$ 台のプロセッサから成る EREW PRAM マシンでは，$O(e/p)$ 最適実行時間を得ることができることを示した [1]．

しかし，10^2 程度のプロセッサ数の大規模汎用 MIMD マシン上で 10^3 以上の点を持つ大規模なグラフに対する最短経路問題に関しては，まだあまり研究されていないようである．このような大規模マシンは密結合することができず，データへのアクセスはコスト高である．従ってグラフをどのようにプロセッサに割当てるかが実行効率を左右する．

本稿では，最短経路問題を解く分散アルゴリズムを提案し，並列推論マシンの実験機であるマルチ PSI/V2[10] 上での大規模な格子状グラフに対するマッピング方式について考察する．マルチ PSI/V2 は疎結合マルチプロセッサで，8×8 のメッシュネットワーク上に最大 64 プロセッサまで接続される．実験したマッピング方式は，グラフの局所性を多く保つ 2 次元単純マッピング方式，プロセッサ稼働率の高い 2 次元多重マッピング方式，そして理論的にはプロセッサ数の 3 分の 2 の台数効果を得ることのできる 1 次元単純マッピング方式である．それぞれの方式を用いての実行結果を示し，実際の台数効果と通信オーバヘッドについて解析する．

2 分散アルゴリズム

この節では，1 点から全点への最短経路問題に対する分散アルゴリズムを与える．最短経路問題は，グラフ理論の用語を用いて次のように表される．与えられた n 個の点の集合 V と e 個の辺の集合 E によって，有向グラフ $G = (V, E)$ を定義する．辺は点の順序対である．点の対から非負の実数値へのコスト関数 c を与える．V の点 v_i から点 v_j への辺に対して，$c(v_i, v_j)$ をその辺のコスト，あるいは長さという．考察を容易にするため，点 v_i から点 v_j への辺が存在しない場合，$c(v_i, v_j) = +\infty$，また，各点 v_i に対して，$c(v_i, v_i) = 0$ であるとする．$l+1$ 個の点 $v_0, v_1, \ldots, v_l$ において，すべての $i(= 0, 1, \ldots, l-1)$ に対して点 v_i から点 v_{i+1} への辺 e_i が存在する時，辺の列 $e_0, e_1, \ldots, e_{l-1}$ を点 v_0 から点 v_l への経路と呼ぶ．ある経路に対して，その経路に含まれる辺のコストの和をその経路のコストという．グラフのある 2 点間を結ぶ経路のうち，コスト最小のものを最短経路という．以上の仮定のもとで，出発点と呼ぶ特定の 1 点 $v_0 \in V$ から V の各点への最短経路を求める問題を，1 点から全点への最短経路問題と呼ぶ．

与えられた有向グラフ $G = (V, E)$ に対して，任意に通信可能な p 個のプロセッサ $P_0, \ldots, P_{p-1}$ を用いて

最短経路問題を求める分散アルゴリズムを与える．点集合 V を適当に直和分解する．すなわち，部分点集合 $V_i(i=0,1,\ldots,p-1)$ は，$V = V_0 \cup V_1 \cup \cdots \cup V_{p-1}, V_i \cap V_j = \emptyset (i \neq j)$ なる性質を持つ．

各プロセッサは，割当てられた各点への最短となる可能性のある経路とそのコストの情報を保持する．これらはコスト経路情報によって更新される．コスト経路情報は，$cp(cost, v_j, v_i)$ の形をしており，出発点から点 v_j への経路でコストが $cost$ のものが存在し，その経路で v_j の直前の点が v_i であることを示す．アルゴリズムの実行中，各点 v_j は出発点から自分までの経路でその時点までに分かっているもののうち，最も低コストの経路のコストとその経路での自分の直前の点をそれぞれ変数 $cost_j$，$path_j$ に保持している．今，コスト経路情報 $cp(cost, v_j, v_i)$ が与えられたとする．この時，$cost < cost_j$ ならば，$cost_j$，$path_j$ の内容をそれぞれ $cost$，v_i に更新し，v_j のすべての隣接点 v_k へのコスト経路情報 $cp(cost + c(v_j, v_k), v_k, v_j)$ を生成する．コスト経路情報は，各プロセッサ内で優先度付き待ち行列に格納され，それに含まれるコスト値に従って小さいものから順に取り出される．プロセッサ P_i が v のすべての隣接点 u に対してコスト経路情報を生成した時，点 u が部分点集合 V_i に属しているならば，その情報をプロセッサ P_i の優先度付き待ち行列に格納する．そうでなければ，情報はプロセッサ間メッセージ通信によって $u \in V_j$ を割当てられたプロセッサ P_j に送信され，プロセッサ P_j の優先度付き待ち行列に格納される．

各点に対するコスト経路情報の初期値は $cost = +\infty, path =$ 未定義，すべての優先度付き待ち行列は空である．アルゴリズムは，最初に，出発点 v_0 を割当てられたプロセッサが，v_0 のすべての隣接点 v_j に対してコスト経路情報 $cp(c(v_0, v_j), v_j, v_0)$ を生成し，優先度付き待ち行列に格納する．すべての優先度付き待ち行列が空になった時，アルゴリズムは終了する．この時点で，各点 v_j の $cost_j$，$path_j$ にはそれぞれ最短経路のコストとその経路での自分の直前の点が保持されている．アルゴリズムを図1に示す．

プロセッサ台数を1とした時，アルゴリズムは Dijkstra の逐次アルゴリズムと同等に動作する．一方，n 個の点をプロセッサ n 台に1つずつ割当てた場合，アルゴリズムは Chandy と Misra の分散アルゴリズムでメッセージの上書きが可能なバッファを用いる場合 [3] とほぼ同様となる．一般に複数プロセッサを用いた場合，コスト経路情報を格納する優先度付き待ち行列も複数存在することになり，優先度管理が分散化されることになる．従って，プロセッサ P_i で優先度付き待ち行列から取り出されて処理中のコスト経路情報より本来なら優先度が低く，後で処理されるべきものが，他のプロセッサで同時或いは先に処理されてしまう恐れがある．これによって，本来 Dijkstra のアルゴリズムでは生成されない無駄なコスト経路情報が発生する．しかし，コスト経路情報の優先度管理をしない分散アルゴリズムの最悪の場合の計算時間は，点の次数 (隣接する辺の数) が有界であっても点の総数の指数オーダになることもあり，たとえ管理が分散化されたとしてもコスト経路情報の優先度管理は探索の枝刈りをするために非常に有効である．またアルゴリズムは，Dijkstra のアルゴリズムと同等の計算時間オーダで最短経路を求めることが証明される．このアルゴリズムの計算時間オーダは，点の次数が n に依存しないある値で抑えられる時，$O(e \log n)$ である．

3 疎結合マルチプロセッサでの負荷分散

ここでは，疎結合マルチプロセッサ上で我々の分散アルゴリズムを効率良く実行するためのマッピング方式について考察する．10^4 以上の点を持つ大規模なグラフに対して，10^2 以上のプロセッサを持つ大規模な疎結合マルチプロセッサ上で問題を解くことを前提とする．

疎結合マルチプロセッサ上で効率良く負荷分散を行うには，主に2つの要素，すなわち，高プロセッサ稼働率とプロセッサ間通信の抑制が鍵となる．プロセッサ稼働率を上げるには，各プロセッサへ均等に仕事が割り付けられるように，問題全体を細かい部分問題に分割し，プロセッサ当たり多くの部分問題を割当てるようにする．一方，疎結合マルチプロセッサではプロセッサ間通信は非常にコスト高なので，プロセス間通信や一般のデータアクセスはできるだけ局所的に処理できるようにするのが望ましい．しかし，これは多くの場合，より多くの部分問題への分割と相反する結果となる．次の節では実験から，負荷分散方式がこれらの2つの要素に及ぼす影響を示す．

4 実験と解析

我々は，適当なグラフを設定し，疎結合マルチプロセッサであるマルチ PSI 上でいくつかのマッピング方式の実験を行った．その実験結果を以下に示す．

4.1 実験で用いたグラフ

実験で用いたグラフは，200 × 200 の正方格子上に配置された4万個の点に対してすべての隣り合う2点間に双方向の辺を与えた有向グラフである．隅の点のひとつを出発点とした．各辺のコストの与え方に従って，2種類のグラフを実験対象とした．

グラフ 1: 各辺のコストを一定値としたもの．

グラフ 2: 各辺のコストとして1から99までの非負整数の擬似乱数を与えたもの．

4.2 実験で用いた並列マシンと言語

実験は，並列推論マシンの実験機であるマルチ PSI/V2[10] を用いて行った．これは，並列ソフトウェ

```
プロセッサ P_0 の初期化:
    begin
        for プロセッサ P_0 に属するすべての点 v_i do cost_i := ∞, path_i := 未定義 ;
        優先度付き待ち行列を初期化 ;
        cost_0 := 0 ;
        for v_0 のすべての隣接点 v_j do
            cp(c(v_0, v_j), v_j, v_0) を v_j が属するプロセッサ P_b に送信する
    end

プロセッサ P_a (a ≠ 0) の初期化:
    begin .
        for プロセッサ P_a に属するすべての点 v_i do
            cost_i := ∞ ;
        優先度付き待ち行列を初期化 ;
    end

プロセッサ P_a の解探索:

    優先度付き待ち行列が空でないならば,
        begin
            優先度付き待ち行列から cp(cost, v_j, v_i) を取り出す ;
            if cost_j > cost then
                begin
                    cost_j := cost ;
                    path_j := v_i ;
                    for v_j のすべての隣接点 v_k do
                        cp(cost + c(v_j, v_k), v_k, v_j) を v_k の属するプロセッサ P_b に送信する
                end
        end

    メッセージ cp(cost, v_j, v_i) を受信したら,
        begin
            優先度付き待ち行列に cp(cost, v_j, v_i) を格納する
        end
```

図 1: 最短経路を求める分散アルゴリズム

アの研究開発環境として新世代コンピュータ技術開発機構 (ICOT) で開発された疎結合型のマルチプロセッサで, 現在, 8 × 8 のメッシュネットワークによって最大 64 台の要素プロセッサを接続することが可能である. 各プロセッサの性能は 130K append LIPS[1]である.

プログラムは, 並列論理型言語 KL1[4] を用いて記述された. グラフの各点とコスト経路情報は KL1 のプロセスとして表現された. KL1 では, 本来のプログラムの意味を記述する部分と, 並列実行に伴う負荷分散や実行優先度などを指定する部分が分離されている. 後者はプラグマと呼ばれ, プログラムが効率良く実行されるために付加的に記述されるものであり, それによってプログラムの意味が変わることはない. プラグマは, プログラム中のボディゴールに書き添えることによって実現される. プロセッサへの点の割当てや優先度管理は, プラグマによって簡単かつ効率良く実現された. すなわち, 優先度付き待ち行列は KL1 処理系が提供する優先度制御機構を用いた. 負荷分散方式はプロセッサ番号を算出するルーチンにより簡単に実現され, 異なる負荷分散方式もプログラムの起動時にパラメータを変更することにより簡単に設定することができた. マルチ PSI/V2 で最初に実行されたのは優先度制御をしない最短経路アルゴリズムで, プロセッサ 64 台で $k = 4 \times 4$ の 2 次元多重マッピング方式を用いた場合, 100 × 100 の正方格子状グラフ (辺のコストは乱数を用いた) の生成と最短経路の探索に約 80 秒を要した. その後, 優先度管理を導入した新しい分散アルゴリズムが実現され, 同様の問題を 7 秒 (グラフ生成に 6 秒, 解探索に 1 秒) で解いた.

[1]LIPS は *Logical Inference Per Second* の省略である. 130K append LIPS は, 1 秒間に 130K 個のコンスセルをアペンドすることができることを意味する.

4.3 マッピング方式と実験結果の解析

試みたいくつかの負荷分散方式とその実験結果について述べ，簡単な解析を行う．

4.3.1 2次元単純マッピング方式

まず，我々は2次元単純マッピング方式と呼ぶ非常に単純な方式を試みた．$p = q^2$ 台のプロセッサを用いる場合，グラフを $q \times q$ 個のブロックに分割し，各ブロックを1つずつプロセッサ配置に従ってプロセッサに割当てる．例えば，16台構成の疎結合マルチプロセッサの論理プロセッサ配置が図2のようになっている時，グラフは 4×4 ブロックに分割される (図3)．プロセッサ P_0 には斜線部分のブロックが割当てられる．この方式は，同数の点を各プロセッサに割当てながら，グラフ内の局所性を最も良く保つものである．

図 2: 16台構成のマルチ PSI の論理プロセッサ配置

斜線部分はプロセッサ0に割当てられる．

図 3: 2次元単純マッピング方式におけるグラフ分割

グラフ2を用いた場合の実験結果を図9，10に示す．図9は，マッピング方式とプロセッサ台数を変化させて実行した場合の実行時間を表す．縦軸は実行時間 (秒)，横軸は台数である．図10は，図9における台数効果を表したものである．縦軸は台数効果，横軸は台数である．2次元単純マッピング方式では，結果はあまり良くないことがわかる．

理由について考えてみよう．コスト経路情報は最初に出発点の属するプロセッサで生成され，その後他のプロセッサへ波のように次第に広がっていく．各プロセッサは波が来るまで暇な状態であり，波が去った後再び暇な状態となるので，暇な時間が比較的多いのである．

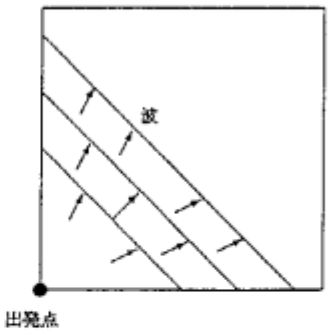


図 4: コスト経路情報生成の波

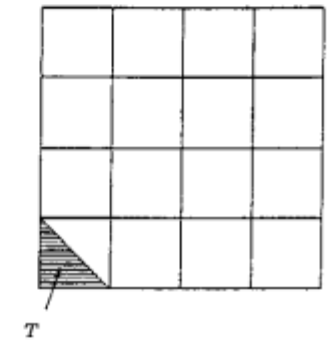


図 5: 2次元単純マッピング方式に対する評価

このマッピング方式における理想的な状態での台数効果を考える．プロセッサ間通信に要するオーバヘッドは無視できる位小さく，部分問題への分割によって生じる無駄なコスト経路情報は一切発生しないと仮定する．すると，コスト経路情報の処理の波上に存在する点の集合は，出発点からのマンハッタン距離[2]が互いに等しいものの集合となる．出発点がグラフのある隅に位置する時，コスト経路情報の処理の波は，図4のように進む．

このような仮定のもとでは，$p = q^2$ 個のプロセッサを用いて問題を解いた場合の実行時間と台数効果を以下のように評価することができる．問題はプロセッサ台数と同数の部分問題に分割されるので，部分問題は全部で q^2 個である．図5で斜線で示された三角形の中の点に対して1プロセッサがコスト経路情報を処理するのに要する時間を T とする．1プロセッサが問題全体を解くのに要する時間はグラフの点の総数に比例するので，1プロセッサのみで実行した場合の実行時間は $2Tp$

[2] 2次元平面上では，マンハッタン距離は点 $p_1 = (x_1, y_1)$，$p_2 = (x_2, y_2)$ に対して，$d(p_1, p_2) = |x_1 - x_2| + |y_1 - y_2|$ によって定義される．

となる．p 台のプロセッサで実行した場合に問題全体を解くのに要する時間は，$2Tq$ である．従って台数効果は，

$$\frac{2Tp}{2Tq} = \frac{q^2}{q} = q = \sqrt{p}.$$

すなわちプロセッサ稼働率は $1/\sqrt{p}$ で，あまり良い値ではない．しかも，この値はプロセッサ台数の増加に従ってますます悪くなるのである．実験結果から得られた台数効果は，グラフ 1 を用いた場合，4 台で 1.97，16 台で 3.87，64 台で 7.77，グラフ 2 を用いた場合はそれぞれ 1.69，3.15，6.10 となっており，プロセッサ数が 4 倍になる時，台数効果は 2 倍弱となっていることがわかる．

我々は，各マッピング方式について 16 台のプロセッサを用いてグラフ 2 に対して実行した場合の，プロセッサ稼働率と通信オーバヘッドを測定した (図 11)．棒グラフは稼働率をパーセントで表したもの ((稼働時間 / 総実行時間) $\times 100$) で，斜線部分がそのうちの通信に費やされた時間を表している．実験結果から，この方式での平均プロセッサ稼働率は 24% という値が得られており，期待される値 25% ($= 1/\sqrt{16}$) にかなり近い．

4.3.2 2 次元多重マッピング方式

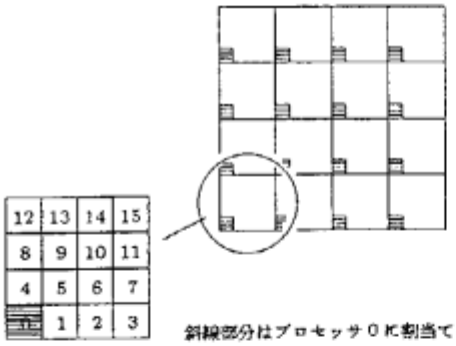


図 6: 2 次元多重マッピング方式におけるグラフ分割

2 次元単純マッピング方式では良いプロセッサ稼働率を得ることはできなかった．これは，各プロセッサがグラフのある 1 部分のみを割当てられていたためである．各プロセッサがグラフ中の分散した幾つかの部分を受け持てば問題は解決されるだろう．2 次元多重マッピング方式では，グラフはまず k 個の大きなブロックに分割され，その後さらに各ブロックが 2 次元単純マッピング方式と同様に p 個のブロックに分割される．各プロセッサは大きなブロック当たり 1 部分ずつ，合計 k 個のブロックを割当てられることになる．図 6 は，$k = 4 \times 4$，$p = 4 \times 4$ の場合を示している．グラフはまず 16 個の大きなブロックに分割され，その後さらに大きなブロックが，2 次元単純マッピング方式と同様にそれぞれ 16 個に分割される．プロセッサ P_0 は，図の斜線部分を担当する．

このようにすることにより，各プロセッサはコスト経路情報の処理の波がグラフ上に散らばった k 箇所の点集合を通り過ぎる際に稼働する．従って，各プロセッサの暇な時間は減少することが期待される．

実験結果から $k = 4 \times 4$ の時の台数効果は，グラフ 1 を用いた場合，4 台で 3.14，16 台で 9.39，64 台で 26.14，グラフ 2 を用いた場合はそれぞれ 2.56，7.25，18.26 であった．同様に $k = 8 \times 8$ の時の台数効果は，グラフ 1 を用いた場合，4 台で 2.88，16 台で 8.91，64 台で 21.34，グラフ 2 を用いた場合それぞれ 2.71，8.13，20.25 となっており，プロセッサ台数に依存せず，2 次元単純マッピング方式に比べてずっと良いプロセッサ稼働率が得られたことがわかる．実際，プロセッサ 16 台で実行した場合のプロセッサ稼働率は $k = 4 \times 4$ の時 80%，$k = 8 \times 8$ の時 94.4% となっている (図 11)．しかし，実行時間は驚くほどには改良されていない．これは，後で述べる無駄な見込み計算 (*speculative computation*) と通信オーバヘッドがかなり増大したためである．

4.3.3 1 次元単純マッピング方式

2 次元多重マッピング方式は，1 プロセッサ当たり複数個のブロックを割当てることによってプロセッサ稼働率を改良しようとするものだった．1 次元単純マッピング方式は，最小のグラフ分割でプロセッサ稼働率を上げようとする方式である．ブロックと波ができるだけ垂直に交わるようにグラフを分割すれば，やはりプロセッサ稼働率を改良することができるだろう．そこで，グラフを単に p 個の細い短冊の形に分割し，それらを 1 つずつプロセッサに割当てる．図 7 は $p = 16$ の場合である．

図 7: 1 次元単純マッピング方式におけるグラフ分割

2 次元単純マッピング方式に対して述べたのと同様の仮定のもとで，このマッピング方式の理想的な状態における台数効果を見積もる．図 8 で斜線で示された三角形ひとつの中の点に対して 1 プロセッサがコスト経路情報を処理するのに要する時間を T とする．すると，$7T$ 時間めに各プロセッサが処理するのは，図中の点で示された部分となる．結局，p 台のプロセッサを用いて問題を解くのに要する時間は，$T(3p-1)$ である．1 プロ

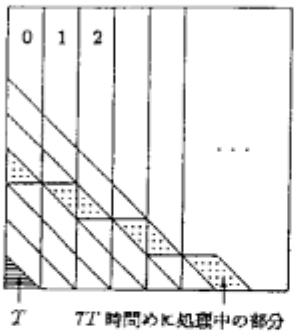


図 8: 1 次元単純マッピング方式に対する評価

セッサが問題全体を解くには $2Tp^2$ 時間かかるから，台数効果は，

$$\frac{2Tp^2}{T(3p-1)} = \frac{2p^2}{3p-1} \approx \frac{2}{3} \cdot p \ (p >> 1\text{の時})$$

となり，プロセッサ台数の約 3 分の 2 の台数効果が得られることになる．これは，$\sqrt{p}$(2 次元単純マッピング方式) ではなく p に比例する，すなわち，プロセッサ台数が増加しても効果の率は下がらないことを意味している．

実験結果から，このマッピング方式では 2 次元多重マッピング方式での実行結果とほぼ同じ程度の性能が得られたことがわかる．台数効果は，グラフ 1 を用いた場合，4 台で 2.86，16 台で 10.12 というように議論された値にほぼ近い値となっているが，64 台では 29.34 で議論された値より悪い．これは，ブロックの境界で生じるプロセッサ間通信のオーバヘッドが原因と考えられる．グラフ 2 を用いた場合では，4 台で 1.97，16 台で 5.69，64 台で 17.34 と，プロセッサ台数が増加するに従って次第に悪くなっている．

プロセッサ 16 台で実行した場合のこのマッピング方式での平均プロセッサ稼働率も 45% となっており，期待される値 68%(= $2 \cdot 16/(3 \cdot 16 - 1)$) より悪い (図 11)．

4.4 議論

実験結果から分かるように，最も単純なグラフと言えるグラフ 1 では殆ど無駄な見込み計算が生じないため，ほぼ理論的な台数効果を得ることができたといえる．理論値より若干悪い値となった主な理由は，通信オーバヘッドである．一方，グラフ 2 での実験では，台数効果は理論値よりも悪い値となった．理由のひとつは通信オーバヘッドであり，もうひとつは無駄な見込み計算である．

グラフがマッピングのためにブロックに分割される時，ブロックの境界ではプロセッサ間通信が発生する．グラフが細かいブロックに分割されるほど，ブロックの境界の長さの合計は長くなり，プロセッサ間通信が増大

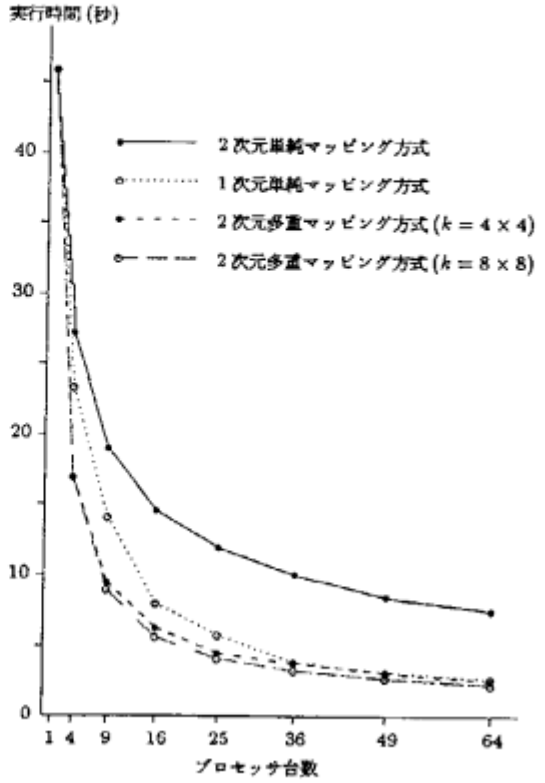


図 9: マッピング方式，プロセッサ台数と実行時間

する．通信オーバヘッドが実際に占める割合はグラフの大きさやプログラム，また，言語の実装方法，マシンなどに依存するが，ブロックの境界の長さの合計に比例する．

図 11では，実際の稼働時間のうちの通信に費やされた部分とそうでない部分の比は，2 次元単純マッピング方式で 5.3%，2 次元多重マッピング方式の $k = 4 \times 4$ の場合で 19.2%，$k = 8 \times 8$ の場合で 34.1%，1 次元単純マッピング方式で 10.2% となっている．これらをそれぞれ境界の長さの合計 $6L$, $30L$, $62L$, $15L$ で割ると (L はグラフ全体の 1 辺の長さ)，8.8%，6.4%，5.5%，6.8% で殆ど一定となっており，期待される結果にほぼ一致している[3]．

無駄な見込み計算量の解析は最も難しい．Dijkstra の逐次アルゴリズムではすべての計算が決定的で無駄がないのに比べ，我々の分散アルゴリズムのコスト経路情報の処理では見込み計算が行われる．何故なら，生成されるコスト経路情報はもし後からもっと良い情報が得られた場合，不必要となるかも知れないからである．そこで，我々は，プロセッサ台数とマッピング方式を変化させた場合の，実行中に生成されるコスト経路情報の総数を測定し，1 台で実行した場合と比較した．図 12は それを示している．超過分が最終的には無駄となったコスト経路情報の合計である．プロセッサ 16 台の場合，2 次元単純マッピング方式では 20.2%，2 次元多重マッピング方式の $k = 4 \times 4$ の時 39.1%，$k = 8 \times 8$ の時

[3] 測定誤差は 5% ～ 10% である．

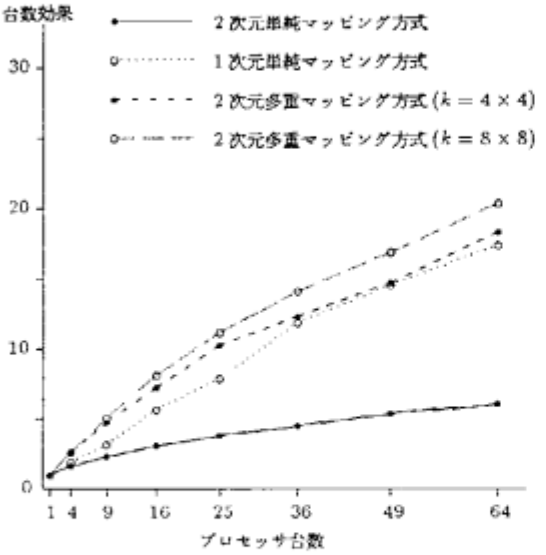


図 10: マッピング方式, プロセッサ台数と台数効果

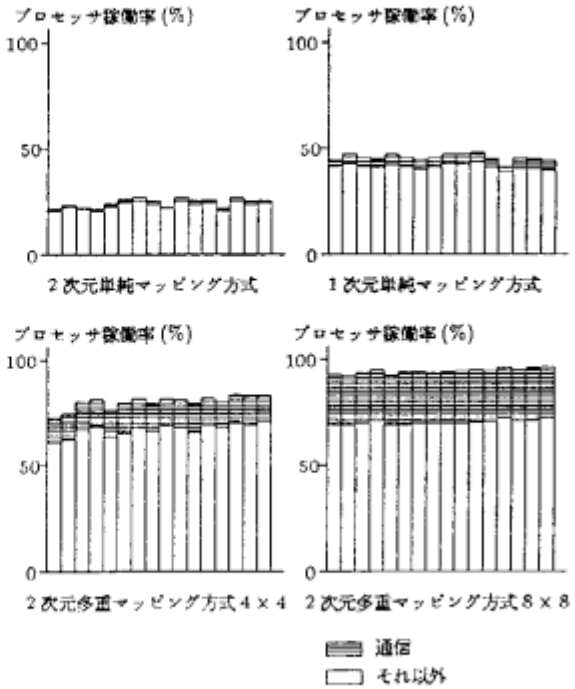


図 11: マッピング方式とプロセッサ稼働率 (プロセッサ 16 台の場合)

20.2%, 1 次元単純マッピング方式では 17.2% となっている. 8×8 分割の時に 4×4 分割の時よりも無駄な見込み計算量が減少する理由ははっきりわかっていないが, 恐らくグラフの性質によるものであろう.

今回の結果の一般性については問題があるかも知れない. 今回の結果は, 局所性の高いグラフについて当てはまる. 例えば, 平面グラフや, 3 次元空間で点が一様な密度で配置され, 辺の長さが制限されているようなものがそうである. このようなグラフは, 実際の応用事例ではよく見られる. 辺が疎でも局所性のあまりないグラフは, マルチ PSI のようなメッシュネットワークで接続されたマルチプロセッサに割当てるのは難しいだろう. このようなグラフに対しては, ハイパーキューブ結合型マルチプロセッサ[4]の方が良いマッピングがあると考えられる.

5 おわりに

最短経路問題は今まで様々な研究がされており 数々の逐次アルゴリズムや並列アルゴリズムが開発されている. しかし, 10^2 以上のプロセッサの大規模汎用 MIMD マシン上で, 点の総数が 10^3 以上の大規模なグラフに対する最短経路問題についてはこれまであまり研究されていない. 我々は, 最短経路問題を解く 1 つの分散アルゴリズムを提案し, 疎結合マルチプロセッサであるマルチ PSI/V2 上での大規模グラフに対する最短経路問題の負荷分散方式について述べた. グラフの局所性を最も良く保つ 2 次元単純マッピング方式や, 高いプロセッサ稼働率を得ることが可能な 2 次元多重マッピング方式, また, 理論的には線形の台数効果を得ることが可能な 1 次元単純マッピング方式について実験を行った. 各マッピング方式における性能を測定し, 実際の台数効果と通信オーバヘッドについて議論した. 我々の実験は今のところ正方格子状のグラフに限られているが, 考察結果は, 局所性の高いグラフ一般に対して適用することが可能である. マッピング方式に関して観察された特徴の多くは, 1 点から全点への最短経路問題に限らず, コスト最小の極大木を求めるものや, 木の深さ優先探索等, その他のグラフアルゴリズムに対しても同様に観察されるであろう.

KL1 言語について特記すべき事項は, 高い生産性と実験の容易さである. データ依存の同期機構は言語のセマンティクスとして用意され保証されているので, プログラマはプロセス間の同期について全く気を使う必要がない. KL1 では, プログラムの意味とマッピングは分離しているため, マッピング方式の変更がプログラムの意味の記述に影響することはないので, 異なるマッピング方式を簡単に試みることができた. 我々はまた, このように無駄な計算が生じるような場合にはアルゴリズムの計算量を減らすために優先度制御が必要不可欠であることを認識した.

[4] プロセッサ台数を p とする時, プロセッサ間メッセージ通信に要する routing の最大値は, 2 次元メッシュネットワークで $O(\sqrt{p})$, ハイパーキューブネットワークで $O(\log p)$ である.

謝辞

ICOT の渕一博所長, 内田俊一研究部長にこの研究の機会を与えて下さったことを感謝します. 最初に優先度制御を用いた分散アルゴリズムの考えを提起された近山隆第 2 研究室室長, 常に有益な示唆と助言をい

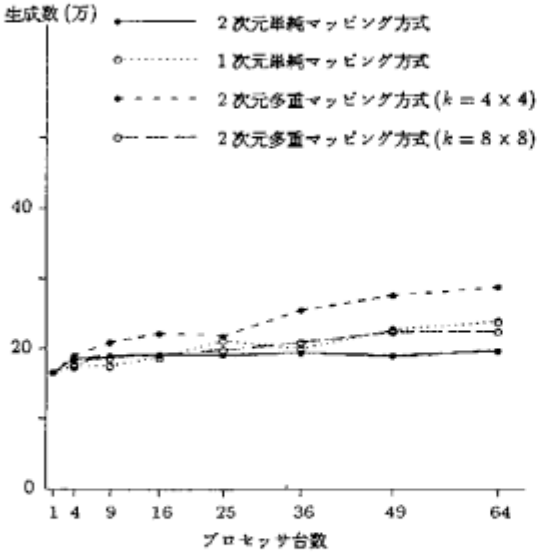


図 12: マッピング方式, プロセッサ台数とコスト経路情報の生成数

ただいた淵和男第1研究室室長, プロセッサ稼働率と通信オーバヘッドの測定をしていただいた中島克人氏(現在, 三菱電機(株)情報電子研究所), この論文を査読して下さった六沢一昭氏(現在, 沖電気工業(株)総合システム研究所), 様々な有益な助言を下さったその他のICOT第1, 2, 7研究室の研究員の方々に感謝します.

参考文献

[1] J. R. Driscoll, H. N. Gabow, R. Shrairman, and R. E. Tarjan. Relaxed Heaps: An Alternative to Fibonacci Heaps with Applications to Parallel Computation. *Comm.ACM*, Vol.31, No.11 (Nov.1988), pp. 1343-1354.

[2] A. V. Aho, J. E. Hopcroft and J. D. Ullman. The Design and Analysis of Computer Algorithms, Addison-Wesley (1974).

[3] K. M. Chandy and J. Misra. Distributed Computation on Graphs: Shortest Path Algorithms. *Comm. ACM*, Vol.25, No.11 (Nov.1982), pp. 833-837.

[4] T. Chikayama, H. Sato and T. Miyazaki. Overview of the Parallel Inference Machine Operating System (PIMOS). In *Proceedings of the International Conference on Fifth Generation Computer Systems 1988* (1988), pp. 230-251.

[5] N. Deo, C. Y. Pang and P. E. Lord. Two Parallel Algorithms for Shortest Path Problems. In *Proceedings of the 1980 International Conference on Parallel Processing*. IEEE, New York, 244-153, 1980.

[6] E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. Numerische Mathematik 1 (1959), 269-271.

[7] R. W. Floyd. Algorithm 97: Shortest Path. *Comm.ACM*, Vol.5, No.6 (1962), p. 345.

[8] M. L. Fredman and R. E. Tarjan. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. *J.ACM*, Vol.34, No.3 (July 1987), pp. 596-615.

[9] D. B. Johnson. Efficient Algorithms for Shortest Paths in Sparse Networks. *J. ACM*, Vol.24, No.1, pp. 1-13.

[10] 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), pp. 436–451.

[11] M. J. Quinn and N. Deo. Parallel Graph Algorithms. *ACM Computing Surveys*, Vol.16, No.3 (Sept.1984), pp. 319-348.

[12] S. Warshall. A Theorem on Boolean Matrices. *J.ACM*, Vol.9, No.1 (1962), pp. 11-12.

Exhaustive versus Pruned Search on the Multi-PSI

Bernard Burg
ICOT
1-4-28 Mita, Minato-ku, Tokyo 108, Japan
burg@icot.icot.jp

April 18, 1990

Abstract

Exhaustive search programs running on MIMD computers show impressive speedups, almost linear with the number of processors. But since it is generally acknowledged that exhaustive search programs lead to combinatorial explosions, they cannot be fitted to real applications, so their fast performance measurements are not very useful in the field of AI.

In aiming to extend the use of parallel computers towards real applications, we study pruned search, in domains dealing with bulky data [1], such as machine learning. New difficulties also appear in task partitioning between processors, load balancing and data spreading. These problems are analyzed, the main parameters are studied and finally, solutions are implemented on the Multi-PSI.

We show that linear speedups cannot be obtained in pruned search running on an MIMD machine, unless the communication costs between processors are neglected. Performance measurements issued from the field of machine learning illustrate our study.

1 Introduction

Most of the textbooks on parallel programming [3, 5, 10], begin by describing hardware architectures, then show parallel implementations of programs such as fast Fourier transform, matrix operations, numerical operations, graph algorithms and combinatorial search. These programs cover a large range of applications. As soon as regularities, either in data or control structures, may be extricated, parallelism proves to be efficient, provided that the target machine has the accurate architecture for the granularity of the problem.

In practice however, one has to deal with different constraints typically stated as follows:
Given a parallel machine and a field of research, find the best parallel implementations.

The hypotheses of this study are as follows:

- The parallel environment is a Multi-PSI [9] system, a loosely-coupled multiprocessor running the concurrent logic programming language KL1 [4]. Up to 64 processors are connected in an 8×8 mesh network.

- The goal of the research is to implement efficient search methods, that work with heavy data in a large search space. As an example of such algorithms, we developed a parallel version of a Michalski-like [8] machine learning algorithm.

The context of this work leads to the following two remarks:

- Most of the implementations of AI algorithms shown in the references have been implemented on massive parallel SIMD machines, or tightly-coupled MIMD machines. Loosely-coupled architectures present a new challenge.

- The search algorithms described in the reference books are mainly exhaustive searches, branch and bound or alpha-beta. These programs regular structures and are well suited for parallel implementation, especially if their data size is small. Unfortunately, such regularity is seldom seen in AI problems, which require more sophisticated pruning techniques and have to deal with bulky data in huge search spaces.

In the following sections the generic problems, the task partitioning, load balancing and data spreading are discussed in detail. Before that, in section 2, we give some more information and the performance measures of the Multi-PSI, so that the reader may have a better understanding of the subsequent implementation choices. Section 3 describes these notions in the case of exhaustive search. In section 4 the pruned search is studied and additional requirements, needed for efficiency are mentioned. Inductive learning is outlined in section 5 and examples of program run are commented on section 6. Section 7 concludes the paper and draws some hints for future research.

[1] This notion is defined in section 5.2

2 The Multi-PSI/V2 and granularity

The target machine of this study is a Multi-PSI/V2, with up to 64 processors connected by an 8×8 network. Each processor has an 80-Megabyte memory. Running our programs on experimental measurements, gave, on average, the following performances:

- One processor performs 30000 reductions per second.
- The transmission network transmits 2000 atomic objects or pointers per second between two processors, whatever their location.

The ratio $\frac{reductions}{transmission} = 15$ sets some limits for the granularity; in other words, if it takes less than about 30 reductions to generate one byte of data, it is faster to generate this data locally than to transmit it.

According to these measurements, the transmission time of the data between processors cannot be neglected, especially in our problem carrying bulky data. Coarse-grain parallelism minimizing the communication flow in the machine will be the guideline of this work.

3 Exhaustive search

Exhaustive search is quite an easy and well-known problem. In this section we describe its most straightforward implementation using an Or-parallelism and static load balancing. The task partitioning between processors is given, then the load balancing and the data spreading are detailed, with some considerations on granularity, size of the data and number of processors.

3.1 Task partitioning

Exhaustive search in an OR-tree can be split to be performed by several processors in parallel. If the tree shows a regular structure it is easy to split it into same-sized subtrees, and balance the work to be performed amongst the processors.

3.2 Load balancing

3.2.1 Static

Since exhaustive search problems allow prediction of the amount of work to be done by each of the processors, it is possible to use static load balancing to allocate all the tasks before starting the job. This method is simple and efficient, well adapted to performing load balancing even when dealing with bulky data.

3.2.2 Dynamic

Dynamic load balancing performs, in theory, a more accurate job, by allocating tasks to idling processors. Nevertheless, dynamic load balancing greatly increases the number of communications in the machine, and thus loses efficiency when used with bulky data.

3.2.3 Implementation

Because we work with bulky data on a loosely-coupled architecture, static load balancing turns out to be the best solution.

3.3 Data spreading

For a simple OR-parallelism scheme, there are two common ways to spread the data towards the processors: the more simple one is list spreading, the more sophisticated form is tree spreading.

3.3.1 List spreading

The task is spread on a list of processors, and the results are returned to the $processor_0$, see Figure 1. The time of data broadcast is proportional to the number of processors. Overhead besides sheer transfer is low. This solution is simple and efficient, as long as the final gathering of the data on $processor_0$, represented as a black dot, can be neglected, and the number of processors is small.

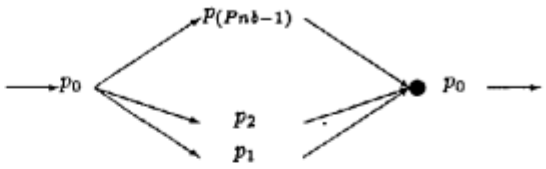


Figure 1: List spreading

3.3.2 Tree spreading

The task is spread on a binary tree of processors. The data diffusion time grows as $log_2(Pnb + 1)$, Pnb being the number of processors. Data gathering is done locally, with three results — its own and the two children's — and takes advantage of the parallelism, see Figure 2. Consequently the size of data-output is reduced as soon as possible, gaining in transfer time. The overhead of tree spreading is higher than that of list spreading.
Tree spreading permits to perform the heavy operation of data gathering (again represented by the black dots) in parallel. As a side effect, it reduces the amount of data to be transmitted.

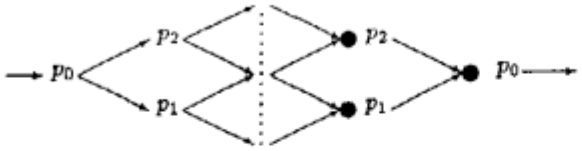


Figure 2: Tree spreading

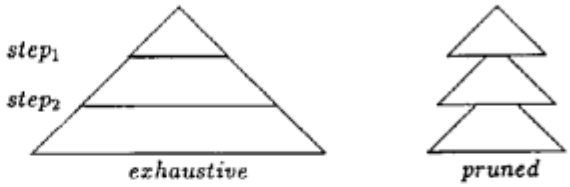


Figure 3: Stepwise pruning in sequential search

3.3.3 Implementation

Both solutions, list and tree spreading are implemented in the FLIB library [2], and have been compared. As long as the merging of processors output is a short operation, list spreading may be used. However, tree spreading is superior when dealing with a high number of processors and bulky data, thanks to the faster data-diffusion through the machine.

4 Pruned search

Exhaustive search shows quite interesting performance, almost linear speedup, unfortunately it's operating domain is restricted to toy problems. In other cases, one has to add some constraints to prune the search as early as possible, controlling the combinatorial explosion. As a consequence, the algorithm loses some of its properties, namely the regular structure well suited to OR-parallelism. We have to examine the repercussions to determine whether the algorithmic implementation still holds.

4.1 Stepwise pruned search

One class of search algorithms is to find some *best path, or cost* in the search space, where best means the minimum of a function. If this function proves to be locally consistent, linear and increasing, a *local cost* may be calculated at each step of the algorithm. Since

$$cost_j = min\,(cost_i + path(i,j))$$

the searches are cut stepwise, by propagating only the *set of local best costs*. As this operation is done easily in dynamic programming, it requires, in a more general case, sorting in order to detect all equivalent elements and keep the "best" ones only. A schematic representation of stepwise pruning is drawn in Figure 3. The more steps in the search, the higher the gains of the pruned search, compared to the exhaustive one.

4.2 Task partitioning

Task partitioning of a pruned algorithm introduces two antagonistic phenomena:

- Because our load distribution is an OR-tree, each of the processors sorts its data locally to find the *local set of best cost*. The intersection of these sets is not empty, so redundant searches are led by several processors.
- The complexity of pruned search is always higher than linear complexity, so splitting the tasks brings hope for super-linear speedup.

Both phenomena are illustrated in Figure 4, where the search is performed independently on two processors. The additional work resulting from local pruning rather than global pruning, is represented by the striped area, we observe that the search space is growing on each step. Conversely, local sorting requires less than half of the work on each processor, thanks to its complexity in $nlog(n)$ for the most efficient sorting algorithms.

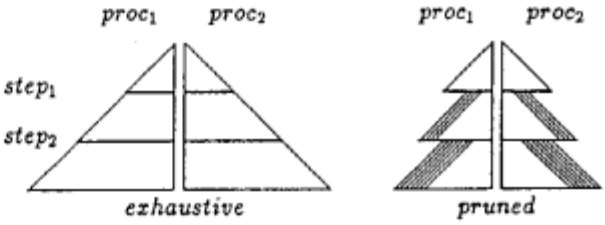


Figure 4: // pruning and task partitioning

4.3 Load balancing

The pruning principle relies on content of the data, and breaks the nice regular structure of exhaustive search. The naive static load balancing no longer guarantees good load balancing, it sets just some limits for the worst case. The perversion of this method, by pruning is shown in Figure 5 where $proc_2$ has much more work to perform than $proc_1$.

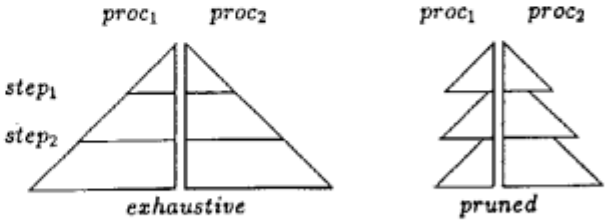


Figure 5: // pruning and load balancing

4.4 Can efficiency be achieved ?

Pruned search in a large search space, with bulky data is a difficult problem especially running on a loosely-coupled parallel computer. Even if hopes for super linear speedups are allowed, two major difficulties will doom them:

- Independent search on the processors implies redundant searches. Communications overcome this problem, but the machine is slowed down by communications.
- Load balancing becomes critical.

The first point is related to the algorithm and the data, and is unavoidable. The whole problem comes from the balance between the price of the communications and the overwork to be done otherwise. An experimental approach is the only solution if no complexity calculations are available. For the second point, the load balancing described in Figure 5 is far from being ideal, however the use of dynamic load balancing [6] seems tedious if used with bulky data. Severe slowdown may result. In the next paragraph we propose an improved version of static load balancing, better suited to deal with bulky data.

4.5 Improved load balancing

The basic OR-tree load balancing consists in cutting the first step into Pnb equal subsets, and process locally on each processor the product between elements. Each processor performs:

$$PE_1 = s_{00} \times s_1 \times \cdots \times s_d$$
$$PE_2 = s_{01} \times s_1 \times \cdots \times s_d$$
$$PE_3 = s_{02} \times s_1 \times \cdots \times s_d$$
$$PE_4 = s_{03} \times s_1 \times \cdots \times s_d$$

See Figure 6. This technique leads to unbalanced trees if some of the spaces are hollow, furthermore, if $step_0$ is constituted by a few elements only, and Pnb is big, the original balance between processors is difficult to obtain, as a consequence of the granularity at this level.

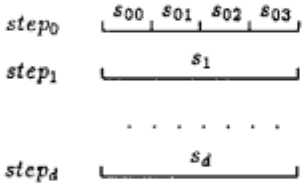


Figure 6: Naive load balancing

An improved load balancing is obtained by splitting the search spaces at each step. It favors a better balance, since the algorithm does not rely on a unique step homogeneity. The decomposition of the search space is performed by decomposing Pnb, and the size of each step, into products of prime numbers. The ideal decomposition of the search space is given by matching Pnb's decompositions with the step ones. If ideal decomposition does not exist, the program searches the suboptimal decomposition, minimizing the differences between processors.

$$PE_1 = s_{00} \times s_{10} \times \cdots \times s_d$$
$$PE_2 = s_{00} \times s_{11} \times \cdots \times s_d$$
$$PE_3 = s_{01} \times s_{10} \times \cdots \times s_d$$
$$PE_4 = s_{01} \times s_{11} \times \cdots \times s_d$$

Figure 7 shows the same example as previously. This time, the two first steps are split.

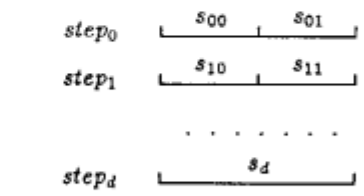


Figure 7: Improved load balancing

5 A case study: Inductive learning

5.1 Description

Inductive learning is an important tool in machine learning. It faces severe restrictions due to its high computational complexity. Inductive learning may be defined as follows:

Suppose we have a set $O = \{o_1, o_2, \ldots, o_l\}$ of objects, each represented by a vector of $j = 1, \ldots, n$ attributes in a domain D_j of values, $o_i = (a_1, a_2, \ldots, a_n)$. For a given number k, we try to find an optimal partition of the whole set O into k classes of objects. Here, these classes are described by a single conjunctive concept and each class has a logical description which is disjoint from other classes. Optimality is defined by a sparseness criterion.

A rough sketch of the clustering algorithm is shown in Table 1. The algorithm begins by choosing k elements at random in O. In step 4, it generates the partitions of D_j containing a_{ij}. To do so, the algorithm exhaustively develops the partitions of D_j by applying the $\mathcal{P}$ function, then it applies the $\mathcal{F}$ function, filtering each element of the partition set, keeping only the ones containing a_{ij}. In the step 5 the direct product of these sets yields the collection of complexes $comp_i$, which are generalizations of the elements o_i $i = 1 \ldots k$ from step 1. The direct sum is then performed in step 6, between the collections of complexes to get the k-complex collection. Finally, the best of the k-complexes is chosen as the partition of O, having the lowest sparseness. The algorithm then chooses a new set of k elements, one in each partition and jumps to 2. The work is stopped when the sparseness of the best k-complex converges

towards a stable result.

```
1 choose k elements o_i ∈ O
2 for each o_i = (a_i1, a_i,2, ..., a_in), i = 1...k
3       for each a_ij, j = 1...n
4             part(i,j) = F(P(D_j), a_ij)
5       comp_i = ×_{j=1}^{n} part(i,j)
6 kcomp = ⋃_{i=1}^{k} comp_i
7 best k-complex choice
8 choose k new elements goto 2
```

Table 1: Outline of the algorithm

An optimal sequential and a parallel version have been implemented, the detail of them are described in [1]. Inductive learning is performed by two pruned searches, the first is to find the complexes $comp_i$ in steps 2 3 4 5, the second to calculate the k-complexes *kcomp* and to choose the best one, in steps 6 7. The parallel implementation performs these searches in an OR-parallelism. It is important to notice the size of the data generated by each step of this program.

5.2 The bulky data

The complexity, as well as the size of the data generated by the program are not yet modeled, let us give some orders of magnitude. Admit the first step deals with data of k bytes, then the worst case analysis gives the following data-sizes in the steps:

1 data-size=k bytes

5 data-size=$k\ 2^{\left(\sum_{j=1}^{n} |D_j|-1\right)}$ bytes

6 data-size=$2^{\left(k\ \sum_{j=1}^{n} |D_j|-1\right)}$ bytes

7 data-size=k bytes

Thus, to run this algorithm on a parallel machine, we have to transmit bulky data between processors, to merge their results. The first pruned search, steps 2 3 4 5 gets very small data, item 1 but generates bulky one, item 5, to be transmitted to the second search in step 6 7. The output of this step is again of small size, item 7.

Let us illustrate our words by the examples later used for the performance measures. The results from the step 5 takes respectively for *ex1*, *ex2* and *ex3*, 49, 393 and 24 Kilo-Bytes. Other examples transmit several Mega-Bytes. Data transmission cannot longer be neglected.

Last but not least, notice that the peak memory consumption is done in item 7. As this operation is done in OR-parallelism, the algorithm fully uses the 5 Giga-Bytes memory of the 64 processors Multi-PSI. As a consequence, our parallel algorithm can handle much larger problems than the sequential algorithm. We do not take advantage of this in the following, since our point is to compare sequential and parallel pruned search algorithms.

5.3 Expected speedup

Since our algorithm prunes its searches, its behavior will be slightly changed in the parallel implementation. This pruning is then done locally on each processor (data communication would cost too much). As a consequence, the processors will carry out some redundant work. It is important to establish measures of this phenomenon to set some limitations to the expected speedup.
An experimental study permitted us to characterize the behavior of the local pruning in Figure 8. It shows that the amount of processing done by one processor — measured with the number of reductions r and represented by the *Experimental* curve — increases as:

$$r = -2.3s^2 + 479s - 449$$

with s the size of the vector $step_0$ to be processed. To obtain a 10 times speedup; that is to divide r by 10, requires that we reduce the size of the input vector from 75 elements to 5 elements. This in turn imposes the use of 15 processors.

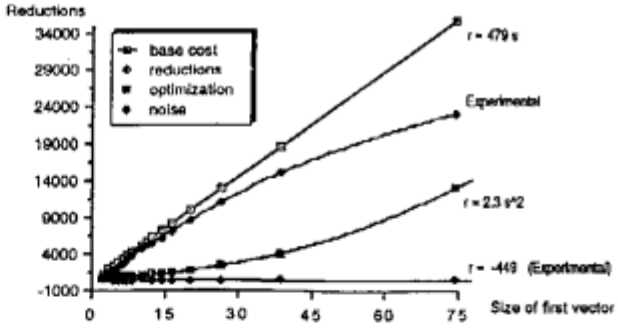


Figure 8: Pruning in parallel

5.4 Explanation

Inductive learning is performed in parallel; nevertheless, result merging is done sequentially or with a lower-level parallelism. To understand this algorithm in a deeper way, we have to separate the parallel computing from the sequential result merging. Figure 9 gives a measure of the parallel work *work*, and the sequential data merging *merging* gathering the previous results. The values are given in Kilo-reductions, for the sum of the processors. Super-linear speedup could be expected, since the number of reductions to be performed for *work* decreases dramatically. In contrast, *merging* increases slightly since it has to merge more and more results, because the local pruning is less efficient than global pruning. The *merging* is very heavy

in inductive learning, and turns out to be the bottleneck in speedup; with 6 or more processors, *merging* becomes preponderant to *work*. It is hard to overemphasize the importance of data merging in such kind of algorithms. We studied two implementations, the

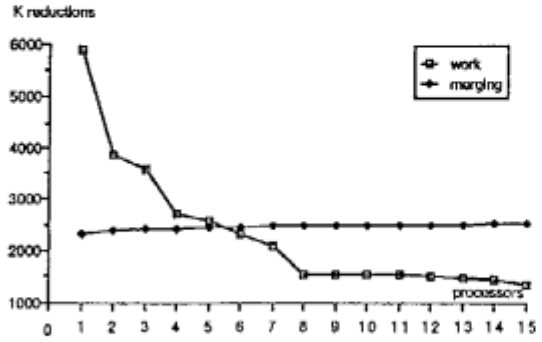


Figure 9: Total number of reductions

first one is the list spreading (cf. 3.3.1) which performs the merging sequentially. The second one, tree spreading (in 3.3.2) performs this operation in parallel on a binary tree. The theoretical speedup of both solutions is drawn in Figure 10. List spreading gets a peak speedup of 3.16 whereas tree-spreading reaches 12.9 . Notice that this function evolves stepwise, according to the depth of the tree of processors merging the data. Compared to Figure 8, the latter estimations show better speedups, because they consider an optimal load balancing between all the processors, as well as an ideal synchronization, in contrast, Figure 8 relies on real measures, with uneven load balancing, despite our efforts.

All this goes to show that an experimental approach is needed to write such kinds of programs, in order to discover the bottlenecks and to choose the most efficient implementation.

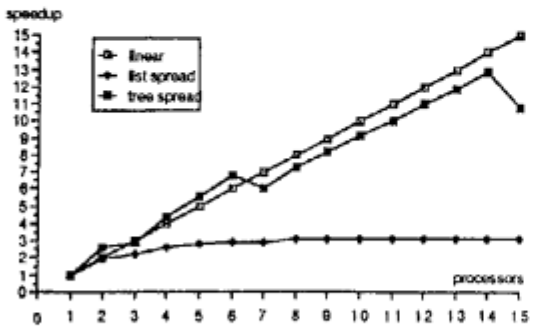


Figure 10: Estimated speedups

6 Results

The following results relate to executions of the sequential and the parallel program. Both versions run on the Multi-PSI/V2 the sequential version run locally on one processor. We evaluate the speedup factor by using up to 15 processors in parallel.

6.1 Sequential versus parallel implementation

The optimized sequential version of the algorithm runs Michalski's train example [7] in 164 seconds. It consists in classifying into two classes the 10 objects issued from a $2^{2\times26}$ large search space. The algorithm takes full advantage of the pruning we introduced.
As shown in Table 2, the parallel algorithm running on one processor has performance very close to that of sequential implementation. The overhead due to spreading, merging and transmitting the data is small compared to the calculation time.

time [s]	sequential	parallel[1 processor]
ex1	720	731
ex2	855	858
ex3	242	247

Table 2: Raw speed

6.2 Performance in parallel

Figure 11 shows the speedup of the same examples, run in the parallel environment. The speedup has been calculated by referring the results to the performance of the sequential version. *Ex1* did a very good job, attaining a tenfold speedup with 15 processors. The two other examples exhibit inferior speedups, only a factor of 7 and 6.2. The speedup curve marks some steps.

The different speedups may be explained by the sparseness of the search space. In *ex1*, the sparseness of the solutions in the research space is higher than in the other examples, furthermore, the distribution of the solutions seems to be homogeneous. These are the best conditions, according to our load distribution. By observing the activity of the processors, we noticed an equal activity for the processors working in *ex1*, whereas in *ex2 ex3*, the load balance turns out to be approximate, thus the lower speedup rates.

7 Conclusion

In this paper we highlight some difficulties when programming pruned researches on a loosely-coupled parallel computer. We show the severe limitations in speedup facing this family of algorithms: first the problems of local pruning bring either, overwork to be performed or supplementary communications, second an additive data merging operation turns out to be very

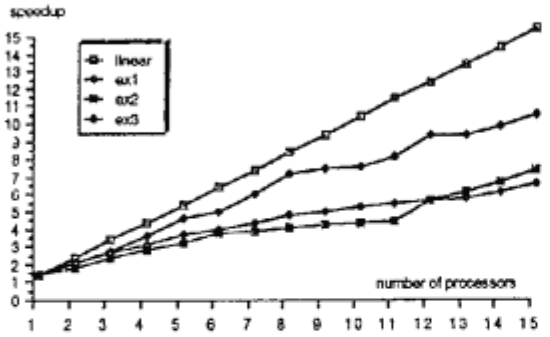


Figure 11: Speedup ratio

heavy, and third it increases the difficulty of load balancing. A single but striking example of pruned research, inductive learning, was implemented in both sequential and parallel versions. The execution on a Multi-PSI/V2 showed the efficiency of the parallel version, the top speedup ratio obtained was 10, with 15 processors. As emphasized, the speedup can hardly be higher. Running this program on the 64 processor machine would certainly require a better load balancing. Dynamic load balancing seems to be an attractive solution, unfortunately it introduces additional communications which doom the performances of the Multi-PSI/V2. Communications are the bottleneck of the Multi-PSI/V2, it urges to improve them.

Acknowledgment

I would like to thank Nobuyuki Ichiyoshi for his comments on the first draft.

References

[1] B. Burg. Inductive learning in a parallel environment. In *France-Japan Artificial Intelligence and Computer Science Symposium*, 1989.

[2] B. Burg and D. Dure. FLIB user manual. Technical Report TR 529, ICOT, 1990.

[3] K.M. Chandy and J. Misra. *Parallel Program Design*. Addison-Wesley Publishing Compagny, Inc, 1988.

[4] T. Chikayama, H. Sato, and T. Miyazaki. Overview of the parallel inference machine operating system (pimos). In *Proceeding of the International Conference on Fifth Generation computer Systems*, 1988.

[5] J.P. Fishburn. *Analysis of Speedup in Distributed Algorithms*. UMI Research Press, Ann Arbor, Michigan, 1984.

[6] M. Furuichi, K. Taki, and N. Ichiyoshi. A multi level load balancing scheme for or-parallel exhaustive search programs on the multi-psi. In *PPOPP*, 1990.

[7] R.S. Michalski, J.G. Carbonell, and T.M. Mitchell. *Machine Learning, an Artificial Intelligence Approach volume II*. Morgan Kaufmann Publishers, Inc., Los Altos CA, 1986.

[8] R.S. Michalski, R.E. Stepp, and E. Diday. A recent advance in data analysis. clustering objects into classes characterized by conjunctive concepts. *Progress in Pattern Recognition L.N. Kanal and A. Rosenfeld (editors), North Holland Publishing Company*, 1981.

[9] 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.

[10] M.J. Quinn. *Designing Efficient Algorithms for Parallel Computers*. McGraw-Hill Book Company, 1987.

Aアルゴリズムと横型探索の試み

星田 昌紀 六沢 一昭 新田 克己
ＩＣＯＴ第７研究室 沖電気工業(株) 総合システム研究所 ＩＣＯＴ第７研究室

1 はじめに

Aアルゴリズムは一良解探索の代表的な手法である。一方、最適解探索及び全解探索の手法のひとつとして横型探索がある。本稿では、Aアルゴリズムと横型探索について説明し、８／１５パズル及び牛パズル(箱入り娘)への適用例をプログラミング中心に紹介する。

2 Aアルゴリズム

状態を評価し、評価値の高い状態を優先的に探索する方法である。評価値 f は以下の式で求める。

f = g + h

ここで g, h はそれぞれ

g : 初期状態から、その状態に至るまでにかかったコスト

h : その状態から、目標状態に至るまでにかかることが予想されるコスト

である。h はあくまで予測値であり、その値が必ずしも正しいとは限らない(ヒューリスティック関数とも呼ばれている)。

3 横型探索

重複のチェックをしながら、初期状態から n 手で到達できる状態の集合 S_n を S_1 から順に $S_2, S_3, S_4 \ldots$ と求めていく方法である。このため最適解を求めることができる。

この方法では、状態を生成するたびにそれまでに見つかっているすべての状態を対象として重複をチェックすることが必要なため、手が進むにつれて記憶すべき状態の数が増え重複チェックのコストも高くなる。従って以下のことが起こる。

「生成される状態は減ってきたのに手の進みかたが速くならない。」

「総状態数が非常に多いので、状態を記憶するメモリが足りなくなって問題が解けない。」

ところが、「状態 a から状態 b へ１手で行けるならば、状態 b から状態 a へも１手で行ける」場合は重複チェックは１手手前までで よい。即ち S_n から S_{n+1} を求める場合は、S_n と S_{n-1} についてのみ重複をチェックすればよい。この理由を以下に簡単に述べる。

> S_{n-2} から１手で到達できる状態はすべて S_{n-1} あるいは S_{n-3} に含まれる。このため S_n から１手で到達できる状態は S_{n-2} には含まれない($S_{n-3}, S_{n-4}, \ldots, S_0$ にも含まれない)。従って、S_{n+1} を求める場合は S_n と S_{n-1} だけについてチェックすればよい。(図１参照)

このため、状態を記憶するのに必要なメモリの大きさや重複チェックのコストは前２回の状態数で決まり、

「生成される状態の減少にしたがって手の進み方も速くなった。」

「総状態数は非常に多いが、各ステップにおける状態数は少ないので問題が解けた。」

ということが起こる。

4　８／１５パズルへの適用例

８／１５パズルでは、探索の進み具合と無関係にひとつの状態から３つ前後の状態が必ず生まれる。従って枝分かれは絶えず多く状態の増え方が激しい。

このため、Ａアルゴリズムを用いて一良解探索を行なう場合は、探索を進める状態の選択を誤ると (f の性能が悪いと) 解が (なかなか) 求まらない恐れがある。また横型探索では、(たとえ重複チェックが１手手前までで よくても) 最大状態数が非常に大きいと状態を記憶するメモリが足りなくなってしまい解が求まらない恐れがある。

4.1　Ａアルゴリズム

状態の評価

g, f は以下のように設定した。

g ： 初期状態からの手数

h ： 終了状態からの各駒のマンハッタン距離の和

重複のチェック

重複のチェックは完全には行なわず、「前回の状態に戻るような“手”(後戻り) だけは避ける」に留めた。即ち、状態ａ → 状態ｂ → 状態ａのような手のみを除外した。この理由を以下に述べる。

- 後戻りを避けることは完全な重複チェックに比べてかなり低いコストで実現できる。マルチＰＳＩのようなネットワーク型並列計算機ではこの差は非常に大きくなる。
- 後戻りを避けるだけで、比較的効率の良いチェックが期待できる (と思えた)。例えば状態ａから後戻りすることなく再び状態ａに戻るには１２手必要とする。また状態ａから２通りのルートで状態ｂに到達するのにもそれぞれ１手と１１手、２手と１０手、...、５手と７手、６手ずつ必要になる。

優先探索の制御

高い評価値を持つ状態を優先的に探索することは、状態をゴールとしＫＬ１処理系の優先度制御機構を使うことで実現した。即ち、与えられた状態の探索を行なう (与えられた状態から探索を始める) ゴールを作り、それを評価値に基づく優先度を付けて fork する方法を用いた。この方法には以下の利点がある。

- 処理系の制御機構を使っているので、優先探索が低いコストで実現できる。
- 状態をゴールとしているので、並列実行が容易 [1] である。新しい仕組みを作ることなく、ゴールを他のプロセッサへ投げるだけで探索処理の分散ができる。

一方、ゴールは一旦 fork してしまうと実行状態にならない限り優先度の変更や abort ができない。このため、以下のことの起こる可能性がある。

1. 評価値の低い状態の探索を行なうゴールには低い優先度を付けるので、評価値が非常に低いゴールは fork されても探索が終わるまで (解が求まるまで) 実行状態にならない。
2. 探索が進むに連れ、そのようなゴールが増加する。
3. ゴールを格納するメモリが足りなくなり、探索が続行できなくなる。
4. メモリを確保するため、低い優先度のゴールは見込みがないものとして abort してしまいたいが、それはできない。

[1] 後戻りを避けるだけのチェックしか行なわないことを仮定している。完全な重複チェックを行なうと、通信処理のコストが非常に大きくなってしまう。

実行結果

以下に１５パズルのいくつかの実行結果を示す。

解の長さ	実行時間	リダクション数
25 手	18.4 秒	約 90 万
30 手	40.0 秒	約 210 万
40 手	945.3 秒	約 4500 万

処理系の優先度制御機構を使わない方法

ＫＬ１自身で優先度制御を記述することができる。図２に優先度制御を行なうプロセス (prioStack/2) の例を示す。prioStack/2 のストリームにコマンド (push, pop) を流すことによって、「評価値付きの状態の挿入」、「最高評価値を持つ状態の取り出し」ができる。

このようにＫＬ１自身で優先度制御を記述すると、処理系の優先度制御機構を使った方式では実現できなかった「優先度の変更」や「abort」が可能になる。しかしこの方法には以下の欠点がある。

- 優先度制御のコストが高い。
- 並列実行の仕組みが以下に示すように複雑になる。
  - prioStack プロセスを各プロセッサに作ることが必要となる。これはプロセッサを意識したプログラミングである。
  - 探索処理の分散は prioStack プロセスが持つストリームへコマンドを流すことによって行なう (単にゴールを投げるだけでは不可能) ので、各プロセッサはすべての prioStack プロセスのストリームを持たなくてはならない。

4.2 横型探索

直径 (最短手数の最大値) を求めることを試みた。重複チェックにはハッシュ表を用いた (図３参照)。可能な状態数は以下に示すように非常に多い。

８パズル：$9!/2 \approx 180K$

１５パズル：$16!/2 \approx 10^{13}$

このため、「８パズルの直径は求まるが、１５パズルは無理だろう。」と思って実行したところ、予想どうりであった。１５パズルはメモリ不足により途中で続行不可能となった。実行結果を図４、５に示す。

5 牛パズル (箱入り娘) への適用例

牛パズル (箱入り娘) は８／１５パズルと比較して駒の動きに制約が多い。後戻り以外に手がないこともある。このため (８／１５パズルと比較して) 枝分かれは少なく状態の増え方は緩やかである。一方、

駒の数 (10 個) ＞ 駒の種類数 (4 種類)

であるため、(８／１５パズルと比較して) 状態の重複することが多いと思われた。

5.1 Ａアルゴリズム

８／１５パズルと同様に重複のチェックは「後戻りだけは避ける」とした。h として以下を試みたが、解くことはできなかった。

- “牛”の位置が下であるほど (出口に近いほど) h が小さくなるようにする。
- “牛”が動いた時 h を小さくする (これは“手”に対する評価である)。
- 空白が”牛”の近くにある時 h が小さくなるようにする (狙い：”牛”の動ける機会を増やす)。
- 空白が隣り合っている時 h が小さくなるようにする (狙い：長方形の駒の動ける機会を増やす)。

「後戻りだけは避ける」程度の重複チェックしか行わないＡアルゴリズムでこのパズルが解けるような h を作ることはかなり難しいようである。

5.2 横型探索

解くことができた。結果を図6に示す。総状態数が 25,000 ～ 30,000 程度なのは意外であった (はるかに大きい数を予想していた)。8／15パズルと異なり、状態数の変化が「増加 → 減少」でない点は興味深い。また牛パズル1と2では状態数の変化の様子が大きく違うのも面白い。

プログラムにおける状態の表わし方を工夫した結果、1ワード (32 ビット) で表現することができた (図7参照)。重複のチェックには8／15パズルと同じくハッシュ表を用いている。

- 状態を1ワードで表現できた。
- 最大状態数が 700 程度と少なかった。

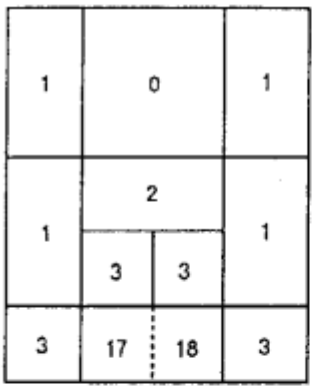
ことから、記憶すべき状態は少なく探索のコストも低くなり、効率良く解くことができたと思われる。

6 おわりに

横型探索が有効である問題も存在することに気がついた。今後は並列実行させることを試みたい。

- ある状態から次の状態を生成するコストが大きい。
- 各ステップにおける状態数が充分多い。

が成り立つ問題ならば、探索の局所性を生かした分散戦略を考えられれば並列実行によるスピードアップは充分期待できる。



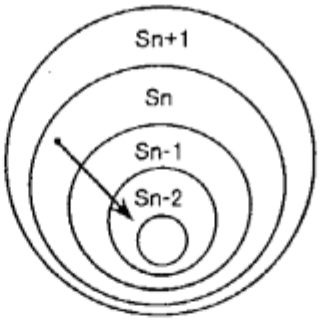


図1 Sn からSn-2へ行くのには2手必要

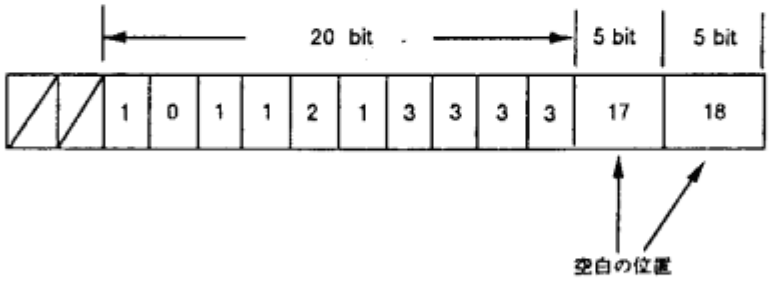


図7 牛パズルの状態の表現

```
% prioStack(Rs,StackV,Top) : 状態の優先度管理を行う
%   Rs : 状態の出し入れの要求ストリーム : push pop を受け付ける
%   StackV   : 要素として、同一優先度の状態のリストを持つベクタ
%   Top      : 最高優先度を指すポインタ(数値が小さいほど優先度が高い)

prioStack([],_,_):- true|true.
% 状態の挿入
prioStack([push(N,St)|Ss],StackV,Top):- Top < N |
%   St : 挿入する状態
%   N  : 優先度
        set_vector_element(StackV,N,Stack,[St|Stack],NewStackV),
        prioStack(Ss,NewStackV,Top).
prioStack([push(N,St)|Ss],StackV,Top):- Top >= N |
        set_vector_element(StackV,N,Stack,[St|Stack],NewStackV),
        prioStack(Ss,NewStackV,N).
% 状態の取り出し
prioStack([pop(St)|Ss],StackV,Top):- vector_element(StackV,Top,[])|
%  ^St : 取り出された状態
        prioStack([pop(St)|Ss],StackV,^(Top+1)).
prioStack([pop(St)|Ss],StackV,Top):- vector_element(StackV,Top,[_|_])|
        set_vector_element(StackV,Top,[St|NewStack],NewStack,NewStackV),
        prioStack(Ss,NewStackV,Top).
otherwise.
prioStack([pop(St)|Ss],StackV,Top):- true |
        St = [],prioStack(Ss,StackV,Top).
```

図2　優先度制御プロセス

```
% sift(Is,Ht,^Os) : 重複のチェックを行なう。
%   Is : 重複チェックを行なう状態が入力されるストリーム
%   Ht : チェックの対象となる状態群が格納されているハッシュテーブル(ベクタ)
%  ^Os : 重複していない状態が出力されるストリーム
sift([St|Is],Ht,Os) :- true |
        hash(St,V,WSt), vector_element(Ht,V,Ss,Ht2), sift2(WSt,Ss,Os,Os2),
        sift(Is,Ht2,Os2).
sift([],_,Os) :- true | Os = [].
sift2(St,[],Os,Os2) :- true | Os = [St|Os2].
sift2(St,[St|_],Os,Os2) :- true | Os = Os2.
otherwise.
sift2(St,[_|Ss],Os,Os2) :- true | sift2(St,Ss,Os,Os2).

hash(St,V,WSt) :- true | % 状態(St) に対してハッシュ値を求める(中身は省略)。
%   St  : 状態
%  ^V   : ハッシュ値
%  ^WSt : St がそのまま返る
```

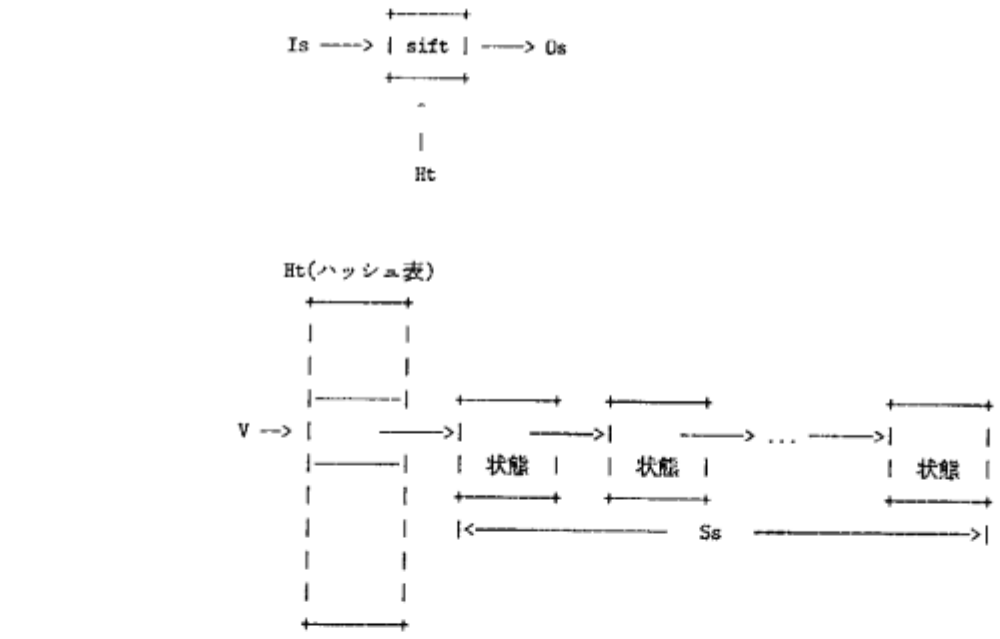
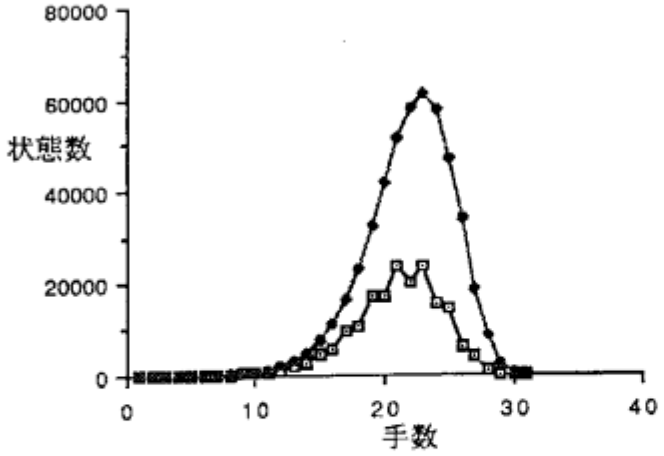


図3　重複チェックプロセス



n	Sn	Sn-1*Op
1	2	2
2	4	6
3	8	12
4	16	24
5	20	36
6	39	60
7	62	104
8	116	186
9	152	280
10	286	456
11	396	716
12	748	1188
13	1204	1862
14	1893	3072
15	2512	4692
16	4485	7536
17	5638	11084
18	9529	16914
19	10878	23396
20	16993	32634
21	17110	41282
22	23952	51330
23	20224	57864
24	24047	60672
25	15578	57178
26	14560	46734
27	6274	34090
28	3910	18822
29	760	8836
30	221	2280
31	2	486

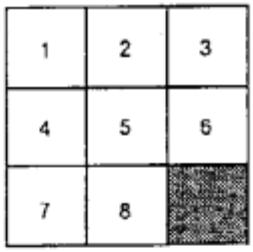
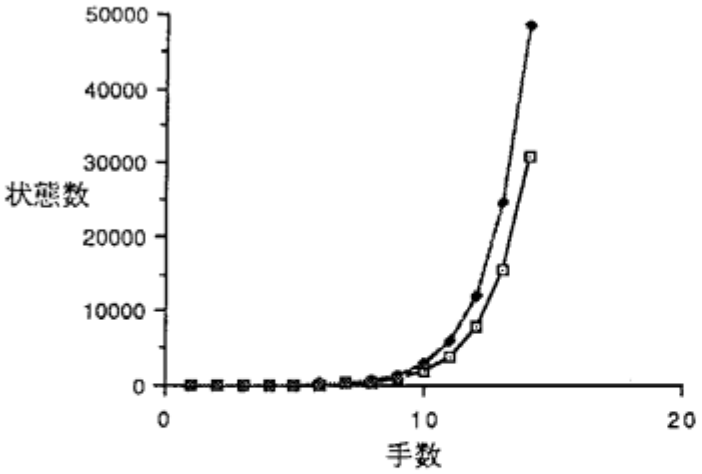


図4　　8パズルの実行結果 (横型探索)



n	Sn	Sn-1*Op
1	2	2
2	4	6
3	10	14
4	24	34
5	54	78
6	107	162
7	212	322
8	446	668
9	946	1420
10	1948	2978
11	3938	6072
12	7808	12182
13	15544	24278
14	30821	48494

メモリ不足により続行不可

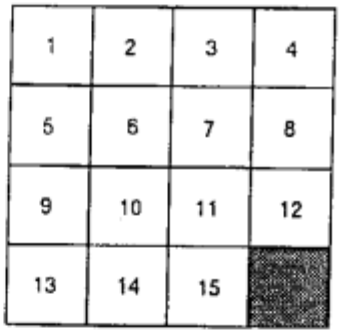
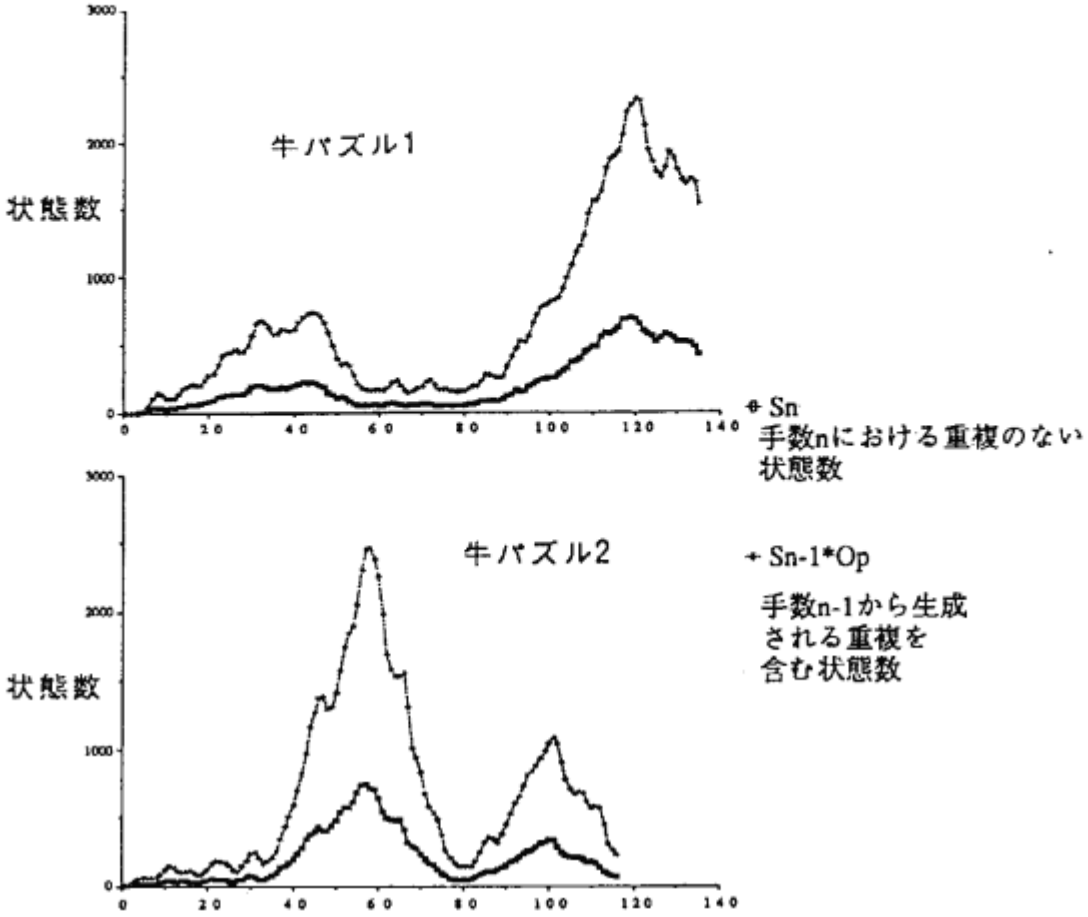
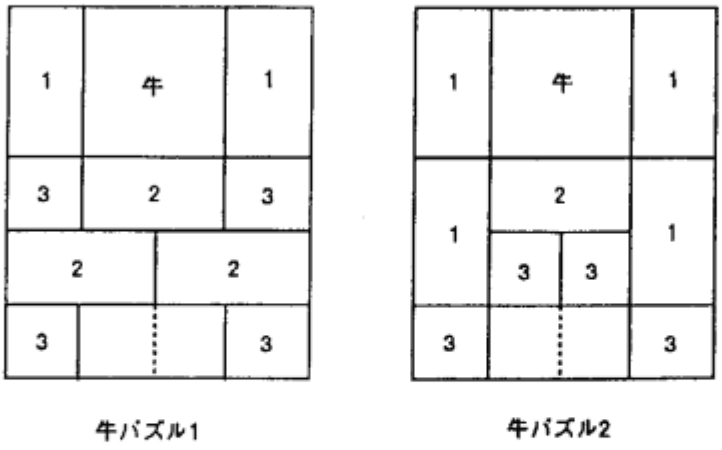


図5　　15パズルの実行結果（横型探索）





牛パズル1　　牛パズル2

図6　牛パズルの実行結果　(横型探索)

ANDOR-IIのマルチＰＳＩ上での実現
(Extended Abstract)

高橋和子　坂本忠昭　竹内彰一
三菱電機(株)中央研究所

要旨

並列問題解決用記述言語 *ANDOR-II* のマルチＰＳＩ上での実現について述べる。並列問題解決システムは問題解決において複数の問題解決機が並列に推論を行ない、かつ、それらの間で問題についての知識を交換し合うような知的協調を可能にすることを目的としており、*ANDOR-II* とそのコンパイラおよび実行系から成る。マルチＰＳＩ上での実現にあたり、コンパイラ自体の並列化とオブジェクト・コードの並列化の双方について検討し実現した。

1　はじめに

ANDOR-II は and,or 両並列性を備えた言語であり、非決定的なものを含む構成要素が互いに影響を及ぼし合いながら動作する系の上での探索・シミュレーションや組合せ問題などに簡明な記述と強力な推論機構を与える。これらの問題では、１つの構成要素の動作の結果が他の要素の動作に影響を与え、しかも全体で同期をとることがないので、複数の要素の影響を受けるものは非常に複雑な動作をする。また、フィードバックのような制御がかかっていると自分自身の動作が再び自分に影響を及ぼすため、動作はさらに複雑なものになる。このような動作系の構成要素に非決定的な要素が含まれると、全体の動作には幾種もの場合分けが生じ、動作のたびにそれは指数的に広がっていく。

ANDOR-II では「色」という概念を導入し、これらの可能世界を独自の色の付いた世界としてとらえる方式を提案する。すなわち同一世界上で実行されるべきゴールやデータにその世界の色を付けて識別子とし、同一色を持つ要素同志のみを計算可能とすることで、複数個の世界の管理を行なう。非決定的な要素のリダクションが起こるたびにデータに付加された色は細分化されていく。このメカニズムはストリーム通信を主とするＫＬ１の構造にうまく照合する。

ANDOR-II で記述されたプログラムはＫＬ１へコンパイルされマルチＰＳＩで実行される。マルチＰＳＩ上での実現に当たり、オブジェクトコードの並列化とコンパイラ自身の並列化の双方を検討し実現した。前者に関しては *ANDOR-II* にプロセッサ割り付けをユーザが陽に指定できるようにし、同時にコンパイル時にある程度自動的にプロセッサ割り付けを行なうようにした。また、プロセッサ割り付けに関しては対象とする問題やデータに依存する要素が大きいため、実験的にさまざまな割り付けが行なえるようなユーティリティ機能を提供した。後者に関してはトップレベルにおけるデータの流れる方向と量を中心に解析し、負荷分散を行なった。

割り付けにあたっては

（１）通信データ量の多いプロセス同志は同一のプロセッサに割り付ける。

（２）通信頻度の高いプロセス同志は同一のプロセッサに割り付ける。

（３）小さな処理しか行なわないプロセス同志は同一のプロセッサに割り付ける。

という原則に基づいて行なうようにした。これらの要素は入力プログラムやデータに大きく依存するが、*ANDOR-II* では計算モデルとその実現形式の特徴からある程度すべてのオブジェクトコードに共通な方式で負荷分散することができる。

2 ANDOR-II 概要

ANDOR-II の述語はＡＮＤ述語とＯＲ述語に分類される。前者はガード付きの節で定義され、後者は無ガード節で定義され、１つの述語についてこの２つが混在することはない。ただしボディゴールとしては両述語の混在を許す。

割り付けに関してはユーザがゴールを Goal @ Where と記述することにより陽に指定できる。ただし Where のデータ型は以下の通りである。

```
Where ::= Direction | PrcNmbr
Direction ::= Direct | ( Direct, Direction )
Direct ::= rt | lt | up | dn
PrcNmbr ::= integer
```

指定の仕方は以下の２つの理由から絶対的なプロセッサ番号ではなく、プロセッサ同志の相対的な位置関係で指定できるようにした。また、絶対的なプロセッサ番号による指定も可能なものとする。特に指定のないものは親ゴールと同一のプロセッサで実行されるものとした。

(1) ゴールを実行するプロセッサの指定をする場合、‘近く’のプロセッサを指定する場合が多く、プロセッサの番号で指定する場合は少ない。

(2) プロセッサ構成に依存せず指定可能な方が望ましい。

Goal'@'Where の意味は‘親ゴールを実行したプロセッサから Where だけ移動したプロセッサでそのゴールを実行せよ’ということである。Direct ではそれぞれプロセッサの構成に基づき、rt（右）、lt（左）、up（上）、dn（下）方向への移動を示す。また PrcNmbr としてとれるのは０から（構成台数－１）までの整数値である。プロセッサの構成は常にメッシュ状であり、位置関係は実際の構成に基づく位置をあらわす。構成上の左右／上下の端点はそれぞれ同一のものとみなす。特に指定がなければ（システム指定がなければ）親ゴールを同じプロセッサで実行するものとする。

3 計算モデル

ANDOR-II で記述されたプログラムは and,or 両並列性を実現するようにコンパイルされる。即ち、論理積の関係にあるすべてのゴール・リテラルのリゾリューションを並列に実行するというＡＮＤ並列性と、ＯＲ述語の論理和の関係にある節のうち適用可能なすべての節によるゴール・リテラルのリゾリューションを並列に実行するというＯＲ並列性を有する。従ってＯＲ述語に関しては全可能性が保持され、各可能性はそれぞれ独自の世界に分岐する。これらの世界には独自の色が付けられ各々独立して推論が行なわれ、結果にはその世界の色が再び付随されて出力される。複数の世界からの出力結果はベクタの形にまとめて流される。このベクタを「色付きベクタ」と呼ぶ。色付きベクタを入力として受け取るゴールにはシェルと呼ばれる特殊カバーが付加される。シェルは色付きベクタを分解して各データを抽出し、データに付随された色の世界でリゾリューションを行なう。１つのゴールに複数の色付きベクタが入力として流れる場合は各入力ベクタから抽出したデータの色の整合性を調べ、同一世界に属するもの同志に対してのみリゾリューションを実行する。*ANDOR-II* の特徴はこのような シェルを使った色付きベクタの処理であり、実

行時におけるボディゴールの入出力データ型によって種々のシェルが存在する。コンパイラの行なう主な処理はシェルの必要なゴールの検出とその型の決定である。

4 コンパイラの処理概要

コンパイラは記述言語 *ANDOR-II* で記述されたソースプログラムを読み込んで、ＫＬ１（プロセスアロケーションの付加されたもの）のコードに変換する。その際 *ANDOR-II* のｏｒ並列をＫＬ１のａｎｄ並列に変換する。変換は上述の色付きベクタに基く手法（S-based）によるが、一部コンティニュエーション法に基く手法（C-based）を採用し実行の高速化をはかる。前者は *ANDOR-II* のａｎｄ並列はそのままａｎｄ並列に変換し、後者は *ANDOR-II* のａｎｄ並列は逐次ａｎｄに変換する。

コンパイラは主として前解析部、解析部、変換部の３つの手続きから成る。まず前解析部でソースプログラムを読み込んでプロセッサ割り付けを取り除いた部分から解析の基本となるデータフローグラフ（DFGs）およびそこに出現する各述語の情報を格納するハッシュテーブル（PL）を作成する。さらに各述語を S-based と C-based かいずれの手法でコンパイルするかを判定してそれぞれの DFGs に分割する。PL は述語／引数の数をキーとする要素の集合であり、述語の種類やシェルの型もすべてここに格納される。解析は PL に格納された情報を使って行なわれ、新たに判定された結果を書き込みながら進行する。

変換部では PL に格納された解析結果を参照して DFGs をＫＬ１のコードに変換する。ソース・プログラムに記述された割り付けに関する部分は分離して ALC として格納され、システム指定の割り付けの情報とともにオブジェクトコードに反映される。実際は OR 述語の実行やシェルの実行などはオブジェクトプログラム内に書くのではなく実行時処理系として実行時に呼び出す。

変換例（ｏｒ述語）

［ソース・プログラム（*ANDOR-II*）］

```
:- or_relation pickup/2.                          % OR-declaration
pickup([X|L],Y) :- Y=X.                           % OR-clause
pickup([_|L],Y) :- pickup(L,Y).                   % OR-clause
```

［オブジェクト・プログラム（ＫＬ１）］

```
pickup_Core(X,Y,w(C),BP,PR) :- true |
   BP=[BPO|BPs],                    % getting the ID of branching point
   BPs={BP1,BP2},                   % dividing branching point stream
   andor:set_Color(C,(c1,BPO),C1),  % refinement of the color
                                    %       for the first clause
   pickup_Core_1(X,Y1,w(C1),BP1,PR),% computation of the
                                    %       first clause
   andor:set_Color(C,(c2,BPO),C2),  % refinement of the color
                                    %       for the second clause
   pickup_Core_2(X,Y2,w(C2),BP2,PR),% computation of the
                                    %       second clause.
```

```
    andor:merge(Y1,Y2,Y).
```

ここでpickup_Coreの引数Kはもともとの引数K加え、色の情報用、カウンタストリーム用、割り付け用の3引数が加わっている。カウンタストリームはOR述語K起因する非決定的な分岐点K独自の番号を割り当てるためのものであり、OR述語を直接／間接K呼ぶ可能性のあるすべての述語K付加される。このストリームは特殊なマネジャKつながっており、並列実行の際複数の箇所で分岐が生じる場合Kマネジャからこれらの分岐点K独自の番号をもらう。

変換例（シェル）

ゴール add(In1,In2,Out) のベクタ入力K対するシェル

［オブジェクト・プログラム（ＫＬ１）］

```
add_MShell_2_1([v(X,Cx)|Xs],Y,Z,w(C0),PR) :-   % input data X with a color Cx
                                                %   C0 is a color of
                                                %   the current world
    true |
    andor:consistent_Color([Cx,C0],R),          % check of color consistency
  ( R=success(C) ->
        add_MShell_2_2(X,Y,Z1,w(C),PR) ;
    R=fail -> Z1=[] ),
    add_MShell_2_1(Xs,Y,Z2,w(C0),PR),
    andor:merge(Z1,Z2,Z).                       % merging solutions
add_MShell_2_1([],_,Z,_,PR) :- true |  Z=[].

add_MShell_2_2(X,[v(Y,Cy)|Ys],Z,w(C0),PR) :-
    true |
    andor:consistent_Color([Cy,C0],R),          % check of color consistency
  ( R=success(C) ->
    add_Core(X,Y,Z0,w(C),PR),       % call of core process
    Z=[v(Z0,C)];                    % solution Z0 is
                                    %    associated with its color C
    R=fail -> Z=[] ),
    add_MShell_2_2(X,Ys,Z2,w(C0),PR),
    andor:merge(Z1,Z2,Z).                        % merging solutions
add_MShell_2_2(_,[],Z,_,PR) :- true |  Z=[].
```

5　並列化について

並列マシン上Kおける実現K当たり、オブジェクトコードの並列化とコンパイラ自身の並列化の2点から検討した。

5.1 オブジェクトコードの並列化

オブジェクトコードでは、ゴールの指定プロセッサへの投げ出しに関してはシェルやプロセスの起動と同様実行時処理系のプロセスをサブルーチンのように呼び出すようになっており、実際にどのプロセッサへ投げられるか（物理的な絶対番号）は実行時に決定される。

5.1.1 割り付けに対する基本方針

割り付けに関してはユーザがソース・プログラムで陽に指定したものと、システムが自動的に行なうものがある。前者は前解析部からデータ ALC で伝播され、後者はユーティリティ機能としてユーザが制御をかけた結果3本の制御ストリーム NoAlcShell, NoAlcOR, Strategy として送られる。

分散の対象となるのはｏｒ述語の実行とシェルの実行部分である。

ｏｒ述語の実行部分では、各ｏｒ世界を分岐させ、結果をマージしているが、分岐された各ｏｒ世界は共通の入力で同時に起動され、しかも互いに影響を及さず独立にリダクションが進む。従って並列性も高く別個のプロセッサに割り付けるのが望ましい。各ｏｒ世界からの結果をマージしている部分は処理自体は大きなものだが、データの通信量も多く、プロセスへのアクセスも頻繁に起こるため、親ゴール（各ｏｒ世界を分岐させているプロセス）と同一のプロセッサで行なう。

シェルはベクタ型入出力から各要素（色の付いた世界）を取り出して色の整合性がとれているものに対してはコアプロセスを呼び、結果をマージする。さらに入出力ベクタの残りの要素に対し再帰的に同等の処理を行なう。各要素に関する処理は互いに独立であり、仕事量も比較的大きい。従ってシェルに関しても別個のプロセッサに割り付けるのが望ましい。また、シェルの中で行なわれる色の整合性チェックのプロセスと各コアプロセスは同時に呼ばれるが、コアプロセスは整合性チェックの結果を受け取るまでサスペンドしており、整合性チェックが終ってから初めて起動される。この2つのプロセスは逐次性が高く同一プロセッサでの実行が望ましい。

ａｎｄ関係にあるボディゴールの実行は負荷分散した方が高効率が得られる場合もあるが、ソースプログラムにおけるデータ依存性に依存する部分が大きく、一概に分散させるのが望ましいとはいえない。コンパイル時にゴール間のデータ依存性を調べる方法もあるが、むしろプログラムによってユーザが指定できるようにする方がよいと思われる。

以上の考察からｏｒの実行とシェルの実行部分は基本的にすべて別個のプロセッサへ割り付けることにする。

しかし、実際はプログラムによっては分散した各コアプロセスの仕事量が少なく、負荷分散による効果が現れない場合も考えられる。そのためのユーティリティとしてｏｒの実行とシェルの実行部分には述語ごとに割り付けを行なわないような制御ができるようにした。

5.1.2 割り付け戦略

割り付け戦略としてはコンパイル時に行なう静的なもの、実行時に行なう動的なものの2種類が考えられる。現在は実現の容易さと実行上の効率を考慮して、以下のような静的割り付けのみを実現している。ユーティリティ機能によりユーザはどの戦略を使うか指定できる。

（1）近傍への割り付け

子ゴールを実行するプロセッサの割り付けの順を親ゴールを実行されたプロセッサの上、右上、右、同一のプロセッサの順とする。

（2）サイクリック割り付け

子ゴールを実行するプロセッサの割り付けの順を親ゴールを実行したプロセッサのプロセッサ番号をN、構成台数をN0とすると、(N+i) mod N0でi=1,2,3,...に対して得られる値を番号とするプロセッサに割り当てる。

（3）ランダム割り付け

子ゴールを実行するプロセッサの割り付けの順をランダムに決定する。

5.1.3 割り付けの実現

割り付けに関する情報はユーザ指定のものは引数ALCとして、システム指定の制御はNoAlcPredsおよびStrategyとして受け取る。NoAlcPredsはシステム割り付けを取ってしまう述語の情報であり、ORに対する割り付けに関するものNoAlcORとシェルに対する割り付けに関するものNoAlcShellがある。Strategyは割り付け戦略である。

①ユーザ指定の部分

ユーザがGoal@Whereと記述したゴールに対しては、このゴールの直前に新たなゴール

```
config:move(_TPrN,Where_i,_TPrM)
```

を加え、このゴール自体は

```
NewBG '@' processor(_TPrM)
```

とする。ただし、_TPrNはヘッドに出現するプロセッサ割り付け用の引数、NewBGはBGの割り付け用の引数を_TPrMとしたものである。

②システム指定の部分

NoAlcPredおよびStrategyに従って割り付けを行なう。

ボディゴールの述語がOR述語pであり、割り付け無しの制御がかからない場合は、このゴールの直前に新たなゴール

```
config:move(_TPrN,Strategy,_TPrM)
```

を加え、自分自身はOR処理用のプロセスを呼ぶ。その際のプロセッサ割り付け用の引数は_TPrM、戦略用の引数にはStrategyで指定されたものする。

シェルに関しても同様の処理を行なう。

さらに、S-basedとC-basedの接合部分ではC-based部分への接続p_Connectを呼び、さらに出力解を出力用のフォーマットに整える処理をするモジュールandorのゴールを呼ぶ。これらのプロセスは独立性が高く、それぞれの仕事量も比較的大きいので以下のようにプロセッサ割り付けを行なう。

p_Connectの直前に

```
config:move(_TPrN,Where,_TPrM),
```

を付加し、このゴール自身は

```
p_Connect(A1,...,An,w(C),BP,_TPrM)@processor(_TPrM).
```

となる。

ただし、BP はカウンタストリーム用であり、述語 p の PLElement の Relation が connect(or), connect(pseudo) の時のみ付加し、w(C) はこのゴールの現在の色を表す。また、_TPrN は親ゴールに出現する割り付け用の変数、_TPrM は新たに導入された変数である。 Where は basic 戦略の場合は up、cyclic 戦略の場合は cyclic、random 戦略の場合は random とする。

5.2 コンパイラ自身の並列化

ANDOR-II コンパイラ・トップレベルのプロセス構成は read_terms, PRA, CDFA, SDFA, CTRA, STRA, write_file の 7 つである。プロセス read_terms は、ファイル InFile からソースコードを読み込む機能を持ち、プロセス write_file は、オブジェクトコードをファイル OutFile に書き込む機能を持つ。PRA は前解析部、CDFA, SDFA はそれぞれ C-based 部, S-based 部の解析部であり、CTRA,STRA はおのおのの変換部である。コンパイラ自身の並列化にあたり、トップレベルにおける 7 つの主要なプロセスの割り付けについてプロセス間のデータの流れ、プロセス間の通信量、プロセスの処理の大きさに関する検討の結果、次のような方針が得られる。

1. プロセス read_terms と PRA との間のデータの量が多く、プロセス read_terms の処理が小さいことから、この 2 つのプロセスは同一プロセッサ上で実行する。

2. プロセス CTRA, STRA と write_file との間のデータの量が多く、プロセス write_file の処理が小さいことから、これら 3 つのプロセスは同一プロセッサ上で実行する。

3. プロセス PRA, CDFA, SDFA については、個々の処理が大きいため、別々のプロセッサに割り付けるものとする。ただし、プロセス PRA と CDFA, SDFA の間、及びプロセス CDFA, SDFA と CTRA, STRA の間にはある程度量の多いデータが流れるため、これらのプロセッサ間は物理的な距離を近くしておく必要がある。

これらの方針に基づいて 7 つのプロセスを 4 つのプロセッサに割り付け、実験した結果、ソース・プログラムのコード量が多いものに関しては負荷分散による効率改善が見られた。

ところで、一般に、コンパイラは逐次処理的要素が強いアプリケーションである。従って、従来のアルゴリズムをもとにした並列化によって、ある程度の実行効率の向上は期待できるものの、それ以上の効率化をはかるにはアルゴリズム自体を並列用に変更しなければならない。しかし、コンパイラの並列アルゴリズムに関しては、検討すべき点が多く残されており今後の課題である。

6 おわりに

現在 and 並列と or 並列を合わせ持つ言語やそれを実現するシステムの研究開発は盛んにおこなわれており or 並列の導入にともない変数の多重束縛の扱いと指数的に増加する世界をいかに効率良く扱うかが主要な論点となっている。

これらの研究と比較すると、*ANDOR-II* の特徴は以下のように述べることができる。まず、記述言語として見ると *ANDOR-II* ではモード宣言と or 関係宣言の他には実行に関するコントロールは一切持たず、さまざまなシェルが制御の機能を持っている。そのため簡明で記述しやす

い言語となっている。また、*ANDOR-II* の実行においては非決定的なゴールが呼ばれるとすぐ実行しており、複数の非決定的ゴールが同時に並列に実行されることもあるため高い並列性が得られる一方、同時に組合せ的爆発を引き起こす要因にもなっている。しかしながら、マルチＰＳＩ上での実現を考えた場合、１ゴールずつ制御をかけて起動しsuspensionを多く引き起こすよりもマルチプロセッサを持つ並列マシンの能力を生かした実現方法の方が我々の目的には適う。

今後の課題として不要な世界の除去による枝刈りの実現と *ANDOR-II* で記述されたプログラムの実行制御や監視を行なう開発環境の設計が考えられる。

参考文献

[1] E.Shapiro, " Systolic Programming: A Paradigm of Parallel Processing," Proc. of FGCS84, 458-470, 1984.

[2] Takeuchi,A., K.Takahashi and H.Shimizu, "A Description Language with AND/OR Parallelism for Concurrent Systems and Its Stream-Based Realization," ICOT TR-229,1987.

[3] Takeuchi, A.,K.Takahashi and H.Shimizu, "On Parallel Problem Solving Language and Its Applicaion," ICOT TR-418, 1988, also in Concepts and Characteristics of Knowledge-based Systems, M.Tokoro,Y.Anzai and A.Yonezawa(eds.), North-Holland, to appear.

[4] Takahashi, K., A.Takeuchi and T.Yasui, "A Parallel Problem Solving Language *AND OR-II* and Its Parallel Implementation," ICOT TR- , 1990.

[5] Ueda,K., "Guarded Horn Clauses," PhD. Thesis, The University of Tokyo, 1986.

スプレイ木の並列データ探索

和田久美子
(財) 新世代コンピュータ技術開発機構

概要

スプレイ木は, D.Sleator と R.Tarjan によって開発された自己調節型の二分探索木である. n 節点のスプレイ木に対する連続したある探索列における各探索の平均実行時間は $O(\log n)$ となることが知られている. しかし, 従来のスプレイ木のアルゴリズムでは木の変形がボトムアップに行われるため, ひとつの木に対して並列に複数のデータの探索を行うことができない. 本稿では, スプレイ木の性質を失わずに木の変形をトップダウンに行えることに着目した, スプレイ木の並列アルゴリズムについて述べる. 本アルゴリズムでは, 挿入, 変更, 及びその存在が保証される検索操作についてそれに引き続く操作を並列に処理することが可能であり, 理論的には n 節点のスプレイ木に対して $\log n$ に比例した並列度を得ることができる. 本アルゴリズムに対して幾つかの評価を行った結果, 無限個のプロセッサが存在する並列実行環境では, 100~500 個程度のデータの連続挿入に対して 4~8 程度の並列度が得られた. 本アルゴリズムは, データフローによる待ち合わせ機構を持つような言語を用いれば効率良く記述することができる. 本アルゴリズムは, 共有メモリを持つ密結合並列計算機上での効果が期待される.

1 はじめに

スプレイ木は, D.Sleator と R.Tarjan によって開発された自己調節型の二分探索木である [2]. 一般の二分探索木に対する最悪の場合の探索時間は木の深さに比例するので, n 個の節点を持つ二分探索木では, 最悪の場合 $O(n)$ 時間を要する. 一方, B- 木に代表される平衡木では, 各探索毎に木の平衡化を行う. n 節点の二分探索木には少なくとも $\log_2 n$ の深さの節点があるので, どんな二分探索木でも最悪の場合の検索時間は少なくとも $\log n$ の定数倍となるが, 平衡木では木を平衡化することで木の深さが $O(\log n)$ となるので, 最悪の場合の検索時間を $O(\log n)$ におさえることができる. しかし, 木の平衡を維持するために, 多くの場合変更アルゴリズムはかなり複雑である.

1.1 スプレイ木とそのアルゴリズム

平衡木の発想は単一探索と木の厳密な平衡化によるものであった. これに対し, スプレイ木は連続した検索列に対する重み付き平衡化に基づく. スプレイ木では, 一度参照された項は次にまた参照される可能性が高いと仮定し, 探索された節点を探索の終了毎に木の根に移動する. 移動の際, 木の変形が行われる. 変形はスプレイと呼ばれる操作により行われる. 木に対して節点 x でスプレイするには, 節点 x が木の根になるまで次のステップを繰り返す.

- Zig の場合

  $p(x)$(x の親) が木の根の子供ならば, x で回転して終了 (図 1-(a)).

- Zig-zig の場合

  x が左の子供で $p(x)$ も左の子供か, または x が右の子供で $p(x)$ も右の子供ならば, $p(x)$ で回転し, x で回転する (図 1-(b)).

- Zig-zag の場合

  x が左の子供で $p(x)$ が右の子供か，または x が右の子供で $p(x)$ が左の子供ならば，x で回転し，もう一度 x で回転する (図 1-(c))．

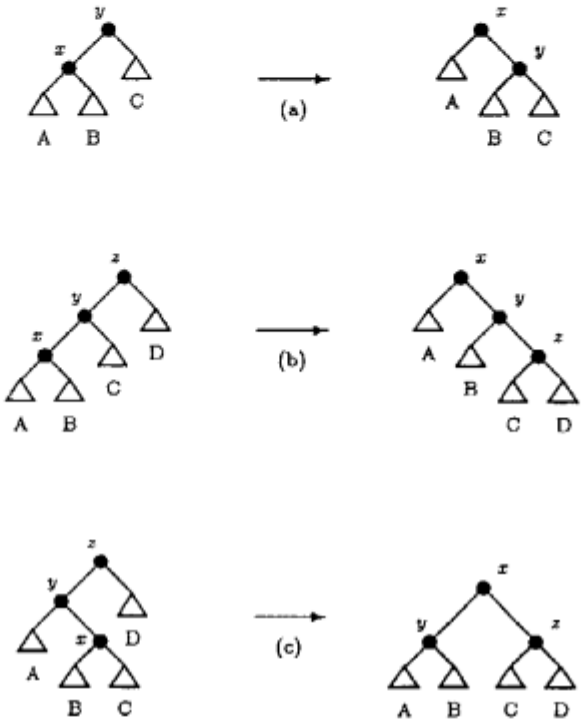


図 1: スプレイ操作

x でスプレイすると，x を木の根に移動するだけでなく，検索中にアクセスしたすべての節点の深さを約半分にすることができる．しかも，この操作によって木全体の任意の節点の深さは，高々 2 増すだけに留めることができる．

スプレイ木へは，一般の二分探索木と同様に，項の検索，挿入，削除，及び，木の併合，分割を行うことができる．アルゴリズムは以下のとおりである．

項 i の検索: 木の根から i を探索して行く．i を含む節点 x を発見したら x でスプレイし，x へのポインタを返す．*null* 節点に到達した場合は，i を含む節点が存在しないことがわかるので，検索時に通ったひとつ手前の *null* でない節点でスプレイし，*null* 節点へのポインタを返す．

木 T_1 と T_2 の併合: T_1 のすべての項は T_2 のすべての項より小さいと仮定する．まず T_1 の中の最大項を検索する．これを max とすると，検索後には T_1 の根は max，根の右の子供は *null* となるので，T_2 を新しく T_1 の右の子供とする．

項 i による木 T の分割: 木 T を，T_1 は i より小さい T のすべての項を含み，T_2 は i より大きい T のすべての項を含むように，T_1 と T_2 のふたつの木に分割する操作である．まず i を検索し，検索後の T の左右の部分木をそれぞれ T_1, T_2 とする．

項 i の挿入: まず i を検索し，到達した $null$ 節点を i を含む新しい節点で置き換え，その後その新しい節点でスプレイする．木を i で分割して T_1, T_2 とし，i を含む根にそれぞれ T_1, T_2 を左右の部分木としてつけるとしてもよい．

項 i の削除: まず i を検索する．i を含む節点を x, x の親を y とする時，x の左右の部分木を併合してこれを y の子供とし，その後 y でスプレイする．i を検索し，その左右の部分木を併合してもよい．

1.2 スプレイ木の効率

スプレイ木は，平衡木ほど厳密な平衡化は行わないが，それでも連続した探索列に着目した場合には平衡木と同等の効率が得られることで注目されている．具体的には，n 節点のスプレイ木に対する連続したある探索列における各探索の平均実行時間は $O(\log n)$ で，十分長い探索列に対しては，その列の探索時間の合計が最小となるように構成された最適の静的二分探索木と同等の良い効率を示すことがわかっている．

1.3 並列操作の可能性

以上に述べたアルゴリズムでは，1 操作毎に木に対してトップダウンに探索が行われた後，ボトムアップにスプレイ操作，すなわち木の変形が行われる．スプレイ操作の最後のステップが完了するまで木の根に配置される項やその子供が決定しないため，引き続く操作に対する探索はサスペンドしてしまい，並列に行うことは不可能である．

木構造に格納されたデータに対して並列に探索を行うには，探索や木の変形をすべてトップダウンに行い，各節点に含まれる項をトップダウンに決定して行くことが不可欠である．

2 並列アルゴリズム

本節では，スプレイ木に対して並列にデータ探索を行うアルゴリズムについて述べる．このアルゴリズムは並列探索を可能にするために，(1) 木の根から葉の方向にのみ進められるトップダウンな探索とスプレイ操作，(2) スプレイ操作に先立つ探索項の木の根への配置，の 2 つのポイントに基づいている．

トップダウンなスプレイ操作: スプレイ操作の結果得られるであろう部分木を論理変数を用いて表し，木の探索を行いながら同時にスプレイ操作も処理するようにする．これによって木に対する操作はすべて根から葉へ向かってトップダウンに行われることになり，スプレイ操作の影響によりまだ確定しない部分木と木の根を除いて残りの部分は次の操作のためのアクセスが可能となる．以下は探索とスプレイ操作を同時に行う方法である．木に対して項 i を含む節点 a を探索し，a でスプレイするには，節点 a が見つかるまで次のステップを繰り返す．

- Zig の場合

  x の子供が a の時，a で回転して終了 (図 2-(a))．

- Zig-zig の場合

  x の左の子供 y の左の子供 x か，または x の右の子供 y の右の子供 x の左右の部分木 L, R のいずれかの下に a があるならば，y で回転し，x を根とする部分木の探索及びスプレイ操作終了後の木 S (この木の左部分木を L?, 右部分木を R? で表す) の根 X? で回転する (図 2-(b))．

- Zig-zag の場合

  x の左の子供 y の右の子供 x か，または x の右の子供 y の左の子供 x の左右の部分木 L, R のいずれかの下に a があるならば，x を根とする部分木のスプレイ操作終了後の木 S (この木の左部分木を L?, 右部分木を R? で表す) の根 X? で回転し，もう一度 X? で回転する (図 2-(c))．

- 探索が失敗した場合

  x の子供に a がないとわかった時，終了．または，y の子供 x の子供に a がないとわかった時，x で回転して終了．

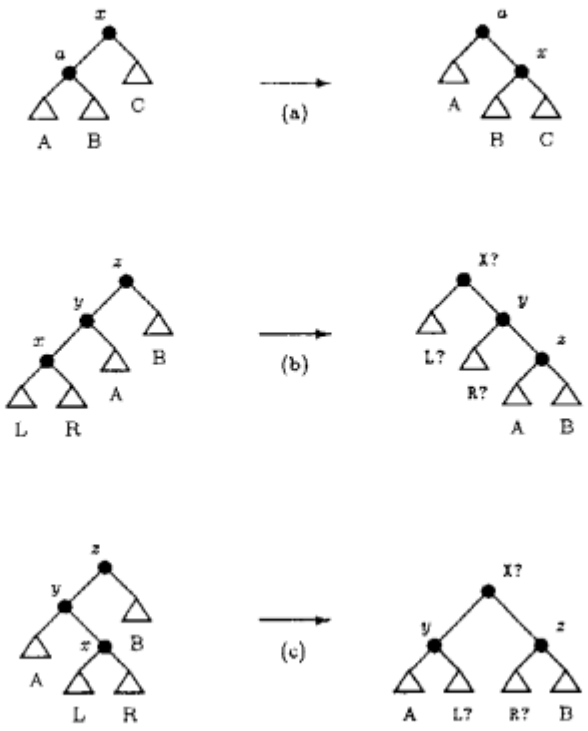


図 2: トップダウンなスプレイ操作

スプレイ操作に先立つ探索項の木の根への配置: 以上のトップダウンなスプレイ操作を行うと，木の根は結局探索の終了すなわちスプレイ操作の終了まで決定しない．これは，スプレイ操作そのものが最終的に探索された節点 (或いはそれが存在しない場合はその親節点) を木の根に移動することを意味しているためである．そこで，探索及びスプレイ操作の結果木の根となる節点があらかじめ確定している場合には，探索及びスプレイ操作に先立って探索項を木の根として配置してしまうことにする．これによって，次の操作に対する探索が木の根が確定されていないためにサスペンドすることはなくなり，現在行っている操作の探索と次の操作に対する探索を並列に実行することが可能となる．

このようなことを行うことのできる操作は，挿入操作である．その他，あらかじめその存在がわかっている場合の検索及び変更操作にも適用することができる．探索が終了しないと木の根となる節点が確定しない削除及び存在しない恐れのある項に対する検索操作に引き続く操作は，探索及びスプレイ操作の終了時まで開始できない．

3 評価

前節で述べたアルゴリズムの並列度に関して簡単な評価を行った．評価には汎用機上の FlatGHC 処理系を用いた．この処理系にはプログラムの並列度を計測するツール [3] が組込まれており，無限個のプロセッサが存在し，すべてのゴールが均等の機会で実行されるような並列環境が得られる．実行時には，実行待ちのすべてのゴールは一斉に試みられる．これをサイクルという．リダクション可能なゴールは 1 サイクル中に必ず実行され，リダクションによって生成された子ゴールは次のサイクルで試みられる．並列度は，総リダクション数を実行終了までに繰り返されたサイクル数で割ることによって求めた，1 サイクル当たりの平均リダクション数である．

表1, 2は, あるデータ列をスプレイ木に連続して挿入した場合の, それぞれのアルゴリズムでの実行結果を示す. データ列として, 昇順に並んだ整数列と乱数列の2種類を用い, データ数はそれぞれ100と200について試みた. 表中のアルゴリズムはそれぞれ, A: ボトムアップなスプレイ操作による従来のアルゴリズム, B: トップダウンなスプレイ操作によるアルゴリズム, C: アルゴリズムBにさらにスプレイ操作に先立ち探索項を木の根に配置する並列アルゴリズム, を表す.

データ数	アルゴリズム	リダクション数	サスペンション数	サイクル数	並列度
100	A	1805	599	805	2.24
	B	1204	301	702	1.72
	C	1304	302	309	4.22
200	A	3605	1199	1605	2.25
	B	2404	601	1402	1.71
	C	2604	602	609	4.28

表 1: 昇順整数列の挿入操作の実行結果

データ数	アルゴリズム	リダクション数	サスペンション数	サイクル数	並列度
100	A	4440	2307	2735	1.62
	B	2853	874	2247	1.27
	C	2953	959	519	5.70
200	A	10080	5421	6394	1.58
	B	6517	2009	5313	1.23
	C	6717	2186	961	6.99

表 2: 乱数列の挿入操作の実行結果

結果からは以下のようなことがわかる. アルゴリズムBはアルゴリズムAに比べて処理が簡潔になったため, リダクション数は約3分の2に減少している. サスペンション数も約半分に減少した. アルゴリズムBの並列度はデータ数100~200では昇順整数列で約1.7, 乱数列で約1.25となっており, 無駄な操作が省かれた分アルゴリズムAより低くなった. またこの数値は, 実行中終始リダクション可能なゴールが1個程度しか存在しなかったことを示しており, 非常に並列度が低かったことがわかる.

アルゴリズムCでは, 高い並列度が得られた. データが適当に定義されたある順序に従って昇順(または降順)に挿入される場合, スプレイ木では挿入場所が木の根に最も近い位置となるため, そのスプレイ操作は非常に軽い処理ですむ. 従って, 評価の一対象とした昇順整数列はアルゴリズムCの効果が最も現れにくい例であるが, 結果からはこのような挿入列に対しても効果が得られたことがわかる. 乱数列の挿入では, 昇順整数列よりも良い結果が得られた.

アルゴリズムCのリダクション数は, アルゴリズムBに比べてデータ数分だけ多い. これは, スプレイ操作に先立って探索項を木の根に配置するためのユニファイを示している. 乱数列に対するアルゴリズムCにおけるサスペンション数は, アルゴリズムBに比べて多くなっている. これは, 挿入のための木の並列探索が可能となったため多数の探索プロセスが生成され, それらが前の探索のためにまだ決定されない木の各所でサスペンドするためである.

木に格納されているデータ数が多くなり木が大きくなると, アルゴリズムCではそれに従って並列に行われる探索の数も多くなるため, 並列度は増加する傾向にある (表3). 一方アルゴリズムA, Bでは, リダクション可能なプロセスは1個程度である (表3).

n 節点のスプレイ木に対する1挿入操作当たりの探索の手間は $\log n$ に比例する. アルゴリズムCでは単位時間当たり一定した複数個の探索プロセスが木に入ることが可能なので, その並列度は理論的には

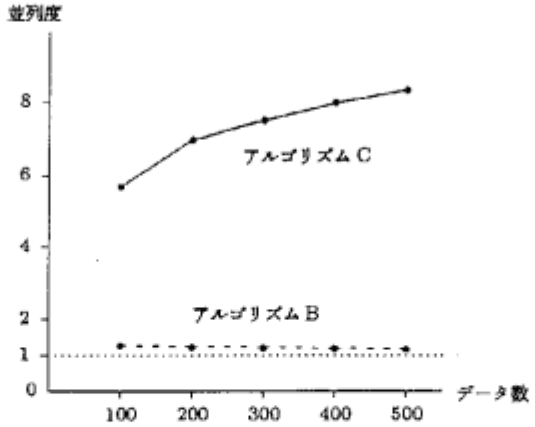


図 3: 乱数列の挿入操作における並列度

$\log n$ に比例すると考えられる．実行結果から，データ数 $N = 2^x$ の 3 種類の乱数列の挿入操作における並列度は，図 4に示されるとおり，x にほぼ比例していることがわかる．

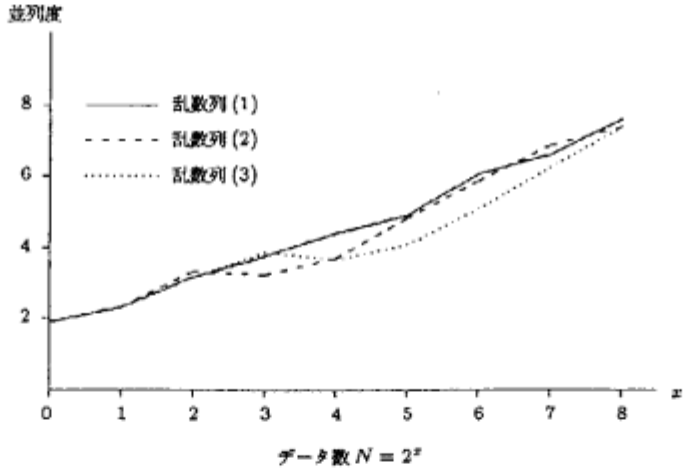


図 4: 3 種類の乱数列の挿入操作におけるデータ数と並列度の関係

4 議論

4.1 並列計算機上での実行

実際の並列計算機上で実行した場合の本アルゴリズムの効率について考える．本アルゴリズムでは，各プロセスは並列にすべてひとつの木のある部分にアクセスする．従って，疎結合並列計算機上でこれを実行するには，複数プロセッサによる木の分散管理を効率良く行うような負荷分散を考える必要があるが，プロセッサ間通信に高いコストがかかるため実現は難しい．本アルゴリズムはどちらかというと共有メモリ型の密結合並列計算機上で効果を発揮するであろう．

評価に用いた処理系では無限個のプロセッサが存在すると仮定されていたが，実際の並列計算機ではプロセッサ数は有限である．現在 ICOT で開発中の並列推論マシン PIM は，疎結合された複数個のクラスタ内に，それぞれ共有メモリを持つ複数個のプロセッサが密結合された形式である．1 クラスタ内には

8プロセッサ程度が用意される．本アルゴリズムでの評価結果から得られた並列度は100~500個のデータに対して4から8程度であった．この値は全実行時間内での平均をとったものであり，実際には連続した挿入列に対する操作の開始時と終了近辺ではもっと低いが，それ以外ではもっと高いことが予想される．従って，1クラスタ内で実行した場合，高い台数効果を得ることができるであろう．しかもより大量のデータを扱うならば，より高い台数効果を期待することができる．

4.2 改良と問題点

本アルゴリズムにおける最も大きな課題は，削除操作の並列化である．本アルゴリズムでは，スプレイ操作に先立って探索項を木の根に配置してしまうことが並列探索を可能にする鍵となっている．探索の結果探索項が存在しない，或いは探索項を削除する必要が生じる可能性のある操作では，その操作が完了して初めて木の根となる項が確定するので，このような操作では操作の完了までそれに引き続く操作を並列に開始することはできない．その代表的なものが削除及び検索操作である．

検索操作は主に存在する項に対して行われることが多いので，以下のような手段が有効である．すなわち，挿入操作と同様にスプレイ操作に先立って木の根に探索項を配置して探索を開始し，結局探索項が存在しなかった場合にはその項に”存在しない”という特別な印をつけ，のちに印の付いたその項を含む節点を削除する．このようにすれば，検索した項が存在しなかった場合にのみ削除操作が行われ，それ以外の場合に並列性が妨げられることはない．

従って，削除操作に引き続く操作を並列に実行可能とすることが最も重要な問題である．削除操作に対してはスプレイ操作を行わないと定義すると，削除ばかりが行われるような部分操作列に対しては木の自己調節機能が全く働かないことになり，探索の効率を落とすことになる．従って何らかのスプレイ操作を行う必要があり，現在検討中である．

本アルゴリズムのひとつの特徴として，探索される項が複数参照となることがあげられる．このためアルゴリズムをKL1で記述した場合，以下のような問題が生じる．すなわち，KL1処理系ではMRBと呼ばれるメモリ管理方式によって局所GCを行っており，複数参照された項に用いられたメモリは，それが構造体であった場合局所GCによって解放することが不可能となる．しかし実際問題では，各節点に格納する値は探索のキーとなる項とデータ部の対となっていることが多い．このような場合，本アルゴリズムではキーのみが複数参照となる．キーは探索の高速化のために簡潔な値とするのが一般的であるから，大抵の場合は構造体以外であり全く問題はない．たとえ構造体であってもそれほど複雑な値がキーとして用いられることはないであろうから，その部分が局所GCによって回収不可能となってもそれほど問題はないであろう．

スプレイ操作は木の深さ2ごとに探索経路の形状をチェックして行われる．一回のスプレイ操作につきその部分木の深さは1ずつ減少する．最後に深さ1分が余った場合，木の深さは変化しない．従来ボトムアップに行っていたスプレイ操作をトップダウンに行うように変更した結果，スプレイ操作の余りは木の根側でなく葉側に来ることとなった．これによって，木の深さ削減の影響は以前より木全体の変形に影響することとなったが，実行上の効率の変化は余りないようである．ただし，この点についてもさらに検討が必要である．

5 おわりに

本稿では，挿入，変更操作に対してスプレイ木を並列に探索する並列アルゴリズムについて述べた．本アルゴリズムでは，理論的にはn節点のスプレイ木に対して$\log n$に比例した並列度が得られる．本アルゴリズムは，データフローによる待ち合わせ機構を持つような言語を用いれば効率良く記述することができる．それ以外の言語では，待ち合わせを陽に記述すれば実現することができるが，効率面で問題が生じるかも知れない．汎用機上で動作するFlatGHC処理系を用いて，本アルゴリズムに対する幾つかの評価を行い，スプレイ木への挿入操作が並列に効率良く行われることを示した．

謝辞

本研究に関して有益な助言をいただいたICOTの近山隆第2研究室室長, 市吉伸行氏をはじめ, その他のICOTの研究員, PIMOS開発メンバーの方々に感謝します.

参考文献

[1] A. V. Aho, J. E. Hopcroft and J. D. Ullman. The Design and Analysis of Computer Algorithms, Addison-Wesley (1974).

[2] D. D. Sleator and R. E. Tarjan. Self-Adjusting Binary Search Trees. *J.ACM*, Vol.32, No.3 (1985), pp. 652-686.

[3] 杉野 他. 並列論理型プログラムの特性解析 (1) – 動特性解析ツール. 第35回情報処理学会全国大会 5Q-9 (1987).

An overview of FLIB

Bernard Burg Daniel Dure*

ICOT
1-4-28 Mita, Minato-ku, Tokyo 108, Japan
April 19, 1990

Abstract :

The FLIB library is a set of modules which is provided to help people programming in KL1 on PDSS and on the Multi-PSI system. This manual describes in details the predicates found in these modules and give guidelines on such issues as synchronization, parallel programming, etc. Our point has been both to relieve the user from the tedium of rewriting trivial functions, and to underline some problems commonly encountered. While we do not display any magic solution for the problem of distributing an algorithm over a loosely coupled network of processors, we present some simple examples which, we hope it at the least, set out the typical trade-offs to operate during and after design of the program, and demonstrate that in the current environment, performance analysis is 99% experiment-driven.

概要：

本ライブラリ FLIB は、PDSS システムあるいは Multi-PSI システムの上で KL1 言語を用いてプログラムを書くユーザーにとって有用ないくつかのプログラムモジュールを提供するものである。各モジュール中の述語には詳細な説明を付け、また並列プログラミングにおける同期の問題などに関してガイドラインを与えた。わたしたちが狙いとしたのは、ユーザーが基本的な述語を最初から作り直す手間を省けるようにすること、そしてプログラムを書く際によく起こる代表的な問題点に関して説明することである。疎結合ネットワークによるマルティプルプロセッサーにおいて、与えられたアルゴリズムに対してプロセスをどう分散させるべきかという問題について万能な方法が提供できるわけではないが、このレポートでは少なくともいくつかの簡単な例を挙げてプログラム開発中または開発後に考えなければならない典型的なトレードオフの問題を切り出すことができたならば幸いである。またパーフォーマンスの分析は、現在の並列環境では 99% 実験的に進めて得られたものである。

*current address: LIENS, 45 rue d'Ulm, 75005 Paris, France

Forewords

FLIB[1] [1] is a set of utility libraries, written in the KL1 language. It aims at improving various aspects of program development under PDSS and Multi-PSI, such as portability, synchronization of parallel programs, etc. It is by no mean a comprehensive library of functions, such as the ones commonly found in LISP systems, nor an interface to the environment, such as the UNIX libraries. Rather, we point here some specific problems with which we have been confronted during our development experiments. Therefore, some obviously useful functions were not introduced. On the other hand, we took care to program demonstratively, and to test both functionality and performance of these functions.
We dare think that the source code of these libraries can be usefully consulted by novice programmers. It gives useful clues as to practical realization of simple, low-level functions, and as to management of multi-processing. Of course, only the point of view of the authors is engaged here.

1 Introduction

This document describes in the large each library, and provides a list of the functions which can be found therein. So far, 7 modules have been developed:

fel This module forms a "stable" layer around system-related predicates and a few built-in predicates whose usage or precise syntax varies from PDSS to Multi-PSI. Its usage greatly relieves programmers from portability problems.

list This module shelters some predicates to manage or convert data organized into lists.

matrix This module introduces a format for matrices and provide some basic predicate to make computations on integer matrices, or convert them.

par This module holds predicates which can be used for synchronization or to spread computation over several processors, while taking care of duplicating data or setting up communication streams.

string This module holds predicates to manage, recognize and convert character strings.

util This module contains various utilities, to monitor activity, copy data, send messages on the console, etc.

vect Last of all, this module contains some predicates to manage or convert data organized into vectors.

Before the list of modules and the description of predicates they contain, the sequel of this section holds some comments on the library and its usage. After module description, the whole section 9 is dedicated to the detailed presentation of two programming examples.

2 Module fel

The predicates in this section have been created to improve portability of KL1 programs between PDSS and Multi-PSI systems. Some of them are already obsolete, as both systems slowly but hopefully converge. They are kept for the sake of compatibility with previous versions.
Predicates have been arranged according to the following topics:

virtual device open: `open_window`, `open_file`, `timer`

strings: `appendstring`, `sub_string`, `setsubstring`, `char_to_ascii`

comparison: `mini`, `maxi`

atom: `atom_to_name`, `name_to_atom`, `atom_to_number`

code: `get_code`

shoen: `shoen_execute`, `shoen_raise`

version: `version`

[1]This work as been sponsored by INRIA, Rocquencourt, France through a grant and exchange program with ICOT, Tōkyō, Japan.

3 Module list

This module contains predicates to help management of data organized as lists. Curiously, whereas this data structure is highly suited to the representation of graphs, tree structures and dynamic data structures, little support is found for it in KL1. Besides, in the current implementation of KL1 on the Multi-PSI, the transfer of lists is done element by element, which makes it very unsuited to exchange of large chunks of data, for which vectors, or even strings are to be preferred. I'm sorry to say that nothing indicates that this would be about to change soon. In the following, predicates are arranged according to the following topics:

basic list operations: `length`, `sync_length`, `append`, `sync_append`, `reverse`, `rec_reverse`, `sync_rec_reverse`, `subst`, `sync_subst`, `rec_subst`, `sync_rec_subst`, `remove`, `sync_remove`, `rec_remove`, `sync_rec_remove`, `member`, `sync_member`

sublists operations: `sublist`, `sync_sublist`, `setsublist`, `sync_setsublist`, `split`, `sync_split`

set operations: `union`, `sync_union`, `inter`, `sync_inter`

sorting: `simple_comparator`, `comparator`, `sort_a`, `sort_d`, `sync_sort_a`, `sync_sort_d`, `quicksort_a`, `quicksort_d`, `sync_quicksort_a`, `sync_quicksort_d`, `nodoub`, `sync_nodoub`

arithmetic: `mini`, `sync_mini`, `maxi`, `sync_maxi`, `sum`, `sync_sum`, `product`, `sync_product`, `andl`, `sync_andl`, `orl`, `sync_orl`

data conversion: `list_to_string`, `list_to_vector`, `rec_list_to_vector` `sync_rec_list_to_vector`

version: `version`

4 Matrix module

A 2-dimensional matrix of *integers* is represented as a vector of vector. Each element in the first vector corresponds with a column. First element in a column or a row is denoted by 0.
Predicates of this module are arranged according to the following topics :

type checking and creation: `matrix`, `new_matrix`

access to elements: `matrix_element`, `set_matrix_element`, `matrix_col`, `set_matrix_col`, `matrix_line`, `set_matrix_line`

basic matrix operations: `add`, `sub`, `mult`, `transpose`, `trace`

data conversion: `matrix_to_vector`

version: `version`

5 Module par

The name of this module is derived from the word "parallel", as some of the predicates therein may help achieving the high deed of using several processors on the Multi-PSI.
The specification of some of these predicates is a little bit hairy, and we would advice our puzzled reader to have a look on the examples or into the actual code. We shall improve the text according to requests...
The following predicates are organized according to the following topics :

parallel processing: `create_list_of_p`, `sync_create_list_of_p`, `apply_list_of_p`, `create_2_tree_of_p`, `sync_create_2_tree_of_p`, `apply_2_tree_of_p`

synchronization: `sync_wait_list_of_p`, `sync_rec_wait_list_of_p`, `sync_wait_2_tree_of_p`, `sync_rec_wait_2_tree_of_p`

version: `version`

6 Module string

The following predicates are an attempt at pattern matching, for the main part. We also provide some operations usually found for lists. Predicates are arranged according to the following topics:

basic string primitives: `reverse` `find_substr_bm`, `findsubstr`, `sub_string`, `setsubstring`, `split`

sorting: `comparator`

data conversion: `string_to_vector`, `string_to_list`, `sync_string_to_list`

version: `version`

7 Module util

This module is the historical startpoint of the FLIB library, as you could guess from its "vague" name. It contains various predicates which can neither fit in another module nor contribute alone to a new module. Some predicates were moved to other modules in the previous version, but we left here, for the sake of compatibility, the original predicates, which are in fact calling the code in the newer modules.
Predicates are gathered according to the following topics:

integer to string and vice-versa: `dec_int_to_name`, `hex_int_to_name`, `bas_int_to_name`, `dec_name_to_int`, `hex_name_to_int`, `bas_name_to_int`

generic copy & compaction: `copy`, `sync_copy`, `flash_copy`, `object_to_string`, `string_to_object`

synchronization: `sync_wait`, `sync_rec_wait`

console output: `p_console`, `sync_p_console`, `flash_p_console`

enhanced timer & statistics: `timer`, `tart_stats`

random number generator: `random`, `random_bound`

version: `version`

8 Module vect

This module offers predicates of similar shape to the ones as in the module `list`, but is suited to evaluation over vectors. Predicates are arranged according to the following topics:

basic vector primitives: `append`, `reverse`, `rec_reverse`, `sync_rec_reverse`, `subst`, `rec_subst`, `sync_rec_subst`, `remove`, `rec_remove`, `sync_rec_remove`, `member`

subvectors: `findsubvect`, `subvect`, `setsubvect`, `split`

set operations: `union`, `inter`

sorting: `comparator`, `sort_a`, `sort_d`, `quicksort_a`, `quicksort_d`, `nodoub`, `sync_nodoub`

set arithmetic: `mini`, `maxi`, `sum`, `product`, `andv`, `sync_andv`, `orv`, `sync_orv`

vector arithmetic: `vector_add`, `vector_sub`, `vector_scal_product`

data conversion: `vector_to_list`, `sync_vector_to_list`, `rec_vector_to_list` `sync_rec_vector_to_list`, `vector_to_string`, `vector_to_col_matrix`, `vector_to_line_matrix`

version: `version`

Be aware when you use vectors that multiple references to vectors cause indirections when modified elements are further accessed. This implies increased acces time.

9 Examples

Two toy problems are presented: searching occurrences of a subvector in a vector and sorting a list. We introduce first the sequential version of these programs, then the parallel version. It shows the use of FLIB's parallel facilities and highlights difficulties related to parallel programming. Time measurements follow, in order to give some advice to the next programmers about the nature of the technical choices to do.

9.1 Search of a subvector in a vector

The basic algorithm used to perform the search is from Knuth-Morris-Pratt. We begin with the sequential implementation, *sequential* the `vect:findsubvect` of FLIB, then we make a naive parallel version *naive*, splitting the task on the Multi-PSI processors. Finally we develop a more sophisticated version named *final*, including data-compression, synchronization and time measurements. These versions have been run on the multi-PSI and the results are commented.

9.1.1 Performance measurements

Three examples have been run. We search for a subvector of 15 elements in a 5000 elements vector. These vectors have been randomly generated, taking elements of a given vocabulary. The vocabularies for the three examples are:

ex1 {a,b,c,d}
ex2 {[a,[b,c]],[1,2,3,4],[apply,add,a,b,c,d],mult,[1,2,3,4],5,6}
ex3 {[a,[b,c,d,e,f]],[1,2,3,4,5,6],[a,a,a,b,c,d,g,h,i],[m,[1,a,b,c,[2,3],4],5,6]}

kmp		sequential	2 processors		5 processors		10 processors		15 processors	
			naive	final	naive	final	naive	final	naive	final
ex1	Kred	20	54	109	76	176	112	288	150	406
	s	0.5	1.3	2.3	2.1	3.2	3.7	5.1	5.4	7.2
ex2	Kred	20	138	477	275	919	501	1660	732	2410
	s	0.5	9	9.9	24.4	17.8	49.3	31.5	74.5	45.3
ex3	Kred	20	166	654	344	863	639	2277	937	3308
	s	0.5	13.6	13.5	29.4	25	56.8	44.4	83.2	65.8

Table 1: Time and reduction measures of kmp

As a first comment of Table 1, let's note that the sequential version runs in constant time, whatever the vocabulary is. It is always faster than the parallel version. This is explained by Figure 1. First of all, to speed-up the transmission,

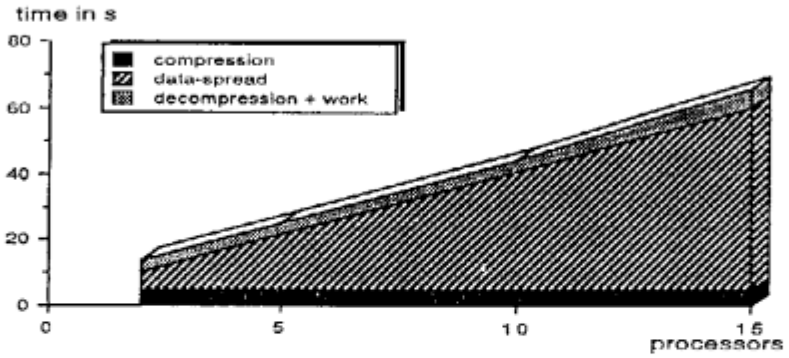


Figure 1: search of a subvector in *ex3*

we compress the data. This takes approximatively 5 seconds. Then, this data is spread over the processors. This takes between 5 to 55 seconds, according to the number of processors. The data is then decompressed, in about 3 seconds, and finally the kmp algorithm is performed in less than 0.5 seconds.

Because of the high simplicity of kmp search, the sequential version is several degree of magnitude faster than parallel implementations on Multi-PSI/V2. The ratio between data transmission and the work to be carried out is too high. Time measures and the related listings show that the additional work to be done by the parallel version is not reasonable (with 15 processors, we have to make 165.4 times more reductions than the sequential version).
Another interesting point concerns the speed of the naive parallel version which is faster than the final version of kmp in example 1. In fact, in the case of a simple vocabulary, the data compression used in the final version does not change the transmission performances, but introduces some compression overhead. The more the data becomes hairy, the more the compression is efficient. The two last examples show it. We can note that the final version of kmp has a much higher *reduction-rate* $= \frac{Reductions}{Time}$. This illustrates the vital role of synchronization and inter-processor communication. For a discussion of evaluation criteria, one can refer the paper of K. Taki [2].

9.2 Sorting and removing doubles

We implemented a sort to remove all the double elements in a list. For the sake of clarity, this program has been designed for integers only. The sorting algorithm we use in the sequential version is the quick-sort [3]. Subsequently, a simple loop removes consecutive doubles.
We took this algorithm and put it in the parallel environment. The tree-spreading function of FLIB provides an efficient data spreading structure to do that. The first parallel version *ver1* spreads the whole data set to the processors. Each of them performs a sort on its data and then merges the results with those issued by the child-processors. The second parallel version *ver2* differs by the data transmitted to processors. Each of them takes its own part of the work and shares the data to be sorted between the children. Performance measurements give an idea about the gained efficiency.

9.2.1 Performance measurements

This program was run to sort 3 lists, from 100, 500 and 5000 atomic elements. The results are shown in Table 2.

sort		sequential	2 processors		5 processors		10 processors		15 processors	
			ver1	ver2	ver1	ver2	ver1	ver2	ver1	ver2
100	Kred	3.8	10	9.3	17	16	29	28	42	39
	s	0.16	0.27	0.35	0.3	0.36	0.35	0.44	0.41	0.46
500	Kred	61	46	34	41	32	53	42	69	53
	s	0.8	0.51	0.6	0.41	0.46	0.5	0.47	0.51	0.5
5000	Kred	5104	2650	1677	1197	841	764	552	680	529
	s	58.4	15.5	15.7	3.8	3.8	2.3	2.8	2.1	2.1

Table 2: Time and reductions measures with *sort*

By dealing with 100 elements, the parallel implementation looses, compared to the sequential one. With 500 elements, the best performance is obtained with 5 processors using the first parallel version. To sort 5000 elements, the first parallel implementation using 15 processors turns out to be faster. The parallel performance increase is super-linear. One can observe the evolution of the number of reductions performed by the program. It simply means that the sequential version is far from optimality (the number of reductions gives obvious information). Figure 2 represents in detail, the work done to sort 5000 elements. Roughly, the data compression time is constant, the data spreading increases slightly, the sorting time decreases dramatically. This result is due to the complexity of the sorting algorithm. Although related to quicksort, which is in $\mathcal{O}(n \log(n))$, its complexity tends to $\mathcal{O}(n^2)$ when data are not well distributed in the initial list to sort. This figure of complexity becomes in parallel, considering the cost of the merge operation, $\mathcal{O}\left(\left[\frac{n}{Pro}\right]^2\right) + \mathcal{O}(n \log n)$. When sort involves 5000 elements, the detailed analysis predicts a slowdown for more than 15 processors, since the transmission time increases more than the decrease of the processing time. The tree-data structure used in this algorithm allows fast data transmissions and merging of the partial results in parallel.

9.3 Conclusion

The two examples we showed above lead to some hints in parallel programming on the Multi-PSI:

- The search of a subvector in a vector by the kmp algorithm is a brilliant counter-example of the usefulness of parallelism. In the worst case, the parallel implementation with 15 processors takes 166 times longer (naive version), or makes 165 more reductions (final version) that the sequential version. These awful results are due to the simplicity of the kmp algorithm, compared to the size of the data to be transferred.

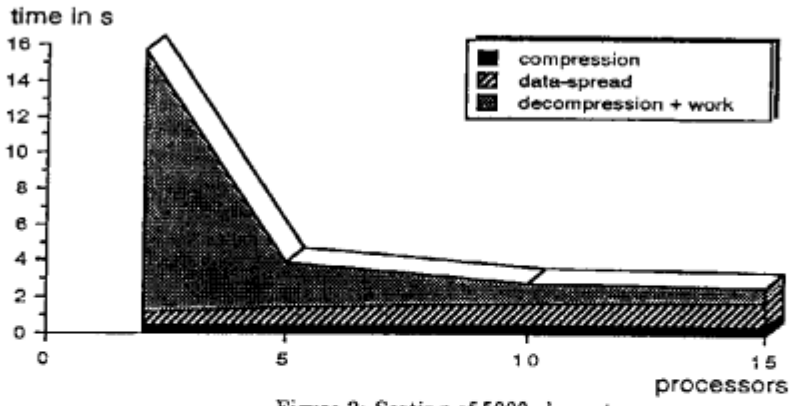


Figure 2: Sorting of 5000 elements

- The sorting algorithm showed in contrast a super-linear speed-up, when dealing with big examples. In fact, the principle used by the parallel implementation transformed the sorting, breaking the complexity of the original algorithm. The parallel algorithm shows however a speedup limit, with a certain number of processors. By using more processors, time is wasted in communications, etc.

Conception of efficient parallel programs remains an art. More work has to be done towards a better knowledge of the multi-PSI.

References

[1] B. Burg and D. Dure. FLIB user manual. Technical Report TR 529, ICOT, 1990.

[2] K. Taki. *Programming of future generation computers*, volume 2, chapter Measurement and evaluations of the Multi-PSI/V1 System. North-Holland, 1988.

[3] K. Ueda. Introduction to guarded horn clauses. Technical Report TR 209, ICOT, 1986.

PIMOSの資源管理におけるストリームの扱いについて

㈱三菱総合研究所　藤瀬　哲朗

1　はじめに

PIMOS(Parallel Inference Machine Operating System)は,ICOT で開発中の並列推論マシン([1],[2])の能力を最大限に引き出すために,高並列に動作するプログラムを効率的に制御するオペレーティング・システムである([5]).

PIMOS は計算機資源(CPU資源,メモリ資源,入出力資源)の管理を行い,利用者の過ちから処理系全体を,ひいては実行中の利用者プログラムを守ることを基本機能としている.そのため利用者プログラムから直接計算機資源を操作させずに,計算機資源保護のための枠組みを付加し,仮想化された資源として利用者に提供する([4]).本稿では,OS資源(入出力機器などのOSが定義し,利用者に提供する計算機資源)に対する付加管理機構の実現方式,特にストリームの扱いについて述べる.

2　OS 資源管理の基本方針

(1) 管理単位

PIMOS では,計算機資源を管理する単位をタスクと呼ぶ.利用者はタスクを生成してその上でプログラムを実行することにより,プログラム実行の中断,再開,放棄などを行う.OS 資源の獲得,解放もタスク単位で管理される.すなわち,タスクが終了するときには,そのタスクが使用したすべてのOS資源が解放される.

なお,タスク自身もOS資源である.タスクがタスク(子タスクと呼ぶ,また前者を親タスクと呼ぶ)を生成した場合,子タスクは見かけ上親タスクのメタ・インタプリタにより動作しているように位置づけている.親タスクが終了する場合,同様に子タスクも終了し,かつ子タスクが使用したOS資源も解放する.

(2) 管理の基本方式

PIMOSは,利用者プログラムと同様に並列論理型プログラミング言語KL1 で記述される.OS資源はKL1節

user(Svariable),　os(Svariable)

で示される通り,共有変数を介した通信によりタスク(利用者プログラム)から利用される.このことは,KL1のユニフィケーションがプロセス間通信であることを表現している.なお,ここで呼ぶプロセスとは適当な処理モデルの単位であり,一般的に定義されたものではない.

図 1　　OS資源の利用

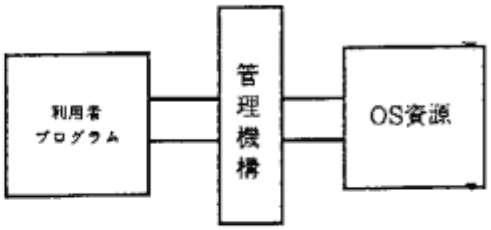


図 2　　OS資源の管理

利用者プログラム中の資源を高並列に動作する環境下で管理することは,容易なことではない.例えば逐次計算機ではよくOSの処理中に利用者プログラムが勝手な動作をしないように利用者プログラムの実行を禁止することを行う.しかし高並列環境下では実行の局

所的な逐次実行を行うためには,他の並列に動作するプログラムをすべて停止させることになり,明らかに並列性を犠牲にすることになる.PIMOSではPIMOS中のデータ部分を利用者が操作するためには共有変数を利用せざるを得ない.PIMOSは共有変数を介したメッセージの処理手順を制御することで並列性を犠牲にすることなく問題を解決する.PIMOSでは,共有変数をプロセス間のストリームとして利用することで,ファイルの読み書き等のOS資源に対する連続的なメッセージ通信をモジュラリティ良く実現する(図1).従ってPIMOSではこのプロセス間のストリームを管理することが,OS資源を管理することに相当する.換言すればPIMOSは入出力装置等に管理機構を付加することで,入出力装置等をOS資源として利用者に対して提供する(図2).

(3) 資源の階層化

OS資源はタスク単位で管理されるため,各OS資源の管理機構をタスク自身の管理機構がさらに管理する必要がある.タスク管理機構自身も動作中にもかかわらず管理機構からはずれることを防ぐために,前述したタスクをOS資源とみなすことで,さらに一階層上のタスクの管理機構が管理する.このように階層的資源管理機構を導入することは,実は資源管理機構への利用者プログラムからの通信の集中防止にも役立つ.

現在のところ,PIMOS中で階層性のある資源はタスクだけである.もちろんタスク以外でも階層性のある資源は全く同じ機構により実現可能である.OS資源の管理単位はOS資源そのものである.

3 資源木による管理方式

PIMOSは使用するすべてのOS資源を,資源木と呼ばれるストリームで資源管理機構を結合したプロセスの木構造で管理する.

(1) 資源木の構成要素

PIMOSの階層構造例を図3に示す.この例中には3つのタスクが存在し,それぞれのタスクは以下のOS資源を使用している.

親タスク: 1つの入出力装置,子タスク1,子タスク2
子タスク1: 2つの入出力装置
子タスク2: 1つの入出力装置

図3の右側部分を資源木と呼び,タスクとは別の領域に置かれる.資源木の各ノードは,タスク階層に対しては木構造になり,タスク内資源は,それぞれ輪構造で結合している.図3に中にも表れる資源木の構成要素について以下に説明する.

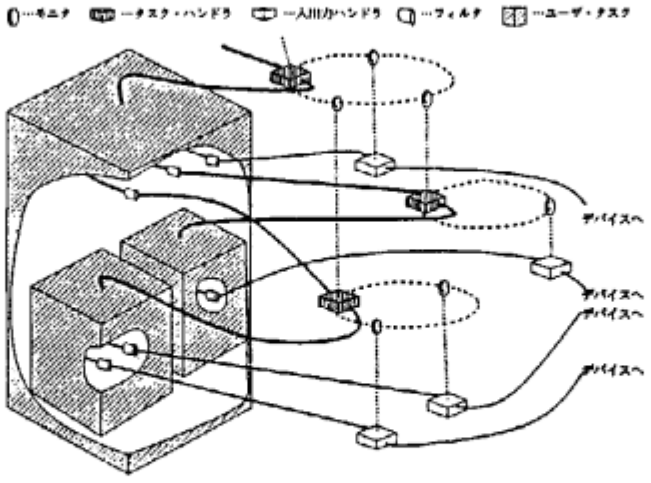


図 3　資源木の構造例

タスク・ハンドラ:
タスクの実行管理およびタスク内の子資源の管理を行い,親資源からは逆に管理されている.実際に階層構造を利用した資源である.
タスク内で使用中の入出力装置や子タスク等のOS資源は,それらに対応するモニタ(後述)を輪状に結合して管理される.これらの資源に対する操作は,このモニタを通して子資源のハンドラに送られる.

入出力ハンドラ:
入出力装置を資源としてみせる機能をもつ.タスクと入出力を実際に行う装置(ドライバと呼ぶ)との間に介在し,デバイス・ハンドラに相当する処理とメタ的な問い合わせの処理を行う.

モニタ:
名前の通り子資源の情報を保持し,そのハンドラを監視することで子資源を管理する機能をもつ.タスク・ハンドラからみれば,子資源の代理人に相当し,タスク・ハンドラはすべてこのモニタに対して子資源の操作を行う.

フィルタ:
タスク側の誤りから処理系全体を保護するプロセスである.

(2)局所資源管理モデルとストリームの扱い

利用者は資源を利用するために,ストリームを介して資源木中のプロセスと通信を行う.また,PIMOS自身も資源木を構成するプロセス間のストリームを介して各資源の制御を行う.

PIMOS中の各資源は並列に動作するため,各資源は利用者やPIMOSの要求により任意の時点で勝手に強制終了/通常終了する可能性がある.PIMOSの資源管理機能では,資源群全体として起こり得る様々な事象に対処するために,図4に示す局所的な資源管理モデルを設定した.典型的な通信パターンを以下に示す.

⑤→⑥:
利用者プログラムから対象資源の利用

①→③:
親資源から対象資源へ制御/問い合わせ

①→③→⑦:
親資源から対象資源の子資源へ制御/問い合わせ

⑧:
子資源から対象資源へ問い合わせ

⑧→④→②:
子資源から対象資源の親資源へ問い合わせ

⑤→⑦:
利用者プログラムから対象資源の子資源へ制御/問い合わせ

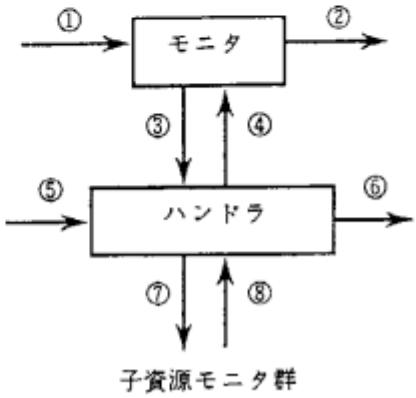


図 4　局所資源管理モデル

その他にも様々な処理ケースが考えられるが,最も注意を要する部分は終了処理部分である.例えば,子資源の利用を終了したと同時にこの時に利用者プログラムが強制終了した場合,子資源の終了処理と利用者プログラム・タスクの終了処理の競争になる.最初に述べたように処理自身の逐次性は保証されないため,この両者がどのような順で処理されても正しく動作する方式を採らなければならない.PIMOSでは局所管理モデルに次の規則を適用することにした.

規則

1 資源は自身を終了するために,親資源から終了許可を得る必要がある.そして親資源からのストリームを親資源が切り離した時に初めて終了することができる.
2 資源は自身を終了するためには,子資源の終了を確認する必要がある.
3 資源は子資源から終了許可要求があった後,子資源へ終了許可を送り,2度と子資源へのストリームへメッセージを送っては

いけない．ただし，子資源からのメッセージは受信する．最後に子資源が資源へのストリームを閉じた時(すなわち子資源からメッセージが来ることがないことが保証される時)，子資源へのストリームを閉じる．

4 資源は自身の強制終了時，子資源に強制終了命令を別途発するが([3])，資源の終了処理はあくまで1，2，3に従う．

終了手順を図4で説明する．

終了手順

1 ⑤が閉じられる(PIMOSには親資源が強制終了命令を発すると，自動的に⑤が閉じられる機構が備えられている)．もちろん強制終了時には子資源に強制終了命令を発する．
2 子資源の終了を確認するため⑦に終了確認通知を送る．この通知を受け取った子資源のモニタは今後，対応する子資源へメッセージを送らないこととする．
3 ⑧からすべての子資源に終了通知が送られたことを確認した後，④に終了許可を要求する．
4 モニタは④から終了許可要求を受け取ると，もしモニタが①から既に終了確認通知を受け取っていた場合，子資源に終了許可通知を送る．また受け取ってない場合には①から終了確認通知を受け取った時，子資源に終了許可通知を送る．
5 ③から終了許可通知を受け取る．このとき初めて終了できる準備が整ったので，以下の最終処置を行う．
6 ⑦の通信路を閉じる．
7 次に⑧の通信路が閉じられたら④の通信路を閉じる．
8 モニタが④の通信路が閉じられるとモニタは①と②を結合し，③の通信路を閉じる．そしてモニタは消滅する．
9 ③の通信路が閉じられたら，最後に⑥を閉じて資源は消滅する．

一見，非常に複雑な処理のように感じるかもしれない．しかし動作ステップが少なく，なおかつ並列推論処理マシン上では処理待ちの間に他の処理を妨げない．もちろん途中で他から処理を妨げられることはないので，終了処理は必ず終了する．

(3) プロセッサ間をつなぐ資源管理ストリーム

複数プロセッサ上でPIMOSは次のように分岐している．基本的にPIMOSの対象資源の管理部分(およびドライバ)は，すべて利用者プログラム中の資源獲得部分が動作中のプロセッサ上に節操もなく形成される．親資源の資源管理部とプロセッサが異なる場合，資源のモニタは親資源が動作するプロセッサ上に生成され，他の部分は新しいプロセッサ上に形成される．図4の③および④がプロセッサ間をつなぐ資源管理ストリームとなる．これは次のことを意味する．親資源のプロセッサ中では，モニタが資源の代理人を務める．親資源がその子資源の管理を終了することは，プロセッサ間の通信路を閉じてモニタが消滅するに相当する．

4 おわりに

PIMOSでは，モデルをそのまま局所資源管理プロセスとして実現し，そのプロセスを単純に結合することで一貫した管理機構を実現した．本モデルの実現には多重同期処理を非常に数多く必要とし，通常デバッグが難しいところである．しかし多重同期処理自体がKL1の言語原始要素であるため，個々の同期にとらわれず，高いレベルでモデルを作ることにより容易に実現できた．また今回は省略したが，どの処理系でも行っているバッファリングを正しく適用することにより，管理機構の計算負荷がほとんど無視し得ることがわかった([6])．

PIMOSの資源管理部はプロセス・モデルを構築し，プロセス間を結ぶストリームとそのプロトコルを決めることで，容易に実現できた．しかし実際によく見直してみると実現方式がオブジェクト指向そのものであることもわかった．

謝辞

本研究に関して有益な助言を頂いたICOT第2研究室長の近山隆博士, 三菱電機の佐藤裕幸氏および沖電気工業の宮崎敏彦氏に深く感謝する.

参考文献

[1] Goto, A. et al.: Toward a High performance Parallel Inference Machine -The intermediate State Plan of PIM -, Technical Report TR-201, ICOT (1986).

[2] Taki, K.: The Parallel Software Research and Development Tool: Multi-Psi System, Technical Report TR-237, ICOT (1986).

[3] 松尾他: PIMOS のタスク管理方式 - タスク終了時の資源解放-, 第36回情報処理学会全国大会論文集 3D-4, pp.293-294(1988).

[4] 藤瀬他: PIMOS の階層的資源管理, 第37回情報処理学会全国大会論文集 5P-3, pp.251-252 (1988).

[5] Chikayama,T. et al.: Overview of the Parallel Inference Machine Operating System (PIMOS), Proc.of FGCS'88, Vol.1, pp.230-251(1988).

[6] 佐藤他: PIMOS の資源管理方式, 情報処理学会論文誌 Vol.30 No.12, pp.1646-1655 (1989).

表 3: 各ストリームのデータ量とプロセス数

ストリーム番号	(i)			(ii)		
	D	P	M	D	P	M
0	1	9	9	1	9	9
1	28	11	308	6	11	66
2	64	7	448	15	7	105
3	16	1	16	4	1	4
4	76	1	76	18	1	18
計	185	29	857	44	29	202

ストリーム番号	(iii)			(iv)		
	D	P	M	D	P	M
0	1	9	9	9	1	9
1	4	11	44	11	4	44
2	11	7	77	7	8	56
3	4	1	4	1	2	2
4	16	1	16	1	12	12
計	36	29	150	29	27	123

D: データ数
P: プロセス数
M: データ数 × プロセス数

表 4: 単一プロセサでのリダクション数

文番号	(i)	(ii)	(iii)	(iv)	(i) — (ii)	(i) — (iii)	(i) — (iv)
1	1,669	969	678	619	1.72	2.46	2.52
2	1,451	810	572	458	1.79	2.54	3.17
3	7,388	3,577	2,919	2,600	2.07	2.53	2.84
4	4,405	2,408	1,736	1,603	1.83	2.54	2.75
5	9,547	4,856	4,348	3,784	1.97	2.20	2.52
6	15,307	7,516	6,135	5,501	2.04	2.50	2.78
7	47,340	18,740	16,682	14,538	2.53	2.84	3.26
8	16,495	8,709	6,810	6,112	1.89	2.42	2.70
9	17,398	9,106	7,834	6,745	1.91	2.22	2.58
10	1,949,802	686,495	611,624	512,706	2.84	3.18	3.80
11	2,844,476	915,663	825,516	736,253	3.11	3.45	3.86

表 5: 単一プロセサでの実行時間 (msec)

文番号	(i)	(ii)	(iii)	(iv)	(i) — (ii)	(i) — (iii)	(i) — (iv)
1	71	61	64	53	1.16	1.11	1.34
2	69	56	52	52	1.23	1.33	1.33
3	163	115	90	88	1.42	1.81	1.85
4	117	87	83	71	1.34	1.41	1.65
5	202	119	113	103	1.70	1.79	1.96
6	302	160	148	131	1.89	2.04	2.31
7	811	344	304	270	2.36	2.67	3.00
8	307	175	149	151	1.75	2.06	2.03
9	332	190	167	150	1.75	1.99	2.21
10	29,068	10,522	9,363	8,634	2.76	3.10	3.37
11	43,104	14,000	12,764	11,955	3.08	3.38	3.61

も、同様の順に減少しており、改良版 PAX では約 $\frac{1}{7}$ になっている。従って、実行速度の向上がかなり期待できる。

改良版 PAX はオリジナル PAX のプロセスとストリームを逆転させているため、各ストリームでの (i), (ii), (iii) のプロセス数と (iv) のデータ数が等しくなっている。一方、オリジナル PAX のデータ数と改良版 PAX のプロセス数は異なっており、前者の方がかなり多くなっている。これは、オリジナル PAX では、ある (非) 終端記号まで解析されると可能な右側の識別子をすべて送るのに対して、改良版 PAX では、二つのとなり合う (非) 終端記号が両方とも解析されないとその間に相当するプロセスが起動されないため、改良版 PAX のプロセス数の方が少なくなっていると考えられる。。

4.3 単一プロセサでの測定

単一プロセサ上でのリダクション数及び実行時間を表 4, 5 に示す。リダクション数の減少及び処理速度の向上とも、(i)< (ii)< (iii)< (iv) の順になっている。特に改良版 PAX (iv) は、リダクション数で、$\frac{1}{2.5} \sim \frac{1}{3}$ になり、実行時間で、$\frac{1}{1.3} \sim \frac{1}{3.6}$ になっている。

前節でのデータ数 × プロセス数の減少比率より、リダクション数や実行時間の減少比率が小さいのは、データ数 × プロセス数には補強項の実行が含まれておらず、実際の解析では補強項の実行比率が大きいからであろう。

4.4 複数プロセサでの測定

例文 11 の複数プロセサでの実行時間を表 6 に示す。オリジナル PAX, 改良版 PAX とも、各ストリーム毎にプロセサを割り付けることで、プロセサ間の通信量を抑える負荷分散方式を採用している。表中の (i') は、この負荷分散

表 6: 複数プロセサでの実行時間 (msec)

PE 台数	(i')	(i)	(ii)	(iii)	(iv)
1	43,104	43,104	14,000	12,764	11,955
2	110,130	46,996	21,420	20,306	21,003
4	79,078	34,120	16,507	16,098	15,922
8	61,521	27,646	12,228	12,749	12,650
16	72,576	21,131	8,947	8,227	9,913
32	69,251	13,167	6,386	6,151	8,133
$\frac{1}{32}$	0.62	3.27	2.19	2.08	1.47

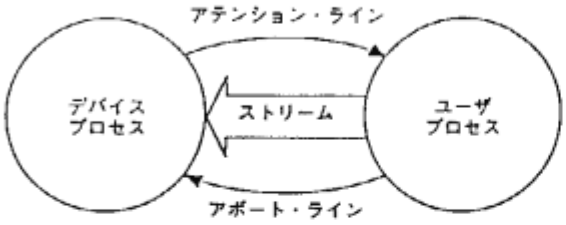


図1：ストリームとライン

デバイス・プロセスは，アボート状態と正常状態があり，デバイスが生成された時点ではアボート状態である．アボート状態では，すべてのIO要求メッセージは無効になり，メッセージのStatus変数をabortedにして処理を完了するため，デバイス・プロセス生成後の最初にresetメッセージをデバイスに送り正常状態にする必要がある．

このようにアボート状態でのメッセージを全て無効にするのは，ストリームにバッファリングされたメッセージをクリアする意味を持つ．また，アボート・ラインが具体化されるとデバイス・プロセスはアボート状態になる．アボート・ラインが具体化された時点で処理中のメッセージがあれば，それは中断されるものとする．ラインが張られていない場合のアテンションやアボートの要求は無視される．

以下に，ウィンドウを例にとり，ユーザからの割り込みと，それによるIO処理の中断を順を追って説明する．

1. ユーザ割り込み

   ウィンドウが割り込みのキー入力を検出すると，その旨をアテンション・ラインを具体化することで，ユーザ・プロセスに通知する．ユーザ側では，アテンション待ちプロセスがアテンション・ラインの具体化を待っている．

2. IOの中止

   ユーザ・プロセスが割り込みを受け取り，その結果としてその時点までのIO処理を中断する必要を生じた場合は，アボート・ラインを具体化する．ウィンドウ・プロセス側では，アボート・ラインの具体化によりその時点でのIO処理を中断し，アボート状態になる．以後，resetメッセージが到着するまでのIO要求は全て無視される．

3. IOの再開

   IOを中断した後で再度IOを再開するには，ユーザ・プロセスからresetメッセージをデバイスに送出する．

このようにresetメッセージをストリームに流すことで，それぞれのアボート・ライン，アテンション・ラインがストリーム中を流れるIO処理メッセージに及ぼす影響の範囲（スコープ）を明確にすることができる．つまり，それぞれのラインのスコープは，そのラインを張るためのresetメッセージから次のresetメッセージまでとなる．

もし，割り込みの結果IO処理を中断しないのであれば，単にアテンション・ラインを張り直すためにresetメッセージをデバイスに送ればよい．

PIMOSでは，アボートによる処理の中断後，アボートされた処理を再開できるようになっている．この機構は，デバイス・プロセスがアボートされたメッセージを内部のバッファに保持しておき，resendメッセージを受け取ることで保持されたメッセージを再度処理する．

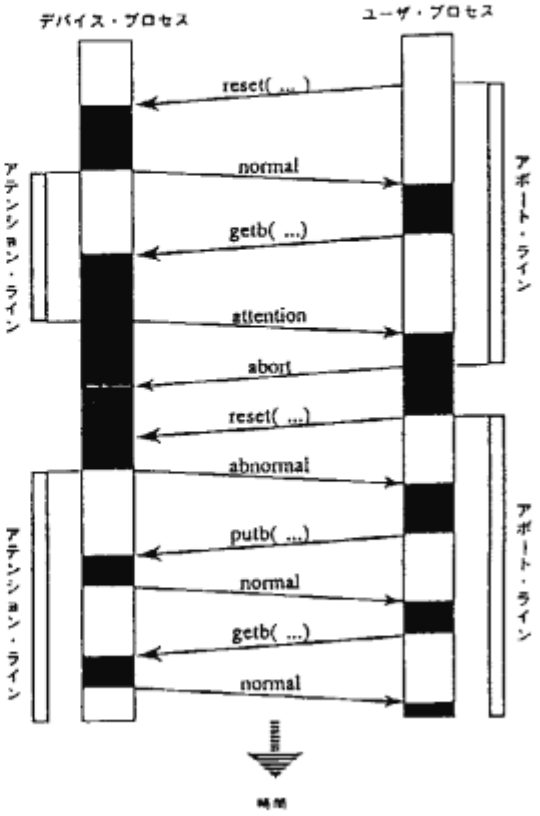


図2：ラインによる非同期処理の例

5 おわりに

1本のストリームにラインと呼ばれる機構を用いることで，ストリームを逆行するアテンションとメッセージを飛び越えた処理の中断を実現する方式について述べた．複数のストリームを用いる場合に比べ，機構が比較的単純でありながら，目的を果たすことが可能になった．この方式は，既にPIMOSにおけるデバイスを制御するストリームに実装されている．

この機構は，デバイスに限らず非同期通信が必要なプロセス間通信に一般的に応用可能である．

6 謝辞

本研究に関して有益な助言を頂いたICOT第2研究室の近山室長をはじめとするPIMOSの開発メンバーに深く感謝する．

7 参考文献

[1] 佐藤他：「並列論理型OS-PIMOS(1)」，第35回情処全国大会，4D-3，1987-9

[2] 堀他：「PIMOSの入出力管理」，第36回情処全国大会，2D-2，1988-3

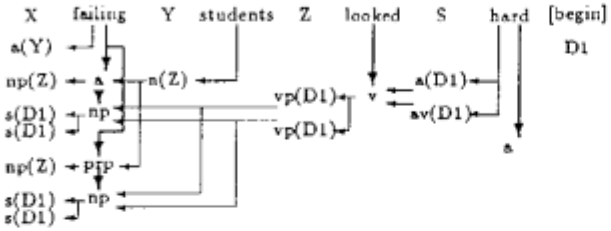


図 5: 改良版 PAX による文の解析例

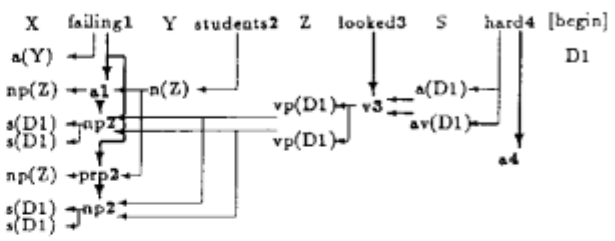


図 6: 改良版 PAX による文の解析例 (負荷分散付き)

ら流れてくる conj(Y) を (h) の第 2 節によって受け付けて np_conj を呼び出し、それが Y から流れてくる np を (i) の最初の節によって受け付けて、新しい np を呼出している様子が示されている。この時、変数 Y が、conj により、変数 Z が np により渡され、最終的に Z が新しい np の入力引数として渡されている。非終端記号が入力引数を運んでいたのは、このように新たな非終端記号が生まれた時に、正しい入力データを渡すためである。

3.3 改良版 PAX での解析例

"Failing students looked hard." という文を構文解析する例を示す。この文を解析するには、以下のようなゴールを呼び出す。

```
(l)  ?- fin(X1), merge(X, X1),
        failing(Y1, X), merge(Y, Y1)
        students(Z1, Y), merge(Z, Z1),
        looked(S1, Z), merge(S, S1),
        hard([begin], S).
```

最終的に求めたいものは、文すなわち s であるから、解析が成功するのは、与えられた文の単語を全て用いて s が構成された時である。これは、ストリーム X から s([begin]) が送られた時であり、(l) の fin という述語は、この s([begin]) を理解するための述語であり、文の解析が成功するのは、fin が s([begin]) を受け付ける時、かつその時に限る。

図 5 に、PAX による構文解析の様子を示す。図中の太い矢印は、プロセスの起動 (述語の呼び出し) に対応し、細い矢印は、(非) 終端記号の通信に対応する。また、X, Y, Z, S は、単語間に置かれた共有変数で、各プロセス間で通信するために、ストリームとして使われている。

改良版の PAX も、オリジナルの PAX と同様にストリームを階層的に使用してそれぞれのプロセスが通信を行う。

3.4 負荷分散方式

改良版の負荷分散方式は、当面オリジナルの負荷分散方式と同じにする。つまり、単語間を結ぶストリームにプロセサを割り付け、最初に起動する単語に相当するプロセスをそのプロセスの右のストリームと同じプロセサ上で起動する。そして、各プロセスが、ストリームからのデータを読み込もうとすると、そのストリームに割り付けられたプロセスに移動する方式である。

この負荷分散方式を採用するために各述語に以下の節を追加する。

```
(m)    inter_node([PE|In], Out) :- integer(PE) |
                inter_node(In, Out)@processor(PE).
```

また、前に述べた例文を解析する場合は、以下のように各述語を起動する。

```
(n)  ?- fin([0|X1])@processor(0),
        merge(X, X1)@processor(0),
        failing([1|Y1], X)@processor(1),
        merge(Y, Y1)@processor(1),
        students([2|Z1], Y)@processor(2),
        merge(Z, Z1)@processor(2),
        looked([3|S1], Z)@processor(3),
        merge(S, S1)@processor(3),
        hard([begin], S)@processor(4).
```

この例文の解析で、各プロセスがどのプロセサで実行されるかを示したのが、図 6 である。プロセス名の横に示した数字が、実行されるプロセサを示している。

改良版の PAX では、プロセス間の通信量がオリジナルの PAX に比べて少ないので、別の負荷分散方式も考えられるかも知れない。

3.5 オリジナル PAX への改良方式の適応

今回の改良方式をオリジナル PAX に適応する方法を考えてみる。

今回の改良では、識別子の代わりに (非) 終端記号をデータとして受渡しすることにより、プロセス間で通信されるデータ量を抑えているが、オリジナル PAX でも、識別子の数 (種類) を減らすことで、プロセス間の通信量を抑えることがある程度可能である。

オリジナル PAX での識別子は、文法規則内の特定の位置を示していた。しかし、識別子の本来の意味は、文法規則の左隅からどういう (非) 終端記号列を解析できたかを示すことである。従って、文法規則の左隅から同じ (非) 終端記号列が続く限り、同じ識別子を割り付けることが可能である。例えば、文法規則 (1) ～ (14) は、以下のように識別子を重複して割り付けることができる。

レコードの Rid の集まりがまず求められる．これを集合と呼ぶ．複雑な条件の検索は，単純な索引を利用した検索とその結果である集合間の演算 (Union, Intersection, Difference) により求められる．レコードの構造に木構造を許す非正規関係では Rid は複雑な構造になる場合があるが，ここではレコード全体の選択情報を示す Rid(整数値) のみからなる集合間の演算に限定した評価をおこなった．

3.1.1 データ構造

整数値のみからなる集合を，あらかじめ決められた値の範囲によって分割し，それぞれ独立して演算を行なえるような構造とした．

```
<Set> ::= "[" <Partition> { "," <Partition> } "]"
<Partition> ::= "{" <Quantity> "," "[" <Rid> { "," <Rid> } "]" "}"
<Quantity> ::= INTEGER
<Rid> ::= INTEGER | "{" <FromRid> "," <ToRid> "}"
<FromRid> ::= INTEGER
<ToRid> ::= INTEGER
```

ただし，次の制限がある．

- Rid は前から昇順に並んでいる
- Partition に入る Rid の範囲は静的に決められている

3.1.2 演算

Union, Intersection, Difference の各演算を，マージ アルゴリズムを使って求める．

3.1.3 計測と評価

この集合演算では，Partition 間に渡る演算はなく Partition の数だけの並列度が見込まれる．この演算は共有メモリを仮定しているので，同じ範囲をとる Partition はマルチ PSI の同じプロセッサ上に割り当てられるようにデータを配置して計測した．

10 万の集合要素を含む集合どうしの集合演算 (Union) を 64 台版マルチ PSI で計測した時の台数効果を表 1 に示す．集合要素数と PE 台数変えて測定した結果，一つの PE に 7,000 から 8,000 の集合要素を割り当てた時最も台数効果が現れることがわかった．また，分割数は PE 台数と等しいとき，もっとも台数効果があり，PE 台数の 3 倍程度までは著しい低下は見られない．また，演算の処理速度は，結果として求まる集合の要素数はあまり関係せず，比較の回数に依存する．

これらから，実際の集合の演算には全レコード件数を 7,000 で割った数だけの PE があれば十分であり，分割数は，選択される Rid のばらつきを考えて，集合演算に使用する PE の数の 2 倍ぐらいにすれば良いと想像できる．

表 1 集合演算の台数効果

PE の数	2	4	8	16	32	64
奇数集合同士	1.97	3.86	7.61	14.30	24.46	37.93
乱数集合同士	1.99	3.92	7.63	14.80	27.3	—

3.2 レコード操作

主記憶構造に適したレコードの構造を調べることを目的に，評価をおこなった．ただし，レコード操作は軽い処理であり，関係代数演算にはレコード操作を単位とする並列性がレコード数だけ存在するので，レコード操作に含まれる並列性を引き出すことを第一に考えてはいない．レコード操作については，KL1 で記述することの面白みは特にないため，簡単な記述にとどめる．

評価では，二次記憶データベースとの変換時の手間が少ないレコードのストリング表現と，二次記憶データベースとの変換時の手間が多いがその他のアクセスは効率的と予想されるレコードの構造体表現とを実装し，レコード操作の基本操作 (生成 / 読み / 削除 / 追加) についてマルチ PSI で 1 台の PE に割

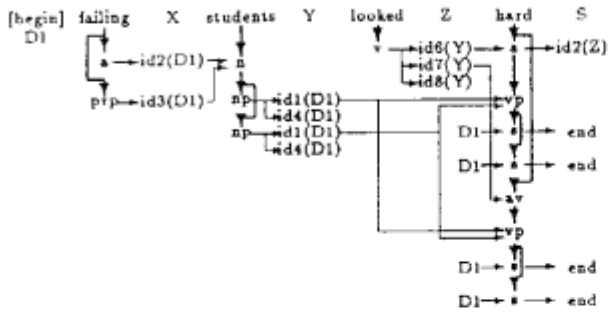


図 2: PAX による文の解析例

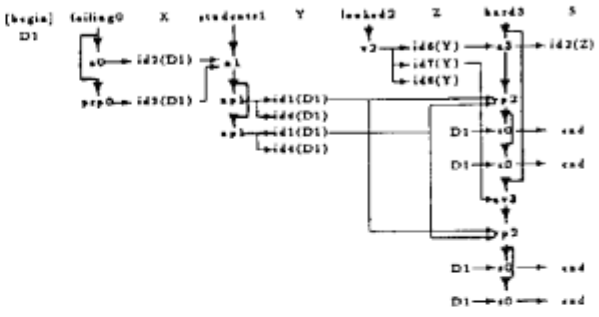


図 3: 負荷分散を行った文の解析例

るので、それぞれの組み合わせで 4 つの解となる。

入力文の単語から始まって、構文解析木を作り上げていく過程が、プロセス間でデータ (識別子) を通信することで行われているのがよくわかる。このように PAX では、最初に単語間に張られた共有変数をストリームとして階層的に使用することで、それぞれの (非) 終端記号に相当するプロセス間で通信しながら解析を進めていく。このようにストリームを階層的に使用する方式を、レイヤード・ストリーム手法 [Okumura87] と呼んでいる。

2.3 負荷分散方式

PAX のアルゴリズムには、次の 2 点に並列性があると考えられる。

and 並列: 独立した非終端記号を生成する過程に並列性がある。

or 並列: 1 つの非終端記号を生成する際に、複数の可能性がある場合、それら複数の可能性同士に並列性がある。

上で述べた 2 つの並列性はそれぞれ次のような特徴を持つ。

and 並列: これは、最初に起動する各単語に相当するプロセスを並列に実行することに相当する。従って、並列度は実行開始時が一番大きく最大 (単語の数) であり、解析が進むにつれて並列度が下がると考えてよい。

or 並列: これは、文法規則に曖昧性があれば、それらを並列に実行することに相当する。従って、並列度は文法規則の曖昧性に依存するが、and 並列より大きいと考えてよい。

次に、プロセサ間通信の観点から負荷分散を考えてみる。PAX のプロセスはタイプ 2 が多く、左の兄弟からのストリームを読んで、右の兄弟へのストリームへデータを送るという動きをする。そのため、左右の兄弟が別プロセサで実行されているとすると、多対多という莫大なプロセサ間通信が起き、並列実行の効果が現れなくなってしまう。従って、上記の and 並列 / or 並列 どちらを取っても、プロセサ間通信が多くなり、並列実行の効果は期待できない。

そこで、プロセサ間通信を抑える負荷分散方式が考えられた。これは、単語間を結ぶストリームにプロセサを割り付け、最初に起動する単語に相当するプロセスをそのプロセスの左のストリームと同じプロセサ上で起動する。そして、各プロセスが、ストリームからのデータを読み込もうとすると、そのストリームに割り付けられたプロセスに移動する方式である。この負荷分散方式では、同じストリームを読むプロセスが 1 つのプロセサに集まるので、多対 1 のプロセサ間通信しか起きない。また、この負荷分散方式では、データの代わりにプロセスをプロセサ間で移動することになるが、データ移動とプロセス移動のコストの比率より 1 つのプロセスが多くのデータを読み込む場合に、この負荷分散方式の長所が現れると言える。

この負荷分散方式を採用するためにタイプ 2 の述語に以下の節を追加する。

```
(f)     node([PE|In], Out) :- integer(PE) |
                node(In, Out)@processor(PE).
```

また、前に述べた例文を解析する場合は、以下のように各述語を起動する。

```
(g)  ?- failing([0,begin], X)@processor(0),
        merge(X, X1)@processor(1),
        students([1|X1], Y)@processor(1),
        merge(Y, Y1)@processor(2),
        looked([2|Y1], Z)@processor(2),
        merge(Z, Z1)@processor(3),
        hard([3|Z1], S)@processor(3),
        merge(S, S1)@processor(4),
        fin([4|S1])@processor(4).
```

この例文の解析で、各プロセスがどのプロセサで実行されるかを示したのが、図 3 である。プロセス名の横に示した数字が、実行されるプロセサを示している。

3 改良版 PAX

3.1 改良の概要

以上のようにオリジナルの PAX では、プロセス間で通信が多いという特徴を持っている。従って、プロセス間の通信量 (データ量) を減少させる改良が必要となる。

```
<ベクタ版B+木> ::= "{" "}" | <ノード> | <リーフ>
<ノード> ::= "{" <キー数> "," <ノードリスト> "}"
<ノードリスト> ::= "[" <ベクタ版B+木> "," <キー>  "," <ベクタ版B+木>
                    { "," <キー>  "," <ベクタ版B+木> } "]"
<リーフ> ::= "leaf(" "{" <キー> "," <データ> "}" { "," "{" <キー> "," <データ> "}" } ")"
<キー> ::= INTEGER
```

T 木のデータ構造

T 木は主記憶での木構造に適したものとして，二進木である AVL 木の変形として考案された．AVL 木との違いは，ノードに複数個のデータ格納を許し，リバランスによるデータの移動を減らすことができるとしている．我々は，木に対する順次アクセスの性能も重視したため，B^+ 木と同じ理由から，ノードをプロセスにし順次アクセス用のリンクを持つものと，ベクタで表現し順次アクセス用のリンクを持たないものの二種類を試作した．

(1) リーフノード プロセス版 T 木

リーフノードをプロセスとした T 木のデータ構造を簡略化して示す．これを今後，プロセス版 T 木と呼ぶ．

```
<プロセス版T木> ::= "{" "}" | "{" <左プロセス版T木> "," <制御情報> ","
      <右プロセス版T木> "," "{" <最小値> "," <最大値> "," <データ保持プロセス> "}" "}"
<制御情報> ::= "leftdown" | "balanced" | "rightdown"
<データ保持プロセス> ::= "{" <左右データ保持プロセス> ","
           "{" <キー> "," <データ> "}" { "," "{" <キー> "," <データ> "}" } "}"
<左右データ保持プロセス> ::= "{" <左データ保持プロセス> ","  <右データ保持プロセス> "}"
<キー> ::= INTEGER
```

(2) リーフノード ベクタ版 T 木

リーフノードをベクタとした T 木のデータ構造を簡略化して示す．これを今後，ベクタ版 T 木と呼ぶ．

```
<ベクタ版T木> ::= "{" "}" | "{" <左ベクタ版T木> "," <制御情報> ","
            <右ベクタ版T木> "," "{" <最小値> "," <最大値> "," <データリスト> "}" "}"
<制御情報> ::= "leftdown" | "balanced" | "rightdown"
<データリスト> ::= "{" "{" <キー> "," <データ> "}" { "," "{" <キー> "," <データ> "}" } "}"
<キー> ::= INTEGER
```

3.3.2 演算

木に対する演算として，検索 / 追加 / 削除および範囲検索を実現した．

- 検索処理　　キーを指定しそれに対応するデータを木から取り出す処理
- 追加処理　　キーとデータのペアを木に対して追加する処理
- 削除処理　　キーを指定しそれを木から削除する処理
- 範囲検索　　ある範囲に存在するキーとデータを順に取り出す処理

処理方式に特徴があるのは，範囲検索に対する処理である．以下にその処理概要を示す．

- リーフノードがプロセスで実現されている木
  リーフノード間にパスがあるため，まず最小値を検索しそれ以降はリーフノード間のパスに沿って最大値以上のものが現れるまでを検索結果として返す．比較演算の回数は，最小値にたどり着くまでと，最大値にたどり着くまでのに通過するリーフノード中の最大値との比較と，最大値を含むノード内で最大値にたどり着くまでの比較になる．

並列自然言語構文解析システム PAX の改良

佐藤 裕幸

三菱電機 (株) 情報電子研究所

〒247 鎌倉市大船 5-1-1

Tel: 0467-46-3665, E-Mail: hiroyuki@isl.melco.co.jp

概要

近年、自然言語処理、特に自然言語構文解析の並列化が活発化しており、新世代コンピュータ技術開発機構 (ICOT) でも並列自然言語構文解析システム PAX (Parallel Analizer for syntaX and semantiCS) の研究が行なわれている。

これまでの PAX の研究で、構文解析の処理の過程でプロセサ間の通信量が非常に多いため、複数のプロセサを用いても、処理速度があまり向上しないということが報告されている。そこで、今回、PAX のプロセサ間通信を減少させる改良を行なったので、改良方式及び測定結果を報告する。

PAX のプログラムは、通信量が減少することにより実行時間も減少するという性質を持っている。マルチ PSI 上での測定の結果、改良によって、通信量は $\frac{1}{6}$ に減少し、単一プロセサ上では最高 3 倍強の速度向上が得られた。また、今回の改良は、逐次型の自然言語構文解析システム SAX にも適応できる。

なお、この改良方式の元々のアイデアは、ICOT の瀧氏によって考案されたものであり、筆者が実現 / 評価し、その効果を実証した。

1 はじめに

自然言語処理は多大な計算量を必要とし、処理速度の向上が望まれているプログラムの 1 つである。そのため、近年、自然言語処理、特に比較的アルゴリズムが確立している自然言語構文解析の並列化が活発化しており、新世代コンピュータ技術開発機構 (ICOT) でも並列自然言語構文解析システム PAX (Parallel Analizer for syntaX and semantiCS) の研究が行なわれている。

PAX は、逐次型の自然言語構文解析システム SAX (Sequential Analizer for syntaX and semantiCS) [松本 86] の並列版である。SAX の構文解析アルゴリズムは、並列性を持っているだけではなく、バックトラック及び副作用を用いないため、逐次計算機上でも高速に処理できる。

しかし、これまでの PAX の研究で、構文解析の処理の過程でプロセサ間の通信量が非常に多いため、プロセサ間通信のコストが高いマルチ PSI のような疎結合並列計算機上では、複数のプロセサを用いても、処理速度があまり向上しないということが報告されている [寿崎 89]。

そこで、今回、PAX のプロセサ間通信を減少させる改良を行い、オリジナルの PAX との比較した。

まず最初に、オリジナル PAX の構文解析方法と負荷分散方式について解説する。次にそれをどのように改良したかを記述した後で、この改良方式のオリジナル PAX への適応方法について解説する。そして最後に、今回改良した PAX をいくつかの項目についてマルチ PSI 上で測定したので、その結果及びオリジナル PAX との比較を報告する。また、付録として、簡単な文法に対応する PAX のプログラムを添付しておく。

2 オリジナル PAX

2.1 オリジナル PAX での解析方法

PAX は、逐次型の自然言語構文解析システム SAX の並列化版である。SAX のアルゴリズムは、基本的には左隅構文解析法であり、あるまとまった非終端記号 (または、終端記号) が構成されると、それを基にして、より大きな構文木を構成しようとするが、その時、その (非) 終端記号を左隅の要素とする新しい解析木を作ろうとする。つまり、この非終端記号を右辺の先頭要素として持つ文法規則を選び、その文法規則を完成させようとする。このような文法規則の右辺の 2 番目以降の要素は、新しい (非) 終端記号が得られるたびに、埋め合わされ、完全な解析木へと作り上げられていく。

このような処理を (並列) 論理型言語で実現するために、文法規則の右辺に、以下のようにそれぞれの場所を示す識別子を設ける。

```
(1) s   → np, id1, vp.   (2) np → a, id2, n.
(3) np  →  prp, id3, n.
(4) np  →  np, id4, conj, id5, np.
(5) vp  → v, id6, a.     (6) vp → v, id7, av.
(7) vp  → v, id8, np.    (8) a  → [failing].
(9) n   → [hard].        (10) av → [hard].
(11) n  → [students].    (12) v  → [looked].
(13) conj→ [and].        (14) prp→ [failing].
```

id1 から id8 までが識別子であり、文法規則内の特定の位置を表している。例えば、id1 は (1) の文法規則の np までの解析が終ったことを表している。PAX では、解析された終端記号および非終端記号がすべて KL1 のプロセス

全体としては，範囲検索以外ではベクタ版がプロセス版の二割ほど速く，プロセスにした時の遅さが現れた．範囲検索は四つの間にあまり差はない．ここでは1PEで測定したが，プロセス版が逐次的な処理なのに対し，ベクタ版は並列性がかなりあるので，PIMのクラスタで実行する場合，ベクタ版が有利となると予想される．T木とB^+木との比較では[Lehman 86]の結果ほど大きな差にはならなかった．単一代入を基本とするKL1と副作用を許すC言語の違いが原因であると予想している．

4 まとめ

並列データベース管理システムの基本要素として，ローカルDBMSでの処理で基本となる，集合操作，レコード操作，索引操作について試作をおこないマルチPSI上で計測をおこなった．ただし，試作ではローカルDBMS内の主記憶データベース機能に限定し，共有メモリを持つクラスタで実行されることを前提として設計をしている．

評価結果の簡単なまとめをおこなう．

- 集合操作
  ローカルDBMSでの処理でも並列性はかなり存在し，しかもPIMのクラスタでの実行で，それが高速化につながることが明らかになった．
- レコード操作
  ベクタを利用したレコードからは，その記述性と処理速度ともに満足のいく結果がでたが，ストリングを利用したレコードからは，ストリング処理能力の改善が必要であるという結果がでた．
- 索引処理
  並列処理に向いた範囲検索の方法が見つかった．ただし，これは単一参照を守った場合の結果であり，一般的に言えるのかどうかは疑問である．

その他に，ここではデータとして出さなかったが，KL1で記述しIPEで実行しても，ESPで記述したKappa-IIと同等以上の速度が期待できることが分かった．

データベース管理システムをPIM/PIMOSで作るにあたってネックになりそうなのは，レコード操作で明らかになったストリング処理能力である．ストリング処理能力は二次記憶データベースで非常に重要であるため，ストリング処理のファームウエア化が望まれる．詳しくは，ストリングの部分構造へのアクセス(substring,copy_string_elements,move_string_elements)や，ストリング内の要素検索(search_character)，ストリングの比較(less_than,not_less_thanまたはcompare_string)などである．

また，考えがまとまらなかったので本稿では触れなかったが，並列データベース管理システムなどの分散処理指向のシステムで分散トランザクションやデータの複製をおこなうことを考えると，PIM/PIMOSでどんな種類の障害が発生する可能性があって，それに対しどんな対応をPIM/PIMOSが行ない，それをPIM/PIMOS上のシステムがどこまで知ることができるのかということも重要である．

参考文献

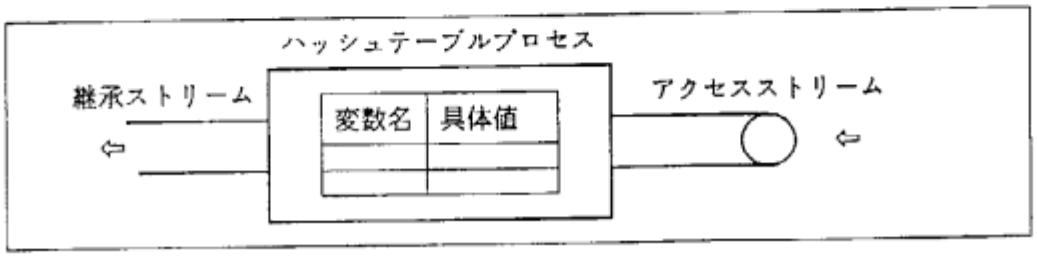
[Lehman 86] Tobin J. Lehman and Michael J. Carey, A Study of Index Structures for Main Memory Database Management Systems, Twelfth international conference on very large data bases, 294-303(1986).

うことによって制約式が解かれる．

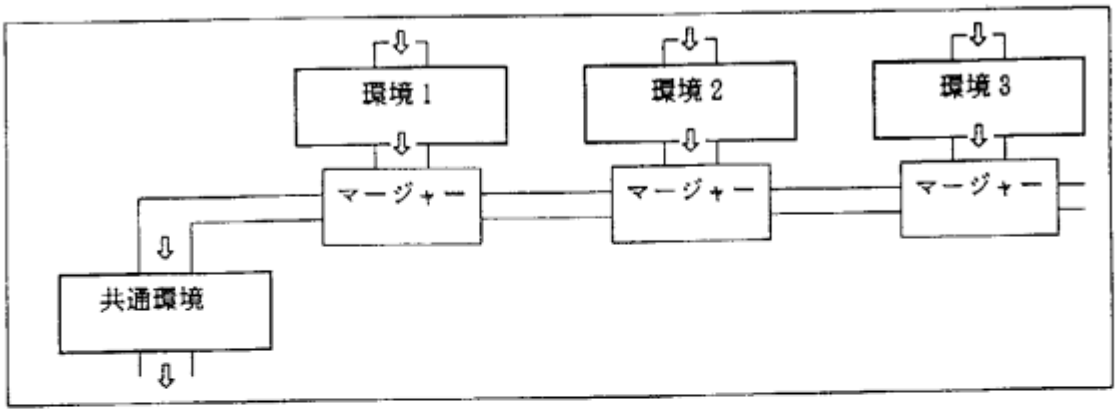
（3） 多重環境の実現方法

現在の実装法では，変数の環境は変数名をキーとするハッシュ表を内部に持つプロセスとして実現している．変数の環境の参照及び更新は，このプロセスのアクセスのためのストリームを介して行う．

ただし，このプロセスは継承のためのストリームも持っていて，自分のテーブルに該当するキーが存在しないときには，継承ストリームに自分が受け取ったアクセス要求を参照の要求に変換して渡す．参照や更新した結果は結局は自分のテーブルに追加する．したがって，継承するストリームの先にあるプロセスのテーブルは参照するだけで更新することはなく，また，一旦参照すると自分の場所に情報を追加するので，２度目からは継承先を探索することはない．



多重環境は，和型によって探索が分岐するとき，および入力ストリームから新たなデータを読み込んだ時に新たなハッシュ表プロセスを生成し，この新しいプロセスがマージャを介して古い環境を継承することによって実現している．この方法は，環境を全て複写するのではなく，環境の差分のみを新規に生成することになっている．

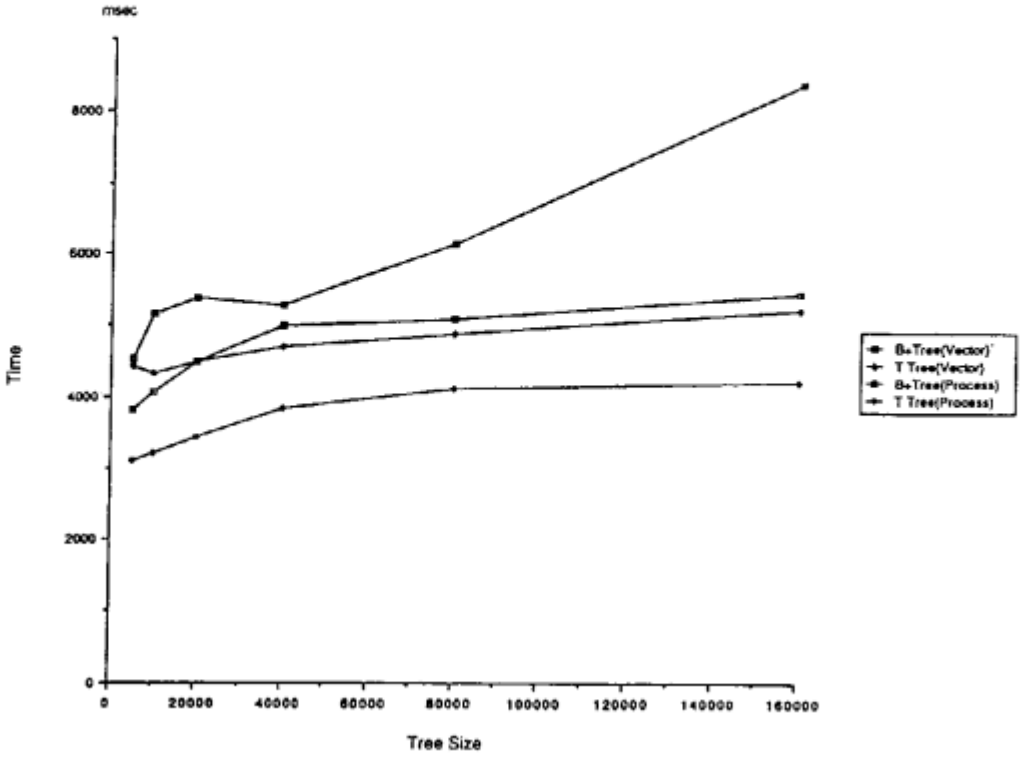


（4）環境の複写

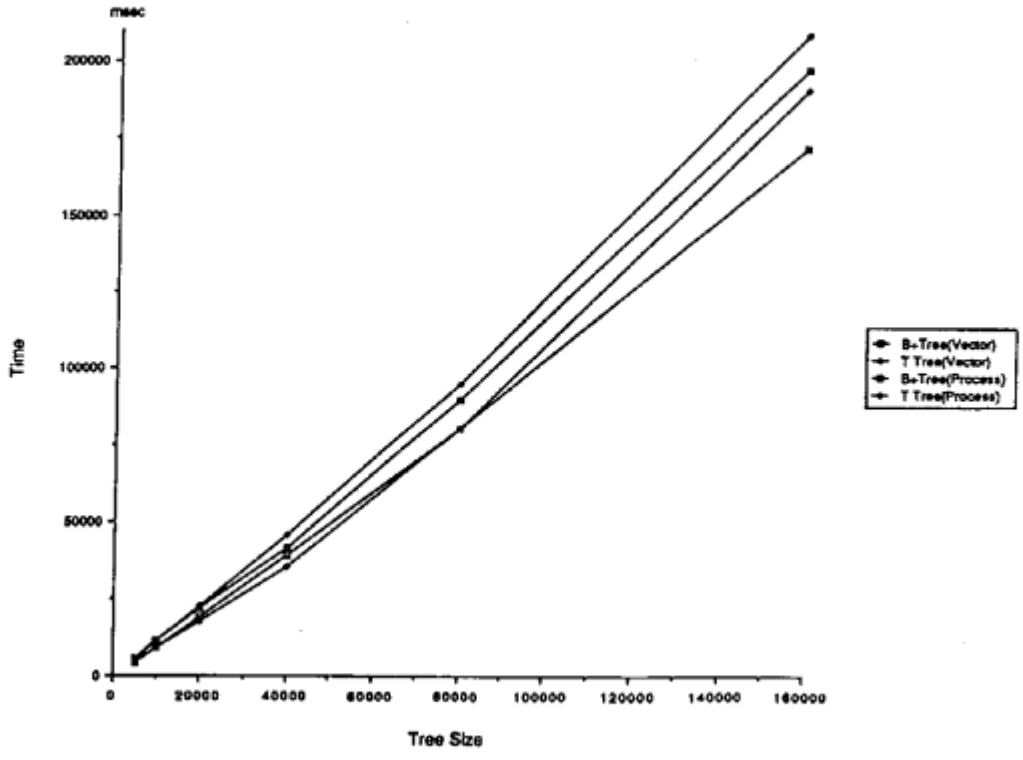
非終端記号までリデュースされた時に，完成した非終端記号のインスタンスとしてその環境が持っている情報の内容は，次にその非終端記号を受け取ったプロセスで，文法との適合検査を行うときに参照され，適合するときにはその環境にマージされなければならない．

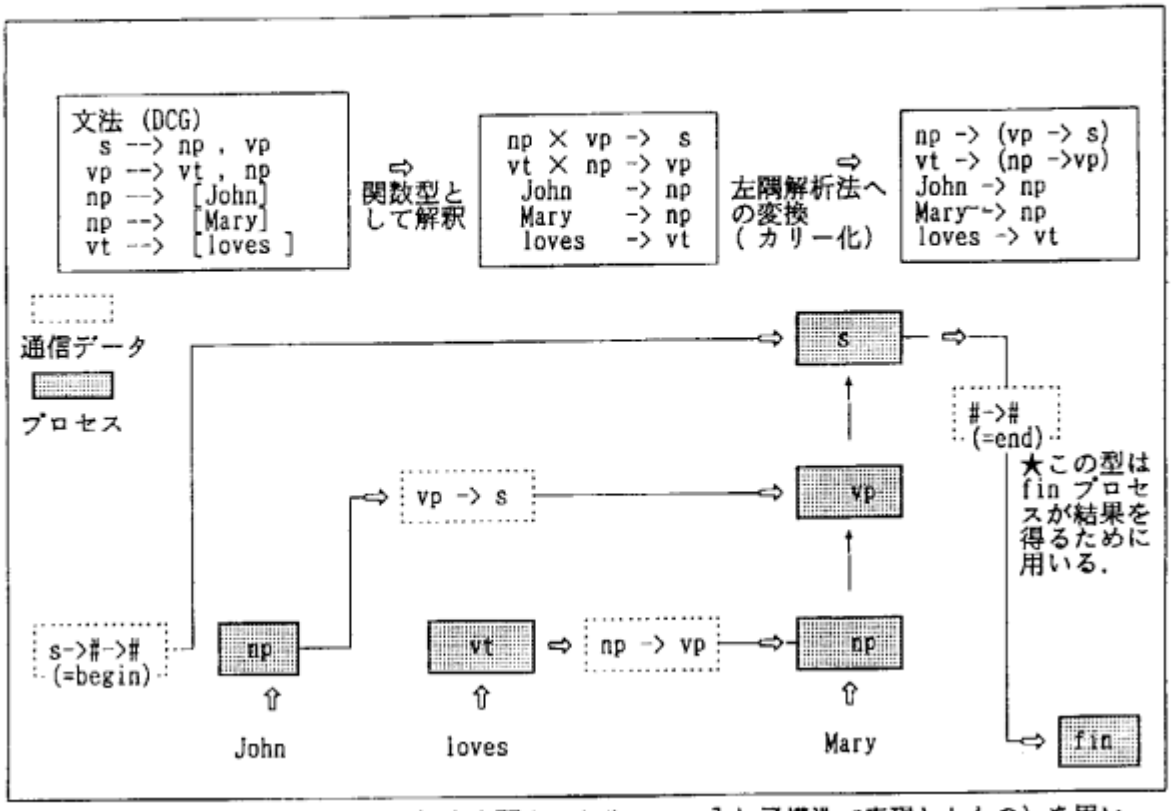
このために，非終端記号を構成するために作られた環境は，非終端記号とともに通信データとして送られる．このとき，非終端記号を受け取るプロセスでは，受け取った非終端記号毎に新しい環境を作成し，その環境を複写する必要がある．

グラフ 3. 削除処理



グラフ 4. 範囲検索処理





ＰＡＸはレイヤードストリーム（探索空間をストリームの入れ子構造で表現したもの）を用いている前の例の lovesと Mary　の間のストリームは（処理が完了した後では）次のような構造になっている.

```
[(np ->vp)* [(vp -> s)* [(s->#->#)* [] ] ] ]
```

これに，表現と型の表示（表現: 型）を明示的に付加すると次のようになる.

```
[  loves :(np->vp)* [John:(vp->s)*  [ {} :(s->#->#)*  [] ] ] ]
```

loves と Mary の間のストリームの第1 の階層は，loves で終わる表現列の対する型のバリエーションを表示している．一方，ストリームの階層は，一つ内側の階層は，その外側の階層に対応している表現列の直前でおわる表現列に対する型を表示している．（例えば，loves　に付随しているストリームの中に John の型が現れる）また，レイヤードストリームは，表現列の区切りごとに与えられた型のバリエーションを '*' という演算の分配法則を用いて共有できる部分を共有させている．'*' という演算は次のようなものである.

```
(em+1,...,en :(T1->...->Ti))* (e1 ... em  :(Ti->Ti+1,...->Tj) )
  = e1 ... em1 em+1...en  :(T1->...->Ti-1 -> Ti+1->...->Tj)
```

ここで型を命題と解釈すると，この演算は次のような推論規則になる

$$\frac{T1 \supset \ldots \supset Ti \qquad Ti \supset Ti+1, \ldots \supset Tj}{T1 \supset \ldots \supset Ti-1 \supset Ti+1 \supset \ldots \supset Tj}$$

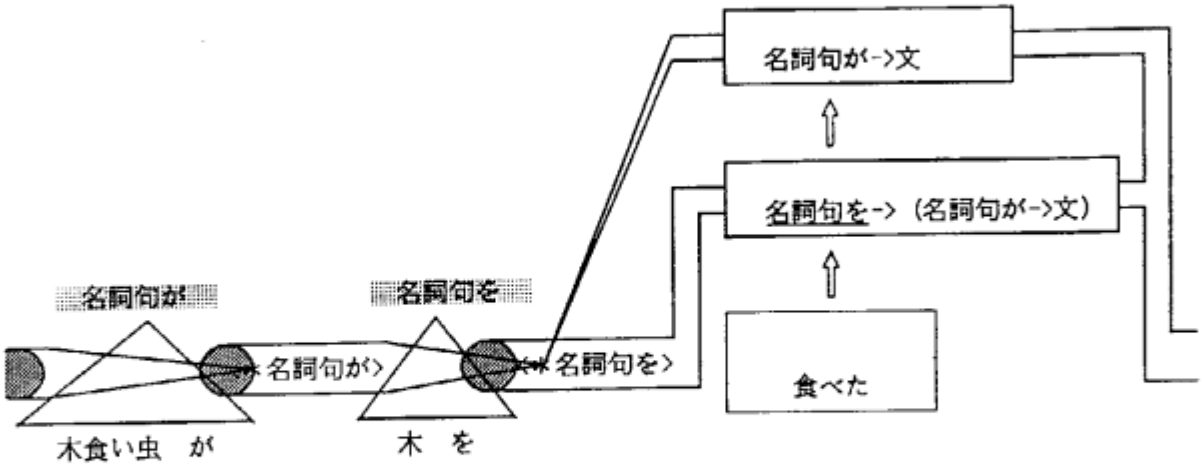
```
        freeze_term(Goals,VarID1,VarID2),
        put_goals(Goals,[],GoalQueue),
        pes(Clauses,Goals,GoalQueue,VarID2,[],Reports).

    pes(_,Goals,[],_,BindTBL,Reports) :- true |
        eval_term(Goals,EGoals,BindTBL),
        Reports=[exit(EGoals)].
    pes(Clauses,Goals,GoalQueue,VarID,BindTBL,Reports) :- true |
        get_goal(GoalQueue,GoalQueue1,BindTBL,Goal),
        exec_body(Goal,RGoals,VarID,VarID1,BindTBL,BindTBL1,Clauses,Result),
        eval_term(GoalQueue,EGoalQueue,BindTBL),
        eval_term(Goal,EGoal,BindTBL),
        eval_term(RGoals,ERGoals,BindTBL1),
        Reports = [goal_queue(EGoalQueue),{Result,EGoal,ERGoals} | Reports1],
        put_goals(RGoals,GoalQueue1,GoalQueue2),
        pes1(Result,Clauses,Goals,GoalQueue2,VarID1,BindTBL1,Reports1).

    pes1(fail,_,_,_,_,_,Reports) :- true |
        Reports = [].
    otherwise.
    pes1(_,Clauses,Goals,GoalQueue,VarID,BindTBL,Reports) :- true |
        pes(Clauses,Goals,GoalQueue,VarID,BindTBL,Reports).
```

述語 mi(Clauses,Goals,Reports) は, メタインタプリタのトップレベルのゴールであり, Clauses にプログラム, Goals に起動ゴールを与えて呼び出すと, 報告ストリーム Reports にトレース情報を返す. このメタインタプリタでは全ての変数は, 述語 freeze_term(X, VarID, NewVarID) によりフリーズされた形で扱われる. X はフリーズする入力項であり, freeze_term/3 は X に含まれる未定義変数全てにベクタ $var(ID) をユニファイする. また, フリーズ処理が終了すると, NewVarID に ID を更新した値をユニファイする. ID は変数識別番号 VarID から始まる整数であり, 変数毎にユニークな変数識別子である. 述語 mi/3 では, プログラム Clauses, 及び起動ゴール Goals を freeze_term/3 によりフリーズしている. また述語 put_goals(Goal,GoalQueue, NewGoalQueue) は, ゴールキュー GoalQueue にゴール Goal を格納し, 格納済みのゴールキューを NewGoalQueue にユニファイする. 述語 mi/3 内では, キューの中に起動ゴール Goals を格納している.

述語 pes(Clauses,Goals,GoalQueue,VarID,BindTBL,Reports) は, 実際にゴールの評価を行なう述語である. Clauses,Goals は freeze_term/3 によってフリーズされたプログラム, 起動ゴールであり, GoalQueue はゴールキュー, VarID は変数識別番号, BindTBL は変数の束縛表, Reports は報告ストリームである.

pes/6 は GoalQueue が空リストになるまでゴールの評価を繰り返す. 先ず, get_goal(GoalQueue, GoalQueue1,BindTBL,Goal) により評価すべきゴールをゴールキューより 1 つ取り出し, 変数 Goal にユニファイする. GoalQueue1 には Goal が取り除かれたゴールキューがユニファイされる. 述語 exec_body(Goal,RGoals,VarID,VarID1,BindTBL,BindTBL1, Clauses,Result) は, ゴール Goal のリダクションを GHC の実行規則に従って試みる. すなわち, ゴール Goal とパッシブユニフィケーション可能な節がプログラム Clauses 内にあるならば, これを試み, 成功する節があるならばその節にコミットし, そのボディ部をリダクションの結果生成された子ゴールとして変数 RGoals に, コミット成功を示すアトム succ を変数 Result にユニファイする. この変数 RGoals

するストリームである．



この実装方法では，ストリームは左から右にデータを流すので，文法規則の右隅の要素に対応する表現をもとに型にプロセスを生成し，その左側の表現と結合して構文木が構成されていくような，右隅解析法になっている．

しかし，右隅解析法の採用は特に本質的ではない，もし我々のシステムで左隅解析法を用いると反対にストリームの中の通信データは右から左に流れることになるというだけの違いである．しかし，日本語では，助詞や述語のような文法的構造を決定する語や句は右隅に現れる傾向があり，それ以外の要素は比較的自由な順序で出現するので，右隅解析法の方が有利であると判断した．

（2）非終端記号が構成されたときの処理

もうひとつの動作は，文法規則の適用の結果，非終端記号が構成されたときの処理である．また，最初に入力された終端記号／非終端記号に対しても基本的にこれと同様の処理を行う．このとき，二つの処理が対になって発生する．

一つは，構成された非終端記号（または終端記号）を現在処理を行っているプロセスの入力ストリームと対にして出力ストリームに流すことである．

もうひとつは，その非終端記号（または終端記号）を表現とみなして辞書／文法にアクセスし，その表現の型を得て，型が「->」の記号を含む場合にプロセスを生成することである．一般に文法や辞書に曖昧性が含まれる場合，型は和形の形で記述されているが，これに対してＯＲ並列的にプロセスを生成し，入力ストリームも同一のストリームをこれらのプロセスの入力ストリームとする．

```
        VarID1,BindTBL1,Reports1).
pes1(fail,_,_,_,_,_,_,_,Reports) :- true |
    Reports = [].
otherwise.
pes1(_,ProcID,NumOfProc,Clauses,Goals,GoalQueue,
    VarID,BindTBL,Reports) :- true |
    pes(~(ProcID + 1),NumOfProc,Clauses,Goals,GoalQueue,
        VarID,BindTBL,Reports).
```

述語 pes/8 は, 前述の述語 mi から pes(NumOfProc,NumOfProc,Clauses,Goals, GoalQueue, VarID,BindTBL,Reports) という形で呼ばれる. ここで NumOfProc は全プロセッサ数である. 呼ばれた直後は GoalQueue は空ではないので 2 番目の節がコミットされ, ゴールキューの中身, すなわち起動ゴールの代入例が eval_term/3 で求められ, 報告ストリーム Reports に出力される. この出力の後, プロセッサ識別子 0 番から NumOfProc−1 番までが順に実行される. すなわち, あるプロセッサ識別子 ProcID を持つプロセッサのリダクションを試みたならば, ProcID を 1 インクリメントし, その ProcID を持つプロセッサの行なうリダクションとして実行を試みるのである. ProcID がインクリメントされてゆき, 全プロセッサ数と等しくなったならば, ゴールキューの代入例を報告ストリームに流し, 再びプロセッサ識別子 0 番よりリダクションを試みる.

前述のメタインタプリタと大きく異なるのは, ゴールキューである. このゴールキューではキュー内のゴールに, タグとしてプロセッサ識別子が取り付けられている. 述語 get_goal(ProcID,GoalQueue, GoalQueue1,BindTBL,G) は get_goal/4 に対して, 変数 ProcID で示されたプロセッサ識別子と同じ識別子のタグの付いたゴールのみを返すよう変更されている. またトレース情報の出力においてもどのプロセッサでのリダクションであるかを示すため, 出力情報にタグとして ProcID を付加している.

3 パフォーマンスモニタによる動作情報表示

パフォーマンスモニタにおける動作情報表示部は, Multi-PSI FEP 上で ESP 言語を用いて開発したものである. この表示部は, メタインタプリタから出力される動作情報を一定の形式に変換し, 対象プログラム実行時における, プロセッサ負荷状況の時間的推移を視覚的に表現することを目的としている. ここでメタインタプリタが負荷として扱っているのは, ゴールキュー中に存在するゴール数である.

動作情報の表示は, 全プロセッサの負荷状況を同時に表示するために, 時間 − 負荷 − プロセッサの三次元グラフで示した. また, 目的に合わせた表示を得るために, 三次元コンピュータ・グラフィックス [Sato 84] を用いて表示した. 動作情報を任意の視点で表示し, 変更することを可能とした. さらに, 任意の時間あるいはプロセッサに注目した表示を行うために, 時間 − 負荷及び, プロセッサ − 負荷の各二次元グラフ表示を可能とした.

3.1 三次元表示機能

三次元表示では, 座標原点側をプロセッサ 0 としており, 負荷の最も大きい点を 100% に合わせてスケールを決定している. また, グラフ上の時間は, プロセッサが一つでも立ち上がった時点を 0 とし, 全てのプロセッサに負荷がなくなるまでを表している.

3．構文解析手法

ＬａＰｕｔａの型推論機構を本章では構文解析機構として説明する．ＬａＰｕｔａの構文解析手法は，基本的にはＰＡＸ［松本 86］をもとにＩＣＯＴの瀧氏によって最初に考案され［佐藤 90］，我々が再発見したものである．ＰＡＸとの違いは，プロセスとストリームの関係が反転している点で，ＰＡＸが，ストリームの構造で探索空間を表現していたのに対して，ここで提案する方式はプロセスの親子関係で探索空間を表現するものである．

3．1．ＬａＰｕｔａとＰＡＸの得失

この方式は，ＰＡＸのように適用可能な全ての文法規則をプロセス間の通信ストリームに流すのではなく，文法規則が実際に完全に充足されたときに構成が成功した非終端記号のみをストリームに流すことによって，プロセス間の通信量を減少させている．

適用可能な全ての文法規則と実際に構成が成功した非終端記号とでは，非終端記号の方が量的に少ないのは明らかなので，通信量の観点からはＬａＰｕｔａの方式の方が有利である．通信量を削減したことのトレードオフとして処理過程で生成されるプロセスの数はＬａＰｕｔａの方が増える．

3．2．ＬａＰｕｔａの辞書／文法

ＬａＰｕｔａでは，終端記号と非終端記号の区別は無い．辞書および文法規則は次のように記述される．

人間 ： 体言語基．	(DCG で書くと	体言語基 --> 〔人間〕．)
が ： (体言語基 -> 名詞句)	(DCG で書くと	名詞句 --> 体言語基，〔が〕．)
述語 ： (名詞句 -> 文)	(DCG で書くと	文 --> 名詞句，述語 ．)

この記述法で，「 :」の左を表現，右を型と呼ぶことにする．

3．3．構文解析の基本動作

ＬａＰｕｔａ構文解析の基本動作は，大きく分けると，文法規則の充足のための処理と非終端記号が構成された時の処理の２つの動きから構成されている．

を並列実行させたからといって飛躍的な台数効果が得られるとは限らない. それは, 並列メタインタプリタでは, 実行終了条件 (総てのゴールキューが空である状態) やデッドロック条件 (総てのゴールがサスペンドしている状態) を 1 リダクション毎に同期をとって判定する必要があるからである. 図 4. にその処理イメージを示す.

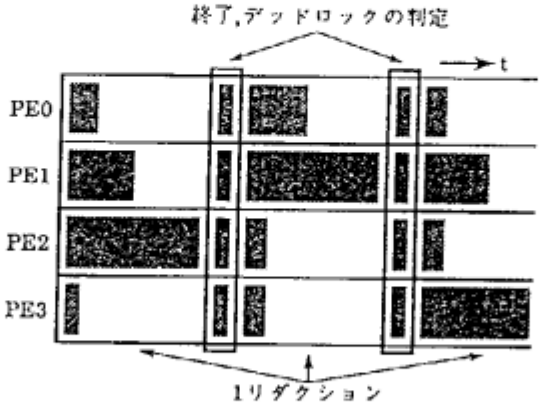


図 4. 情報収集のための同期処理

4.2 パフォーマンスモニタの拡張

パフォーマンスモニタによる負荷状況の推移を用いて, 詳細なプラグマ設定支援を行うためには, トレース情報を参照できる機能が必要である.

この, トレース情報の参照機能は, パフォーマンスモニタで選択したある時点におけるゴールキューの内容, あるいは, その時点の負荷を発生させるきっかけとなったリダクションを示すことを目的とする. そこで, 負荷情報とメタインタプリタから出力させたトレース情報を対応させることにより, プラグマ設定支援を行えるような機能の拡張を考えている.

5 まとめ

KL1 プログラムの設計支援システムにおいて, プラグマ設定支援を目的として開発した, メタインタプリタを利用したパフォーマンスモニタを紹介した. 現時点におけるメタインタプリタの実行速度は, 実用的とは言い難いものであり, この, 実行速度の向上を目指した並列メタインタプリタを現在開発中である. 今後は, さらにトレース情報と負荷情報のリンクおよび, プロセッサ間の通信量を示す機能等の開発を目指す.

参考文献

[Tanaka 88] 田中, 的場 : 変数管理をする *GHC* の自己記述, *ICOT Technical Report:TR-374, 1988*

[Tanaka 87] 田中, 太田, 的場, 神田 : *GHC* による仮想ハードウェアの構築とリフレクト機能について, 日本ソフトウェア科学会第 *4* 回大会論文集, *B-5-3, 1987*

[Sato 84] 佐藤義雄 : 実習グラフィックス, アスキー出版局, *1984*

いる．

⑤ 2 階の型

s : {X | X!subj=NP, X=VP }

この表記は，s という表現の型が，集合の抽象化の表記で表された型の集合（型）すなわち2 階の型をもつ型である．この型を経由して制約解消系と型推論系が融合する．

（2） 推論規則

基本的なものに対する型割当の集合を型宣言と呼び，Γで表記する．Γは辞書や文法規則や意味モデルにあたる

① トートロジー

$$e:t \in \Gamma \quad \Rightarrow \quad \Gamma \vdash e:t$$

この推論規則は，辞書のアクセスや文法規則のアクセスなどに対応する．

② 関数適用

$$\frac{\Gamma \vdash e:t \qquad \Gamma \vdash f:t \rightarrow s}{\Gamma \vdash ef:s}$$

この推論規則は，句構造規則がe という表現によって充足されることなどに対応する．

③ 和型削除

$$\frac{\Gamma \vdash e: [t_1,\ldots,t_n]}{\begin{array}{l}\Gamma \vdash e:t_1 \\ \cdots \\ \Gamma \vdash e:t_n\end{array}}$$

この推論規則は，型に曖昧性があるときに探索空間を分岐させることに対応している．

④ 対称関数適用

$$\frac{\Gamma \vdash e:t_j \qquad \Gamma \vdash f: [t_1,\ldots t_{j-1}, t_j, t_{j+1},\ldots t_n] \rightarrow t}{\Gamma \vdash ef: [t_1,\ldots t_{j-1}, t_{j+1},\ldots t_n] \rightarrow t}$$

この推論規則は，引数をとる順序が自由な関数の適用を表している．

⑤ 対象化

$$\frac{\Gamma \vdash e : t \qquad \Gamma \vdash t : s}{\Gamma \vdash e : s}$$

この推論規則は，型がものとしても型宣言されているならば，型をものとして扱ってよいということを主張するものであるが，このような推論規則が型推論としてはあまり適切ではない．

現在の時点では，対象化が可能な型は原子型に限っている．

この推論規則の存在によって，ものと型との境界は不明確になる．

ジェクトの階層が重なることもある．仮想機械オペレーティングシステムでは，オペレーティングシステムの中にもメタ / オブジェクトの階層があることになる．

メタプログラムの機能を実現するにはインタプリタを用いるのが簡明である．しかし，インタプリタを用いると実行効率の面で通常一桁程度以上の損失が生じる．部分評価技術などを用いてこの損失を軽減はできるが，現在の技術水準では数倍程度の損失はまぬがれない．ことに，メタ / オブジェクトの階層が重なると，この損失は階層一段ごとに乗ぜられ，最終的な効率は指数関数的に低くなっていく．

この効率上の問題は KL1 に翻訳されるような言語階層を積み重ねることでは解決できず，メタレベルプログラムとオブジェクトレベルプログラムを同じ土俵 (機械語) で動かすことが必要である．そのためには，メタプログラミング機能を言語自身に導入することが不可欠である．以下に，そのようなメタプログラムの持つべき機能を上げる．

失敗の伝播の制限　FGHC では，すべてのゴールは論理積関係にある．このため，ひとつのゴールが失敗すると，全体が失敗することになる．このような言語でメタレベルプログラムを記述すると，オブジェクトレベルプログラムに何らかの問題があって実行に失敗すると，メタレベルプログラムを含めたシステム全体が失敗することになってしまう．そこで，失敗の伝播範囲を制限する機構が必要になる．

メタ制御機能　メタレベルプログラムには，たとえば，暴走したオブジェクトレベルプログラムを強制的に停止させるなど，オブジェクトレベルプログラムの実行の制御を行えるメタ制御機能が必要である．

監視機能　メタレベルプログラムはオブジェクトレベルプログラムの実行の様子を何らかの意味で監視できなくてはならない．

例外処理機能　メタレベルプログラムはオブジェクトレベルプログラムの実行中に生じた例外事象を検出し，例外事象に応じた適当な処理を行えなくてはならない．

1.1.2　荘園機能

概要　メタプログラミング機能を導入するためには，すべてが平板に論理積となっている FGHC に，なんらかの構造を持ち込む必要がある．これを実現するために導入したのが荘園の機能である．

荘園は以下のようなプリミティブを用いて生成する．[1]

execute(Goal, Control, Result, Tag)

ここで，各引数は以下のようなものである．

Goal: 荘園の中で実行すべきゴール．

Control: 荘園内の実行を制御するためのコマンドを荘園の外部から送るストリーム．

Result: 荘園内の実行の結果によるさまざまな情報を，荘園外部に伝達するためのストリーム．

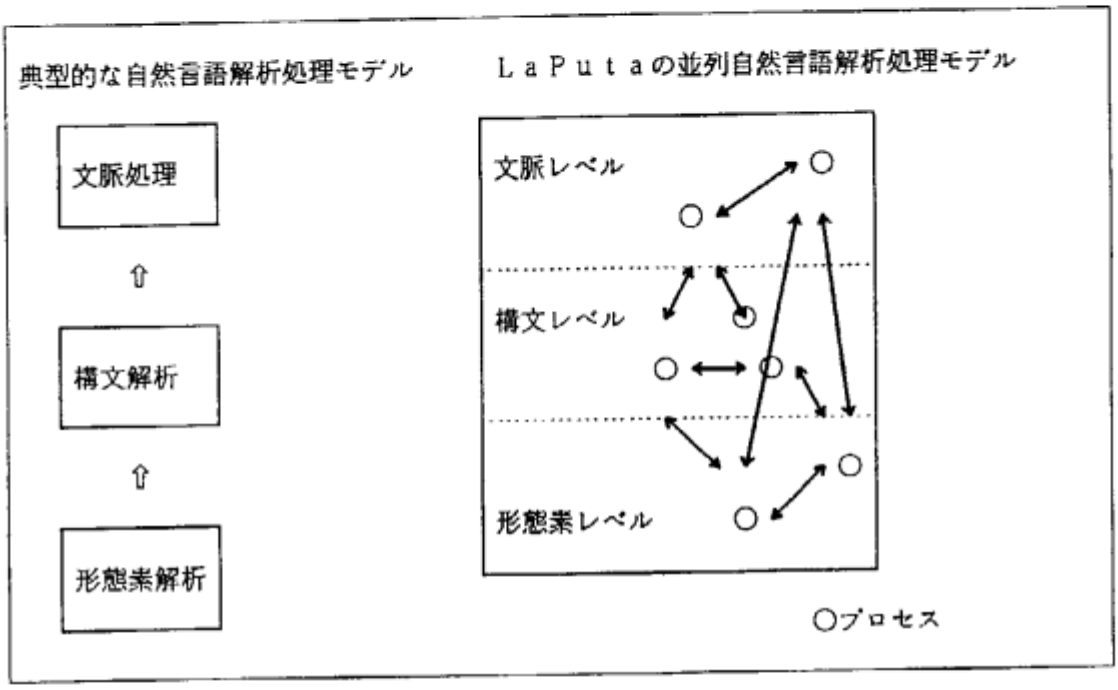
Tag: 荘園がネストしている場合，それらを識別するためのタグ．

各々の引数の詳細については後述する．

荘園の実行　上述の方法で生成した荘園内のゴール群は，荘園外とは独立した論理積を成す．すなわち，荘園内での失敗は荘園内に閉じたものであり，荘園外のゴールを巻き添えにすることはない．また，荘園内のゴールの実行の終了は実行結果のストリームに通知される (後述)．概念上，荘園はインタプリタを機械語レベルで実現したものであると考えて良い．

[1] 実際の仕様はもう少し複雑だが，ここでは詳細を捨象して述べる．

レベルの違いはここでは単にデータの違いであって処理方法の違いではない．そして様々なレベルのデータが同時並行的に処理されることによって並列協調処理が実現される．また，開発の観点から見ると，各レベルの文法はそのレベルの視点で開発することが可能であり，異なるレベルの情報を利用することも可能になる．また，言語現象ごとに個別に開発することも可能である．そして，このようにして開発された文法は，最終的にはマージされて並列的に実行されることになる．



処理モデルの比較

2．基本原理

このような処理モデルを考えると，別の問題が生ずる．処理過程で他の処理プロセスと動的に協調させようとすると，プロセス間の呼出関係が組み合わせ的に増加し，プロセスの構造が爆発的に複雑化するために開発が非常に困難になるという問題である．これを解決するために，我々は形態素解析や構文解析といった色付けの無い汎用的な処理機構を中心とした自然言語解析システムを構築することを考えた．

そして，この汎用的な処理機構の原理として，型推論［Reynolds 85,Hindley 86］と部分項の単一化に基づく制約解消［向井 89 ］を採用することにした．

型推論は与えられた表現に型宣言と呼ばれる環境の下で型を与えるための推論である．表現を単語の列とし，型を品詞もしくは句や文と考え，型宣言を辞書および句構造文法と考えると，入力として与えられた単語の列に対して，辞書に基づいて品詞を割当てたり，句構造文法に基づいて句や文としてまとめあげる構文解析は型推論に対応している．また，入力を文字列とし，形態

NewGoal: 例外を起こしたゴールの代わりに実行すべきゴールを指定するための変数.

例外の原因別に適当なタグを割当てる. 例外の報告は, 例外を起こした荘園, その親荘園, そのまた親, という先祖荘園の中で, 例外原因タグにマッチするタグを持つ最も近い先祖荘園の報告ストリームに流す. 荘園生成時に指定するタグはビットマスクで, 例外原因タグとビットごとの論理和を取って, ゼロでない荘園に報告する. こうすることによって, 特定の例外だけを取り扱い, 他はより外側の荘園に任せるような記述が可能になる.

例外の原因 例外の原因としては, ボディ部の組込み述語に対して誤った型の引数を与えた場合, 処理できない値の引数を与えた場合 (演算あふれやゼロ除算など), そしてゴールが失敗した場合などがある. KL1 ではユニフィケーションの失敗や, ガード部の成功する候補節がないことによる失敗は, 例外として扱う. したがって, 荘園の実行が失敗によって終了することはない. 失敗した際に荘園の実行を放棄したければ, 制御ストリームから放棄メッセージを送れば良い.

ガード部では例外が生じることはない. ガード部の組込み述語の型が述語の引数型にあっていない場合 (たとえば加算の引数にアトムを与えたような場合) は, 例外とせず単に失敗とする. これは, 例外による中断 / 再開の機構を簡素化するためである.

積極的な例外の生起 ユーザが積極的に例外を起こすことができるように, 以下のような組込み述語を用意する.

raise(Info, Data, Tag)

ここで, 各引数は以下のようなものである.

Info: 例外に関する情報. この引数が完全に (構造体の場合はその要素も, 要素が構造体の場合はそのまた要素と, 構造体全体が) 具体化されるまで例外の生起は遅延される.

Data: 任意のデータ. この引数は具体化されていてもいなくてもよい.

Tag: 例外のタグを指定する.

例外を取り扱うプログラムは例外を出すプログラムのメタプログラムである. このメタプログラムがデッドロックすることを防ぐためには, 具体化されていることが保証された例外情報が必要である. 組込み述語 raise の引数 Info はこのような目的に用いるためのものである. 一方, 引数 Data は, メタプログラムが中身を読まなくても良いデータ (たとえば, 後述の実行継続に使うデータ) を渡すのに用いる.

例外後の実行の継続 荘園内では例外を起こしたゴールの実行は中断され, NewGoal の具体化を待って, 元のゴールの代わりに NewGoal を実行する. この際, 荘園には NewGoal が具体化されたらそれを実行しようとするゴールがあると考えて良い. このゴールがあるため, NewGoal を具体化しないうちに荘園全体が先に成功裏に終了してしまうことはない.

例外処理を行う側で NewGoal を適当に与えることによって, 言語仕様を拡張することができる. たとえばあるパターンのユニフィケーションの失敗に対する処理を記述することによって, ボディ部でのユニフィケーションの仕様を拡張することもできる. このように積極的に例外を利用する場合, 荘園の実行が自動的に中断されてしまうような仕様は効率面から好ましくない. そこで, 同じ荘園内でも, 例外を起こしたゴール以外のゴールは正常に実行が継続されるものとする.

例外機構の意味 KL1 の例外機構は, すべてのゴールが暗黙のうちに報告ストリームにつながるようなストリームを持っていると考えることで, GHC の範囲内で説明できる. つまり

```
p(X,Y) :- true | q(X,Z), r(Z, Y).
```

のような節ひとつからなる述語定義は

謝辞

後半の例として用いた構文解析のプログラミングの色々な可能性については、ICOT の淵和男氏、三菱の佐藤裕幸氏、富士通の山崎重一郎氏との議論に負うところが大きかった。感謝いたします。また、日頃貴重な意見を賜わる ICOT 並列プログラムワーキンググループの諸氏に感謝したします。

参考文献

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

[Clark & Gregory 86] Clark, K.L. and Gregory, S., "PARLOG: Parallel Programming in Logic," ACM Trans. Programming Languages and Systems, Vol.8, No.1, pp.1-49, 1986.

[Ueda 86] Ueda, K., "Making Exhaustive Search Programs Deterministic," Proc. 3rd ICLP, '*Lecture Notes in Computer Science 225*', pp.270-282, 1986.

[Tamaki 87] Tamaki, H., "Stream-based Compilation of Ground I/O Prolog into Committed-Choice Languages," Proc. 4th ICLP, The MIT Press, pp.376-393, 1987.

[Matumoto 86] Matsumoto, Y., "A Parallel Parsing System for Natural Language Analysis," Proc. 3rd ICLP, '*Lecture Notes in Computer Science 225*', pp.396-409, 1986.

[Okumura & Matsumoto 87] Okumura, A. and Matsumoto, Y., "Parallel Programming with Layered Streams," Proc. 1987 International Symposium on Logic Programming, pp.224-232, San Francisco, September 1987.

[奥村 & 松本 87] 奥村晃、松本裕治、「レイヤードストリームを用いた並列構文解析」、情報処理学会第 35 回 (昭和 62 年後期) 全国大会予稿集、pp.861-862, 1987.

[松本 & 奥村 88] 松本裕治、奥村晃、「Dynamic Layered Stream Programming」、日本ソフトウェア科学会第 5 回大会論文集、pp.377-380, 1988.

[佐藤 90] 佐藤裕幸、「並列自然言語構文解析システム PAX の改良」、KL1 プログラミング・ワークショップ、ICOT, May 1990.

[山崎 90] 山崎重一郎、「並列自然言語解析システム LaPuta について」、KL1 プログラミング・ワークショップ、ICOT, May 1990.

1.2.3 指定方式

優先度の指定は，荘園単位におおまかに指定する方法と，各ゴールごとに細かく指定する方法の両者を用意した．メタレベルプログラム的な意味でおおまかに制御する場合（たとえばオペレーティングシステムがユーザプログラムの優先度を管理する場合）には荘園単位の指定，オブジェクトレベルの知識を元に細かい制御をしたい場合（たとえばアルファベータ探索プログラムで，深さ優先探索をしたい場合）にはゴール単位の細かい制御ができるようにするためである．

ゴール単位の指定はそのゴールそのものの実行の優先度を指定する．荘園に対する指定は，荘園内のゴールの優先度の最大 / 最小値を指定することによって行う．荘園生成のための述語にはこの優先度範囲指定のための引数を追加する．

指定は荘園単位の指定・ゴール単位の指定のいずれも，優先度の絶対値ではなく，実行中の荘園の優先度範囲や指定を行う計算自身の優先度と比べての相対的な値として与える．具体的には以下のいずれかの方法を用いる．

1. 指定を行う計算の属している荘園内での相対値．属している荘園の優先度範囲内のどのあたりで実行するかを，割合で指定する．

2. 指定を行う計算自身の優先度と，属している荘園の最大（または最小）優先度との間のどのあたりで実行するかを，割合で指定する．

絶対値ではなく相対値を用いることには，以下のような利点がある．

- 優先度の絶対値の範囲は処理系ごとに異なることが考えられるが，相対記述ならプログラムには処理系独立な優先度指定記述ができる．

- 局所的に相対優先度指定を付したプログラムを，大局的に荘園全体の優先度範囲を変えて呼び出すことができる．

1.2.4 暴走の停止

優先度の概念がなかったり，オブジェクトレベルの優先度がメタレベルの優先度以上になったりすると，オブジェクトレベルプログラムの暴走をメタプログラムから制止できなくなる．たとえオブジェクトレベルの実行を停止するためのメタプログラムのゴールを実行しようとしても，オブジェクトレベルの暴走のためにメタレベルはたったひとつのリダクションも実行されないことがあり得る．

この問題を解決する方法のひとつに，スケジューリングに公平さを導入する方法がある．公平なスケジューリングでは，実行可能状態にあるどのプログラム部分も有限時間内には少なくとも少しは実行されることを保証する．このためには，幅優先や制限つき深さ優先のスケジューリングを行なえばよい．

一方，優先度の機構を用いてもこの問題は解決できる．オブジェクトレベルが決してメタレベルよりも高い優先度を持たないように，荘園を生成する際の優先度範囲指定を行えばよい．逆に，優先度機構があれば，プログラムの暴走防止の目的で公平なスケジューリングを行う必要はない．

スケジューリングが不公平でよければ，プログラムの局所性を最大限に生かすような，あるいは実装が最も容易なスケジューリング方式を自由に選択できる．極端な例では，公平なスケジューリング方式で最も簡単である幅優先スケジューリングより，公平さを保証しない最も簡単なスケジューリング方式である深さ優先スケジューリングの方が，通常時間的な局所性をはるかに高くでき，ワーキングセットを小さくできる．また，実装も簡単である．

暴走の停止を保証するためには処理系がある程度は優先度を守る保証が必要である．それは，他に低優先度の計算がいくらあろうとも，有限時間内に終了できるような最高優先度の計算は有限時間内に終了する，ということである．

1.3 クローズ間の優先度

GHC では，複数の候補節のどれを用いてもリデュース可能な場合，どの節を用いるかは，言語仕様としては定めないことになっている．これは言語仕様を簡潔に保つことが本来の動機である．処理系の作成

```
queens(Ans) :- true |
   q(LS), qs(LS,[],Out-[]).
q(LS) :- true |
   qX(begin,Q1), qX(Q1,Q2),
   qX(Q2,Q3), qX(Q3,LS).
qX(In,Out) :- true |
   Out = [1*In,2*In,3*In,4*In].
qs(begin,Board,From-To) :- true |
   From=[Board|To].
qs([I*LSi|LSs],Board,From-To) :- true |
   checkQs(Board,I,1,LSi,Board,From-M),
   qs(LSs,Board,M-To).
qs([],_,From-To) :- true | From=To.
checkQs([Q|Qs],Q,D,In,Board,Out) :-
   true | Out=End-End.
checkQs([Q|Qs],I,D,In,Board,Out) :-
   Q=:=I+D | Out=End-End.
checkQs([Q|Qs],I,D,In,Board,Out) :-
   Q=:=I-D | Out=End-End.
checkQs([Q|Qs],I,D,In,Board,Out) :-
   Q=\=I+D, Q=\=I-D, Q\=I |
   D1 := D+1,
   checkQs(Qs,I,D1,In,Board,Out).
checkQs([],I,D,In,Board,Out) :-
   true | qs(In,[I|Board],Out).
```

このプログラムでは'qX'が一つのクイーンに相当し、これによって4クイーン問題の解になり得る組合せがレイヤードストリームの形で構成される。'qs'が部分解に対応するプロセスであり、これがレイヤードストリームを消費しながら新しい部分解へと分岐していく。

後者のプログラムは、クイーンのあらゆる可能な位置をしらみつぶしに調べるのに対し、前者では、レイヤードストリームの構成と共に配置されたフィルターが同時に枝狩りを開始し、しかも密にデータの共有を行なっている。実験による比較では、4クイーンの場合の総リダクション数(頭部単一化の数)が前者と後者とでは、160対1615、6クイーンでは、1,382対236,185であった。クイーンの数が増すとこの差は急激に拡がる。これは、レイヤードストリームを効果的に使った場合の解の共有の効果を顕著に示す例である。

次に文脈自由文法の構文解析の問題を用いて、もう一つのタイプの再帰的記述による問題のプログラミングを説明する。この問題の場合は、記述の右辺も左辺も部分問題しか含まないので、部分問題の解(すなわち、非終端記号)を並列プロセスと考える。右辺の部分問題同志は情報のやりとりを行なう必要があるので、各プロセスは隣のプロセスへ自分の存在を示すデータ(非終端記号名)を送る。相互にデータを送り合う必要はないので、必ず左のプロセスが右のプロセスへデータを送ると仮定しよう。具体的には、各プロセスは左のプロセスから受け取ったデータ(レイヤードストリーム)の先頭に自分の持つ情報を付け加えた新しいレイヤードストリームを右のプロセスへ送れば良い。このようにして、構文解析の並列プログラムを実現することができる。初期プロセスは、解析しようとしている入力を構成する品詞の列に対応するプロセス列である。

右辺の部分解をどのようにして結合するかということに幾つかの選択肢がある。構文解析システム PAX[Matsumoto 86]は、レイヤードストリームに基づくシステムであるが、このシステムでは、例えば、a ← b c d という文法規則に対して、下の(1)に対応するプログラムを生成する。例えばcについて見ると、cは左隣にbが存在することを通信によって知ると、非終端記号bとcが連続して現れたという情報をデータ'b_c'で表現し、それを右隣へ送っている。一方、別の考え方も可能で、cは自分の名前を右へ送っているのであるから、それを受け取ったプロセスdが、cとdが連続して現れたことを示すプロセス'c_d'を生成してもよい。このような考え方が[佐藤 90][山崎 90]によって提案されている。つまり、前者ではプロセスの並びをデータで表現しているのに対し、後者ではそれをもプロセスで表現しているのである。後者の本質的な部分のプログラムを(2)に示す。

この例のように，逐次処理の場合は逐次制御と条件判断で実現できた方式が，並列処理まで拡張して考えるとメタレベルの制御 (優先度の指定と実行の放棄) を用いなければ最適にならない場合が少なくない．このことも KL1 にメタレベルの実行管理機能を導入する大きな動機になった．

2 資源管理機能

2.1 必要性

メタレベルプログラムがオブジェクトレベルプログラムを管理するには，オブジェクトレベルプログラムがどのようなふるまいをしているかを，少なくともマクロには把握していなければならない．前節で述べた実行管理機能を用いれば，オブジェクトプログラムの実行の終了，異常事態の生起といった，計算の進行状況の質的な把握は可能である．しかし，それだけでは量的な側面，たとえばどのぐらい計算が進行しているのか，といったことはわからない．

一般的なメタレベルの機構として導入するには，解いている問題に依存しない計算進行状況の観測機構が適当である．どれぐらいの時間計算しているのか，どのぐらいのメモリを費やしているのか，といった計算機構そのものに関わる測度を用いれば，同じ計算機構を用いているかぎり共通のものであり，メタレベルの計算戦略を立てるためのデータとして有効である．

一方，オペレーティングシステムのようなプログラムを考えると，たとえばユーザプログラムがメモリをすべて消費してしまうと，オペレーティングシステムが動くためのメモリが残っていない，ということも生じ得る．これを防ぐには，メタレベルにはオブジェクトレベルでの消費計算資源を制限する能力が必要である．

そこで，こうした情報を大きなオーバヘッドなく得るために，言語のメタレベル機能の一環として資源管理機能を導入することとした．

2.2 方針

2.2.1 管理単位

資源消費の管理単位は，実行管理と同じく荘園とした．これは資源管理の結果として実行を制御したい場合が多いので資源管理単位を実行管理単位と一致させれば制御が容易であること，処理系実現上も資源消費単位を別に設けるとオーバヘッドを増すこと，の二点が理由である．

資源消費の報告の仕方として，単純にはオブジェクトレベルからメタレベルに資源の消費状況を逐一報告することが考えられる．しかし，この方法ではメタレベルが常にオブジェクトレベルを監視していなければならず，そのオーバヘッドが大変大きくなってしまう．そこで，あらかじめ適当な資源消費許容量をメタレベルから設定し，それが尽きそうになるまではオブジェクトレベルはメタレベルに特に報告することなく動作できるものとした．許容量が尽きそうになったことを知らされたメタレベルでは，

- 許容量を追加して計算を続行させる．
- 許容量の追加はせず，その計算は打ち切る．
- 許容量の追加はせず，いずれその計算は資源枯渇によって中断するにまかせ，当面他の方法をためすが，将来継続することもできるようにしておく．

などの選択が可能である．一回に与える許容量の単位を小さくすれば精度の良い資源管理ができるが，それだけオーバヘッドも増えることになる．

2.2.2 管理対象

管理の対象としては以下のようなものが考えられる．

計算時間: 当該荘園の計算のためにどのぐらい時間をかけてよいか．

メモリ消費量: 当該荘園の計算のためにどのぐらいメモリを消費してよいか．

例 2　グラフの最短経路問題

グラフ内のある決められた始節点と終節点を結ぶ最小コストの経路を発見する問題。

$$f_k(v) = \min_u \left(f_{k-1}(u) + d(u,v) \mid (u,v) \in E\right)$$

$f_k(v)$: 始節点から k 本以内の枝を通って節点 v へ至る最短経路長

$d(u,v)$: u と v を結ぶ枝 (u,v) の長さ

整合性: $f_{k-1}(u)$ と $d(u,v)$ において u が同一節点でなければならない

多様性: すべての可能な節点 u を対象とする

目標関数: $f_k(v)$ の最小値

例 3　N 個の行列の積のコストの最小値問題

与えられた N 個の行列の積を最小回数の乗算によって求めるための計算順序を決定する問題。

$$matrix_{ij} = \min_k \left(matrix_{ik} \oplus matrix_{kj}\right)$$

$matrix_{ij}$: 第 i 行列から第 j 行列までのすべての行列の積を計算するためのコスト

整合性: 右辺の行列が隣接していなければならない

多様性: すべての可能な k $(i \leq k \leq j)$ を対象とする

目標関数: $matrix_{ij}$ の最小値

例 4　文脈自由文法の構文解析

与えられた文脈自由文法に基づいて、N 個の単語よりなる入力が文と認められるかとうかを判定する問題。

$$NT_{ij} = \underset{k_1,k_2,\cdots k_n}{\text{exist}} \left(NT_{ik_1} \oplus NT_{k_1k_2} \oplus \cdots \oplus NT_{k_nj}\right)$$

NT_{ij}: 入力の第 i 単語から第 j 単語によって構成される非終端記号

整合性: $NT_{ij} \leftarrow NT_{ik_1}\ NT_{k_1k_2} \cdots NT_{k_nj}$ という文法規則が存在しなければならない

多様性: すべての可能な文法規則を対象とする

目標関数: NT_{ij} が少なくとも一つ存在する

3.2　条件を実現するためのメカニズム

次の節で示すように、再帰的に記述された問題の解や解の集合を表現するために前節で導入したレイヤードストリームを利用する。本節では、問題の再帰的記述において定義された種々の条件をプログラム上で実現するための基本的なメカニズムを説明する。

レイヤードストリームプログラミングでは、部分問題の解や解を構成する要素がレイヤードストリームを用いて表現され、規模の小さな部分問題の解や解の構成要素からより大きな問題の解を構成するようにプログラムされる。その中で、多様性条件は、次節で説明するように、利用可能な部分解や解要素に対応する情報をレイヤードストリームとして生成する操作として実現される。

一方、整合性条件と目標関数は、レイヤードストリームの中から問題の条件に合致しない部分を取り除くフィルタープロセスとして実現される。整合性条件に対応するフィルターは consistency filter と呼ばれる。また、目標関数を実現するフィルターを preference filter と呼ぶ。これらについては次節で例を用いて説明する。

3 計算量オーダの保存

従来のピュアな (純関数型や純論理型の) 言語では, いわゆる副作用を排除したため, 手続き言語と同じ計算量オーダになるアルゴリズムを用いることができなくなっているものが多かった.

ピュアな言語にすると並列処理が容易になるので, アルゴリズムの持つ理論上の並列度に匹敵するほどのハードウェア並列度を持つ超高並列処理を行なう場合には, 並列処理しやすいという利点が大きく効き, このような欠点は問題にならない場合もある. しかし, アルゴリズム上の並列度が利用可能なハードウェアの並列度よりもかなり大きくなると, 計算量のオーダの違いは致命的になる.

そこで, KL1 では従来の手続き言語と同じ計算量オーダを実現できるベクタの「更新」機構とストリームのマージ機構を用意した. これらの機構を利用すれば, ロック機構を持つような共有メモリのランダムアクセス機構を, 一定の手間でエミュレートできる. つまり, 手続き言語上のどんなアルゴリズムでも, 計算量のオーダとしては同一になるような実現が可能になったわけである.

3.1 総計算量, 並列度と計算時間の関係

最適なアルゴリズムはハードウェアの持つ並列度と問題の大きさとの関係から決まる.

たとえば, 問題のサイズ n に対して $n^3 \log n$ の総計算量になるアルゴリズム A と, n^3 の総計算量のアルゴリズム B があったとする. 逐次処理の場合はもちろんアルゴリズム B が有利であるが, 並列処理の場合はアルゴリズムの並列度が問題になる. アルゴリズムの持つ並列度は, A は n^2 程度, B は n 程度と低かった場合, どちらのアルゴリズムが有利だろうか.[6]

問題のサイズ n を 2^5 程度, ハードウェアの並列度を 2^{10} 程度と仮定すると, 総計算量は A が $5 \cdot 2^{15}$, B だと 2^{15}, 並列度はそれぞれ 2^{10}, 2^5 程度となる. このため, 計算時間は A では $5 \cdot 2^5 = 160$ 程度, B では $2^{10} = 1024$ 程度となり, アルゴリズム A がはるかに有利である. これはアルゴリズム B ではハードウェアの並列度が十分に生かせないからである.

同じハードウェアで $n = 2^{10}$ 程度の問題を解くとどうなるだろう. 総計算量は A が $10 \cdot 2^{30}$, B が 2^{30} 程度になる. 並列度は 2^{20} と 2^{10} 程度になるが, ハードウェアの並列度が 2^{10} しかないので, 実際にはこれに抑えられてしまう. このため, 計算時間は A が $10 \cdot 2^{20}$, B が 2^{20} 程度となり, こんどは B の方がはるかに有利になる.

この傾向は問題のサイズが大きくなるほど顕著になる. これは, ハードウェアの規模が小さ過ぎて, アルゴリズム A の並列度が生かせなかったからである, という見方もあり得るだろう. しかし, 現実に利用できるハードウェアはいくらでも大規模にできるというものではない. また, 計算量理論の考え方は問題のサイズを大きくしていった極限を考えるものであるという立場からしても, アルゴリズム B の方が優れているというのが適当であろう.

このように, ハードウェアを固定して大きな問題を考えると, 総計算量のオーダがアルゴリズムの良否を決める支配的要因であることは, 逐次計算の場合とまったく変わりない.

3.2 ランダムアクセス可能な構造体

従来のピュアな言語では, ランダムアクセスできる記憶, すなわちコンスタントの手間で参照・更新ができる構造体がないのが普通だった. これでは, 手続き処理を前提に開発されたアルゴリズムの多くを, 同じ計算量では実現できない. この点を改善するために, KL1 にはベクタ型のデータとその要素の参照や「更新」を行なう組込み述語を導入した. 導入に当たってはセマンティクスをピュアに保った上で, 想定した使用法を守る限りは「更新」にあたる操作を一定の手間でできるような処理方式を用いている.

KL1 言語の表層上は要素の「更新」を行なうわけではない. 組込み述語

set_vector_element(OldV, Pos, OldE, NewE, NewV)

は「ベクタ OldV の Pos 番目の要素は OldE で, それを NewE に置き換えたようなベクタが NewV である」という意味を持つ. GHC であるから双方向性は失われており, OldV, Pos については実行前に具体化を待ち合わせる. NewV は意味上ここで新しく作られるものであって, OldV とは別のものである.

[6] ここでは通信遅延が無視できるようなアルゴリズムを想定している.

分解の共有に役立つ。また、相異なる部分解に関する情報をストリームに流すことによってOR並列性を達成し、一つの解の異なる部分を別のプロセスに処理させてそれらをストリームの動的な結合によってまとめることによりAND並列性を達成する。特に、動的計画法によって解かれる問題のように、部分解を蓄積して他の部分解を計算する時にそれらを再利用することによって効率を達成する解法は、その特徴を生かすことが可能であり、部分解の共有を自然に実現することができる。

2 レイヤードストリームとその特徴

本論では、CCLについての基本的な知識を仮定し、言語としてGHCを用いることにする。Prologをご存知の読者は、Prologの節の右辺に含まれる述語の実行が並列に行なうことができる言語であると理解していただいてかまわない。ただし、GHCでは、述語の呼び出し時に、呼び出し側の述語に含まれる変数に値の割当が起こる節についてはその節の実行を一時的に中断する。そのような節は、他の述語の実行によって呼び出し側の変数に充分な値が得られて後、実行を再開する。

CCLでは、述語が並列に動くプロセスとして扱われ、通信チャネルとしてストリーム(構造上は、リスト表現)が用いられる。我々が提案するレイヤードストリームは、構文的には単に自分自身と相似な構造を内部に含むデータからなるストリームであるが、次のような特徴を持つCCLの並列プログラムの実現に役立つと考えている。

1. 問題の再帰的な記述からの並列プログラムの導出
2. 並列プログラム中における部分問題の解の共有
3. 問題のAND並列性およびOR並列性両者の抽出
4. ストリーム(すなわち、通信チャネル)の動的な結合

レイヤードストリームは、一般に次のような構造をしている。

$$[a_1*LS_1,\ a_2*LS_2,\ \cdots\ ]$$

ここに、'a_i'は項、'*'は結合記号、'LS_i'は内部に含まれるレイヤードストリームである。レイヤードストリームは、一般的には、問題の部分解の集合を表すと考えられる。項は、問題の部分解の一部を成すデータであり、問題によって異なる。結合記号は、そのようなデータとより規模の小さい部分解の集合を表すレイヤードストリームを結合するための記号であり、ここでは'*'を用いることにする。上の例の先頭のデータ、

$$a_1*LS_1 = a_1*[b_1*L_1,\ b_2*L_2,\ \cdots\ ] \quad \text{は、}$$

$$[a_1*b_1*L_1,\ a_1*b_2*L_2,\ \cdots\ ]$$

という集合を表している。これを再帰的に適用すると、元のレイヤードストリームは、それを結合記号'*'についてすべて展開したデータの集合を表現していると考えることができる。

このような部分解の再帰的な表現によって、後に述べる探索問題の再帰的な記述との対応が明確になる。しかも、部分解を構成する項の部分の生成および処理と、内部のレイヤードストリームの生成および処理を独立に、すなわち、並列に行なうことを可能にする。レイヤードストリームの構造の縦方向の並列処理がAND並列に、横方向の並列処理がOR並列に対応する。また、レイヤードストリームは部分解の共通な部分を共有したデータ構造であり、個別の部分解に対する処理を重複して行なうことを避けることができるので、逐次的に実行しても効率のよりアルゴリズムを得ることが可能になる。

また、ストリームのデータ内部に含まれているストリームが新しく生まれたプロセスに渡されることによってストリームすなわち通信チャネルの動的な結合が行なうことができ、動的なプロセス間通信が実現できる。

4.1 ガード内の実行順序とセマンティクス

Flat GHC が本来の GHC と異なる点は，ガード部にユーザ定義述語の呼び出しを含まないこと，である．このことにより，GHC では必要だった AND-OR 実行木の管理や，ネストした変数レベルの管理が不要になり，処理系作成の容易化，実行効率の改善に利するところが大きい．

本来の GHC ではガード部に複数の呼び出しがあれば，それらは並列に実行する．このこと自体は FGHC になっても変わらないが，組込み述語だけが対象となるので，並列に実行したと同じセマンティクスを逐次実行で得ることは可能である．問題が生じるのはガード内の述語呼び出し間にループするようなデータ依存関係[10]がある時だけだが，複数の出力引数が個別に決まるような組込み述語はないように設計しているからである．

KL1 では FGHC よりさらに仕様を縮小し，ガード部は逐次に，左から右の順で実行するものとした．こうしてもガード部をデータ依存関係に従った順に記述したプログラムならば，そのガード部全体の成功の検出については並列実行の場合と同じセマンティクスを実現できる．ただし，失敗の検出についてはこの限りではない．すなわち，並列に，あるいは異なる順序で実行した場合には検出できる失敗を見つける前に，他の検査が具体化不足のために中断し，節の試行全体が中断する場合がある．

これが問題になり得るのは，実行の失敗になんらかの意味を持たせた場合で，具体的には荘園機構を用いて実行の失敗を計算に利用する場合と，otherwise を用いて節の失敗を他の節の成功の条件とした場合である．このような場合についても，目的とする動きをするプログラムを別の方法で書くことは，効率を問題にしなければ常に可能[11]であり，多くの場合は効率を損なわない簡単な書換え方法がある．そこで，次節に述べるような効率上の問題を重視し，KL1 では逐次実行のセマンティクスをとることとした．

4.2 効率上の利点

ガード部の実行を逐次的であるとしたことによって，多くの場合に無駄な計算を防げる．たとえば

```
p([], _) :- true | true.                          % 1
p([up|S], N) :- true | N1 := N+1, p(S, N1).       % 2
p([down|S], N) :- true | N1 := N-1, p(S, N1).     % 3
p([even(X)|S], N) :- N mod 2 = 0 | X = yes, p(S, N).  % 4
p([odd(X)|S], N) :- N mod 2 = 1 | X = no, p(S, N).    % 5
```

というような述語があったとする．これを

```
?- p(S, 0), q(S).
```

のように呼び出したとする．ここで q はストリーム S の値を決めるような述語である．

仮に p の呼び出しを先に試行したとすると，まだ S の値は決まっていないので，実行は中断する．この際，逐次実行ならば第一引数を調べるだけであるが，失敗の検出をしようとすると第二引数も調べなければならない．整数には奇数が偶数かのいずれかしかない，などという組込み述語の対象ドメインについての知識を一般に言語処理系に求めるのは無理だから，N を実際に調べずに節 4, 5 の両者とも失敗することはない，というわけにはいかない．[12]

この検査は，もし第一引数の値が決まって p の実行を再開した後に，節 1, 2, 3 のいずれかが選ばれた場合は不要なものである．節 4, 5 が選ばれたとしても

- もう一度第二引数を調べる
- 前回に第二引数を調べた結果を覚えておく

のいずれかが必要で，前者なら無駄な計算が，後者なら覚えておくための手間が余分に必要になる．逐次実行のセマンティクスにすればこうした余分な操作は避けられる．

[10] ガードに p(X, Y), q(X, Y) の両者があり，p が X を決めないと q が動けず，q が Y を決めないと p が終了できない，というような関係．

[11] いったんボディに入って，必要な検査の結果を値として返す述語を呼び，その結果をマージして調べれば良い．

[12] N が整数値でなかった場合も失敗することになっている．

5.2.1 現行の方式

Multi-PSI の処理系では，別プロセサ（ないしクラスタ）にある変数の値が必要であることがわかった場合，値の読み出しを要求しておいてそのゴールの実行は中断し，別のゴールの実行に移る．読み出し要求を受けた側では，その変数の値が決まっているかどうかを調べ，決まっていればその値を返し，まだ決まっていなければ決まった時点で値を返すような仕掛けをしておく．

この方式では，変数同士のユニフィケーションについては報告が返らないが，変数の同一性をガードでのユニフィケーションの成功条件とはしないセマンティクスならば問題ない．

5.2.2 変数同一性判定のための方式

変数が同一になった時点で成功とするためには，当該変数が他の変数と同じものになったときにも，どの変数と同じになったかを報告する，という方式が考えられる．この方法は遠くにあるメモリへのアクセスの回数を増やすことになる．また，変数が他の変数とユニファイされることによって，一つの変数について報告される値が刻々と変わっていくので，その全履歴を保持するなどしないと正しく同一性を判定することができない．

前述の方法の問題点は，ゴールリダクションを行なう側で変数の同一性を検査しようとしたことにある．代案として，値の読み出し要求を送るのではなく，二つの変数に適当な識別子をつけた同一性検査要求を送る，という方法が考えられる．この方式ならばひとつの変数に同じ識別子の検査要求がふたつ到達した時に報告すれば良い．しかし，この場合でも，識別子の一意性の管理が必要になること，変数同士のユニフィケーション手続きが複雑になること，などの問題がある．また，ひとつの変数を複数のゴールが待っている場合，処理の手間が待っているゴールの数の二乗に比例する程度になってしまう．[15]

5.3 オーバヘッドの限定の困難

ユニフィケーション対象が両方とも変数である場合，プログラム上も，実行途上の処理系にも，変数の同一性検査が必要となるかどうかはわからない．このため，変数の同一性判定を導入すると，実際に同一性を知りたい場合についてだけでなく，この機能を必要としない場合にも，この機能のためのオーバヘッドがかかってしまう．

こうしたことから，KL1 では変数の同一性判定は言語仕様に含めなかった．

おわりに

現在の KL1 の仕様は，その基礎となった Flat GHC に

- オペレーティングシステムを含む，複雑な大規模プログラムの記述のための機能追加
- 手続き言語のためのアルゴリズムを，同一のオーダの計算量で記述できるために必要な機能追加
- 効率的な実現の困難な機能の削除

を行なったものになっていることを述べた．

KL1 言語の現在の仕様は最終的なものではなく，今後の本格的な利用や実装の経験の積み重ねから，言語仕様や処理方式をさらに改善していきたい．改善方向としては

- 必要で効率的実現が可能な機能の追加
- 効率面で問題のない処理方式を発見した機能の復活

が考えられる．また

- 効率面でのネックになっており，ソフトウェア作成上不要な機能の削除

も考えていくつもりでいる．

[15] 現在はゴールの数に比例する程度．

5.2.1 現行の方式

Multi-PSI の処理系では，別プロセサ (ないしクラスタ) にある変数の値が必要であることがわかった場合，値の読み出しを要求しておいてそのゴールの実行は中断し，別のゴールの実行に移る．読み出し要求を受けた側では，その変数の値が決まっているかどうかを調べ，決まっていればその値を返し，まだ決まっていなければ決まった時点で値を返すような仕掛けをしておく．

この方式では，変数同士のユニフィケーションについては報告が返らないが，変数の同一性をガードでのユニフィケーションの成功条件とはしないセマンティクスならば問題ない．

5.2.2 変数同一性判定のための方式

変数が同一になった時点で成功とするためには，当該変数が他の変数と同じものになったときにも，どの変数と同じになったかを報告する，という方式が考えられる．この方法は遠くにあるメモリへのアクセスの回数を増やすことになる．また，変数が他の変数とユニファイされることによって，一つの変数について報告される値が刻々と変わっていくので，その全履歴を保持するなどしないと正しく同一性を判定することができない．

前述の方法の問題点は，ゴールリダクションを行なう側で変数の同一性を検査しようとしたことにある．代案として，値の読み出し要求を送るのではなく，二つの変数に適当な識別子をつけた同一性検査要求を送る，という方法が考えられる．この方式ならばひとつの変数に同じ識別子の検査要求がふたつ到達した時に報告すれば良い．しかし，この場合でも，識別子の一意性の管理が必要になること，変数同士のユニフィケーション手続きが複雑になること，などの問題がある．また，ひとつの変数を複数のゴールが待っている場合，処理の手間が待っているゴールの数の二乗に比例する程度になってしまう．[15]

5.3 オーバヘッドの限定の困難

ユニフィケーション対象が両方とも変数である場合，プログラム上も，実行途上の処理系にも，変数の同一性検査が必要となるかどうかはわからない．このため，変数の同一性判定を導入すると，実際に同一性を知りたい場合についてだけでなく，この機能を必要としない場合にも，この機能のためのオーバヘッドがかかってしまう．

こうしたことから，KL1 では変数の同一性判定は言語仕様に含めなかった．

おわりに

現在の KL1 の仕様は，その基礎となった Flat GHC に

- オペレーティングシステムを含む，複雑な大規模プログラムの記述のための機能追加
- 手続き言語のためのアルゴリズムを，同一のオーダの計算量で記述できるために必要な機能追加
- 効率的な実現の困難な機能の削除

を行なったものになっていることを述べた．

KL1 言語の現在の仕様は最終的なものではなく，今後の本格的な利用や実装の経験の積み重ねから，言語仕様や処理方式をさらに改善していきたい．改善方向としては

- 必要で効率的実現が可能な機能の追加
- 効率面で問題のない処理方式を発見した機能の復活

が考えられる．また

- 効率面でのネックになっており，ソフトウェア作成上不要な機能の削除

も考えていくつもりでいる．

[15] 現在はゴールの数に比例する程度．

4.1 ガード内の実行順序とセマンティクス

Flat GHC が本来の GHC と異なる点は，ガード部にユーザ定義述語の呼び出しを含まないこと，である．このことにより，GHC では必要だった AND-OR 実行木の管理や，ネストした変数レベルの管理が不要になり，処理系作成の容易化，実行効率の改善に利するところが大きい．

本来の GHC ではガード部に複数の呼び出しがあれば，それらは並列に実行する．このこと自体は FGHC になっても変わらないが，組込み述語だけが対象となるので，並列に実行したと同じセマンティクスを逐次実行で得ることは可能である．問題が生じるのはガード内の述語呼び出し間にループするようなデータ依存関係[10]がある時だけだが，複数の出力引数が個別に決まるような組込み述語はないように設計しているからである．

KL1 では FGHC よりさらに仕様を縮小し，ガード部は逐次に，左から右の順で実行するものとした．こうしてもガード部をデータ依存関係に従った順に記述したプログラムならば，そのガード部全体の成功の検出については並列実行の場合と同じセマンティクスを実現できる．ただし，失敗の検出についてはこの限りではない．すなわち，並列に，あるいは異なる順序で実行した場合には検出できる失敗を見つける前に，他の検査が具体化不足のために中断し，節の試行全体が中断する場合がある．

これが問題になり得るのは，実行の失敗になんらかの意味を持たせた場合で，具体的には荘園機構を用いて実行の失敗を計算に利用する場合と，otherwise を用いて節の失敗を他の節の成功の条件とした場合である．このような場合についても，目的とする動きをするプログラムを別の方法で書くことは，効率を問題にしなければ常に可能[11]であり，多くの場合は効率を損なわない簡単な書換え方法がある．そこで，次節に述べるような効率上の問題を重視し，KL1 では逐次実行のセマンティクスをとることとした．

4.2 効率上の利点

ガード部の実行を逐次的であるとしたことによって，多くの場合に無駄な計算を防げる．たとえば

```
p([], _) :- true | true.                          % 1
p([up|S], N) :- true | N1 := N+1, p(S, N1).       % 2
p([down|S], N) :- true | N1 := N-1, p(S, N1).     % 3
p([even(X)|S], N) :- N mod 2 = 0 | X = yes, p(S, N).  % 4
p([odd(X)|S], N) :- N mod 2 = 1 | X = no, p(S, N).    % 5
```

というような述語があったとする．これを

```
?- p(S, 0), q(S).
```

のように呼び出したとする．ここで q はストリーム S の値を決めるような述語である．

仮に p の呼び出しを先に試行したとすると，まだ S の値は決まっていないので，実行は中断する．この際，逐次実行ならば第一引数を調べるだけであるが，失敗の検出をしようとすると第二引数も調べなければならない．整数には奇数が偶数かのいずれかしかない，などという組込み述語の対象ドメインについての知識を一般に言語処理系に求めるのは無理だから，N を実際に調べずに節 4, 5 の両者とも失敗することはない，というわけにはいかない．[12]

この検査は，もし第一引数の値が決まって p の実行を再開した後に，節 1, 2, 3 のいずれかが選ばれた場合は不要なものである．節 4, 5 が選ばれたとしても

- もう一度第二引数を調べる
- 前回に第二引数を調べた結果を覚えておく

のいずれかが必要で，前者なら無駄な計算が，後者なら覚えておくための手間が余分に必要になる．逐次実行のセマンティクスにすればこうした余分な操作は避けられる．

---

[10] ガードに p(X, Y), q(X, Y) の両者があり，p が X を決めないと q が動けず，q が Y を決めないと p が終了できない，というような関係．

[11] いったんボディに入って，必要な検査の結果を値として返す述語を呼び，その結果をマージして調べれば良い．

[12] N が整数値でなかった場合も失敗することになっている．

分解の共有に役立つ。また、相異なる部分解に関する情報をストリームに流すことによってOR並列性を達成し、一つの解の異なる部分を別のプロセスに処理させてそれらをストリームの動的な結合によってまとめることによりAND並列性を達成する。特に、動的計画法によって解かれる問題のように、部分解を蓄積して他の部分解を計算する時にそれらを再利用することによって効率を達成する解法は、その特徴を生かすことが可能であり、部分解の共有を自然に実現することができる。

2 レイヤードストリームとその特徴

本論では、CCLについての基本的な知識を仮定し、言語としてGHCを用いることにする。Prologをご存知の読者は、Prologの節の右辺に含まれる述語の実行が並列に行なうことができる言語であると理解していただいてかまわない。ただし、GHCでは、述語の呼び出し時に、呼び出し側の述語に含まれる変数に値の割当が起こる節についてはその節の実行を一時的に中断する。そのような節は、他の述語の実行によって呼び出し側の変数に充分な値が得られて後、実行を再開する。

CCLでは、述語が並列に動くプロセスとして扱われ、通信チャネルとしてストリーム(構造上は、リスト表現)が用いられる。我々が提案するレイヤードストリームは、構文的には単に自分自身と相似な構造を内部に含むデータからなるストリームであるが、次のような特徴を持つCCLの並列プログラムの実現に役立つと考えている。

1. 問題の再帰的な記述からの並列プログラムの導出
2. 並列プログラム中における部分問題の解の共有
3. 問題のAND並列性およびOR並列性両者の抽出
4. ストリーム(すなわち、通信チャネル)の動的な結合

レイヤードストリームは、一般に次のような構造をしている。

$$[a_1*LS_1,\ a_2*LS_2,\ \cdots\ ]$$

ここに、'a_i'は項、'*'は結合記号、'LS_i'は内部に含まれるレイヤードストリームである。レイヤードストリームは、一般的には、問題の部分解の集合を表すと考えられる。項は、問題の部分解の一部を成すデータであり、問題によって異なる。結合記号は、そのようなデータとより規模の小さい部分解の集合を表すレイヤードストリームを結合するための記号であり、ここでは'*'を用いることにする。上の例の先頭のデータ、

$$a_1*LS_1 = a_1*[b_1*L_1,\ b_2*L_2,\ \cdots\ ] \quad \text{は、}$$

$$[a_1*b_1*L_1,\ a_1*b_2*L_2,\ \cdots\ ]$$

という集合を表している。これを再帰的に適用すると、元のレイヤードストリームは、それを結合記号'*'についてすべて展開したデータの集合を表現していると考えることができる。

このような部分解の再帰的な表現によって、後に述べる探索問題の再帰的な記述との対応が明確になる。しかも、部分解を構成する項の部分の生成および処理と、内部のレイヤードストリームの生成および処理を独立に、すなわち、並列に行なうことを可能にする。レイヤードストリームの構造の縦方向の並列処理がAND並列に、横方向の並列処理がOR並列に対応する。また、レイヤードストリームは部分解の共通な部分を共有したデータ構造であり、個別の部分解に対する処理を重複して行なうことを避けることができるので、逐次的に実行しても効率のよりアルゴリズムを得ることが可能になる。

また、ストリームのデータ内部に含まれているストリームが新しく生まれたプロセスに渡されることによってストリームすなわち通信チャネルの動的な結合が行なうことができ、動的なプロセス間通信が実現できる。

3 計算量オーダの保存

従来のピュアな (純関数型や純論理型の) 言語では, いわゆる副作用を排除したため, 手続き言語と同じ計算量オーダになるアルゴリズムを用いることができなくなっているものが多かった.

ピュアな言語にすると並列処理が容易になるので, アルゴリズムの持つ理論上の並列度に匹敵するほどのハードウェア並列度を持つ超高並列処理を行なう場合には, 並列処理しやすいという利点が大きく効き, このような欠点は問題にならない場合もある. しかし, アルゴリズム上の並列度が利用可能なハードウェアの並列度よりもかなり大きくなると, 計算量のオーダの違いは致命的になる.

そこで, KL1 では従来の手続き言語と同じ計算量オーダを実現できるベクタの「更新」機構とストリームのマージ機構を用意した. これらの機構を利用すれば, ロック機構を持つような共有メモリのランダムアクセス機構を, 一定の手間でエミュレートできる. つまり, 手続き言語上のどんなアルゴリズムでも, 計算量のオーダとしては同一になるような実現が可能になったわけである.

3.1 総計算量, 並列度と計算時間の関係

最適なアルゴリズムはハードウェアの持つ並列度と問題の大きさとの関係から決まる.

たとえば, 問題のサイズ n に対して $n^3 \log n$ の総計算量になるアルゴリズム A と, n^3 の総計算量のアルゴリズム B があったとする. 逐次処理の場合はもちろんアルゴリズム B が有利であるが, 並列処理の場合はアルゴリズムの並列度が問題になる. アルゴリズムの持つ並列度は, A は n^2 程度, B は n 程度と低かった場合, どちらのアルゴリズムが有利だろうか.[6]

問題のサイズ n を 2^5 程度, ハードウェアの並列度を 2^{10} 程度と仮定すると, 総計算量は A が $5 \cdot 2^{15}$, B だと 2^{15}, 並列度はそれぞれ 2^{10}, 2^5 程度となる. このため, 計算時間は A では $5 \cdot 2^5 = 160$ 程度, B では $2^{10} = 1024$ 程度となり, アルゴリズム A がはるかに有利である. これはアルゴリズム B ではハードウェアの並列度が十分に生かせないからである.

同じハードウェアで $n = 2^{10}$ 程度の問題を解くとどうなるだろう. 総計算量は A が $10 \cdot 2^{30}$, B が 2^{30} 程度になる. 並列度は 2^{20} と 2^{10} 程度になるが, ハードウェアの並列度が 2^{10} しかないので, 実際にはこれに抑えられてしまう. このため, 計算時間は A が $10 \cdot 2^{20}$, B が 2^{20} 程度となり, こんどは B の方がはるかに有利になる.

この傾向は問題のサイズが大きくなるほど顕著になる. これは, ハードウェアの規模が小さ過ぎて, アルゴリズム A の並列度が生かせなかったからである, という見方もあり得るだろう. しかし, 現実に利用できるハードウェアはいくらでも大規模にできるというものではない. また, 計算量理論の考え方は問題のサイズを大きくしていった極限を考えるものであるという立場からしても, アルゴリズム B の方が優れているというのが適当であろう.

このように, ハードウェアを固定して大きな問題を考えると, 総計算量のオーダがアルゴリズムの良否を決める支配的要因であることは, 逐次計算の場合とまったく変わりない.

3.2 ランダムアクセス可能な構造体

従来のピュアな言語では, ランダムアクセスできる記憶, すなわちコンスタントの手間で参照・更新ができる構造体がないのが普通だった. これでは, 手続き処理を前提に開発されたアルゴリズムの多くを, 同じ計算量では実現できない. この点を改善するために, KL1 にはベクタ型のデータとその要素の参照や「更新」を行なう組込み述語を導入した. 導入に当たってはセマンティクスをピュアに保った上で, 想定した使用法を守る限りは「更新」にあたる操作を一定の手間でできるような処理方式を用いている.

KL1 言語の表層上は要素の「更新」を行なうわけではない. 組込み述語

set_vector_element(OldV, Pos, OldE, NewE, NewV)

は「ベクタ OldV の Pos 番目の要素は OldE で, それを NewE に置き換えたようなベクタが NewV である」という意味を持つ. GHC であるから双方向性は失われており, OldV, Pos については実行前に具体化を待ち合わせる. NewV は意味上ここで新しく作られるものであって, OldV とは別のものである.

[6] ここでは通信遅延が無視できるようなアルゴリズムを想定している.

例2　グラフの最短経路問題

グラフ内のある決められた始節点と終節点を結ぶ最小コストの経路を発見する問題。

$$f_k(v) = \min_u \left(f_{k-1}(u) + d(u,v) \mid (u,v) \in E\right)$$

$f_k(v)$: 始節点から k 本以内の枝を通って節点 v へ至る最短経路長

$d(u,v)$: u と v を結ぶ枝 (u,v) の長さ

整合性: $f_{k-1}(u)$ と $d(u,v)$ において u が同一節点でなければならない

多様性: すべての可能な節点 u を対象とする

目標関数: $f_k(v)$ の最小値

例3　N 個の行列の積のコストの最小値問題

与えられた N 個の行列の積を最小回数の乗算によって求めるための計算順序を決定する問題。

$$matrix_{ij} = \min_k \left(matrix_{ik} \oplus matrix_{kj}\right)$$

$matrix_{ij}$: 第 i 行列から第 j 行列までのすべての行列の積を計算するためのコスト

整合性: 右辺の行列が隣接していなければならない

多様性: すべての可能な k $(i \leq k \leq j)$ を対象とする

目標関数: $matrix_{ij}$ の最小値

例4　文脈自由文法の構文解析

与えられた文脈自由文法に基づいて、N 個の単語よりなる入力が文と認められるかとうかを判定する問題。

$$NT_{ij} = \underset{k_1,k_2,\cdots k_n}{\text{exist}} \left(NT_{ik_1} \oplus NT_{k_1k_2} \oplus \cdots \oplus NT_{k_nj}\right)$$

NT_{ij}: 入力の第 i 単語から第 j 単語によって構成される非終端記号

整合性: $NT_{ij} \leftarrow NT_{ik_1}\ NT_{k_1k_2} \cdots NT_{k_nj}$ という文法規則が存在しなければならない

多様性: すべての可能な文法規則を対象とする

目標関数: NT_{ij} が少なくとも一つ存在する

3.2　条件を実現するためのメカニズム

次の節で示すように、再帰的に記述された問題の解や解の集合を表現するために前節で導入したレイヤードストリームを利用する。本節では、問題の再帰的記述において定義された種々の条件をプログラム上で実現するための基本的なメカニズムを説明する。

レイヤードストリームプログラミングでは、部分問題の解や解を構成する要素がレイヤードストリームを用いて表現され、規模の小さな部分問題の解や解の構成要素からより大きな問題の解を構成するようにプログラムされる。その中で、多様性条件は、次節で説明するように、利用可能な部分解や解要素に対応する情報をレイヤードストリームとして生成する操作として実現される。

一方、整合性条件と目標関数は、レイヤードストリームの中から問題の条件に合致しない部分を取り除くフィルタープロセスとして実現される。整合性条件に対応するフィルターは consistency filter と呼ばれる。また、目標関数を実現するフィルターを preference filter と呼ぶ。これらについては次節で例を用いて説明する。

この例のように，逐次処理の場合は逐次制御と条件判断で実現できた方式が，並列処理まで拡張して考えるとメタレベルの制御 (優先度の指定と実行の放棄) を用いなければ最適にならない場合が少なくない．このことも KL1 にメタレベルの実行管理機能を導入する大きな動機になった．

2 資源管理機能

2.1 必要性

メタレベルプログラムがオブジェクトレベルプログラムを管理するには，オブジェクトレベルプログラムがどのようなふるまいをしているかを，少なくともマクロには把握していなければならない．前節で述べた実行管理機能を用いれば，オブジェクトプログラムの実行の終了，異常事態の生起といった，計算の進行状況の質的な把握は可能である．しかし，それだけでは量的な側面，たとえばどのぐらい計算が進行しているのか，といったことはわからない．

一般的なメタレベルの機構として導入するには，解いている問題に依存しない計算進行状況の観測機構が適当である．どれぐらいの時間計算しているのか，どのぐらいのメモリを費やしているのか，といった計算機構そのものに関わる測度を用いれば，同じ計算機構を用いているかぎり共通のものであり，メタレベルの計算戦略を立てるためのデータとして有効である．

一方，オペレーティングシステムのようなプログラムを考えると，たとえばユーザプログラムがメモリをすべて消費してしまうと，オペレーティングシステムが動くためのメモリが残っていない，ということも生じ得る．これを防ぐには，メタレベルにはオブジェクトレベルでの消費計算資源を制限する能力が必要である．

そこで，こうした情報を大きなオーバヘッドなく得るために，言語のメタレベル機能の一環として資源管理機能を導入することとした．

2.2 方針

2.2.1 管理単位

資源消費の管理単位は，実行管理と同じく荘園とした．これは資源管理の結果として実行を制御したい場合が多いので資源管理単位を実行管理単位と一致させれば制御が容易であること，処理系実現上も資源消費単位を別に設けるとオーバヘッドを増すこと，の二点が理由である．

資源消費の報告の仕方として，単純にはオブジェクトレベルからメタレベルに資源の消費状況を逐一報告することが考えられる．しかし，この方法ではメタレベルが常にオブジェクトレベルを監視していなければならず，そのオーバヘッドが大変大きくなってしまう．そこで，あらかじめ適当な資源消費許容量をメタレベルから設定し，それが尽きそうになるまではオブジェクトレベルはメタレベルに特に報告することなく動作できるものとした．許容量が尽きそうになったことを知らされたメタレベルでは，

- 許容量を追加して計算を続行させる．
- 許容量の追加はせず，その計算は打ち切る．
- 許容量の追加はせず，いずれその計算は資源枯渇によって中断するにまかせ，当面他の方法をためすが，将来継続することもできるようにしておく．

などの選択が可能である．一回に与える許容量の単位を小さくすれば精度の良い資源管理ができるが，それだけオーバヘッドも増えることになる．

2.2.2 管理対象

管理の対象としては以下のようなものが考えられる．

計算時間: 当該荘園の計算のためにどのぐらい時間をかけてよいか．

メモリ消費量: 当該荘園の計算のためにどのぐらいメモリを消費してよいか．

```
queens(Ans) :- true |
   q(LS), qs(LS,[],Out-[]).
q(LS) :- true |
   qX(begin,Q1), qX(Q1,Q2),
   qX(Q2,Q3), qX(Q3,LS).
qX(In,Out) :- true |
   Out = [1*In,2*In,3*In,4*In].
qs(begin,Board,From-To) :- true |
   From=[Board|To].
qs([I*LSi|LSs],Board,From-To) :- true |
   checkQs(Board,I,1,LSi,Board,From-M),
   qs(LSs,Board,M-To).
qs([],_,From-To) :- true | From=To.

checkQs([Q|Qs],Q,D,In,Board,Out) :-
   true | Out=End-End.
checkQs([Q|Qs],I,D,In,Board,Out) :-
   Q=:=I+D | Out=End-End.
checkQs([Q|Qs],I,D,In,Board,Out) :-
   Q=:=I-D | Out=End-End.
checkQs([Q|Qs],I,D,In,Board,Out) :-
   Q=\=I+D, Q=\=I-D, Q\=I |
   D1 := D+1,
   checkQs(Qs,I,D1,In,Board,Out).
checkQs([],I,D,In,Board,Out) :-
   true | qs(In,[I|Board],Out).
```

このプログラムでは'qX'が一つのクイーンに相当し、これによって4クイーン問題の解になり得る組合せがレイヤードストリームの形で構成される。'qs'が部分解に対応するプロセスであり、これがレイヤードストリームを消費しながら新しい部分解へと分岐していく。

後者のプログラムは、クイーンのあらゆる可能な位置をしらみつぶしに調べるのに対し、前者では、レイヤードストリームの構成と共に配置されたフィルターが同時に枝狩りを開始し、しかも密にデータの共有を行なっている。実験による比較では、4クイーンの場合の総リダクション数(頭部単一化の数)が前者と後者とでは、160対1615、6クイーンでは、1,382対236,185であった。クイーンの数が増すとこの差は急激に拡がる。これは、レイヤードストリームを効果的に使った場合の解の共有の効果を顕著に示す例である。

次に文脈自由文法の構文解析の問題を用いて、もう一つのタイプの再帰的記述による問題のプログラミングを説明する。この問題の場合は、記述の右辺も左辺も部分問題しか含まないので、部分問題の解(すなわち、非終端記号)を並列プロセスと考える。右辺の部分問題同志は情報のやりとりを行なう必要があるので、各プロセスは隣のプロセスへ自分の存在を示すデータ(非終端記号名)を送る。相互にデータを送り合う必要はないので、必ず左のプロセスが右のプロセスへデータを送ると仮定しよう。具体的には、各プロセスは左のプロセスから受け取ったデータ(レイヤードストリーム)の先頭に自分の持つ情報を付け加えた新しいレイヤードストリームを右のプロセスへ送れば良い。このようにして、構文解析の並列プログラムを実現することができる。初期プロセスは、解析しようとしている入力を構成する品詞の列に対応するプロセス列である。

右辺の部分解をどのようにして結合するかということに幾つかの選択肢がある。構文解析システム PAX[Matsumoto 86]は、レイヤードストリームに基づくシステムであるが、このシステムでは、例えば、a ← b c d という文法規則に対して、下の(1)に対応するプログラムを生成する。例えばcについて見ると、cは左隣に bが存在することを通信によって知ると、非終端記号bとcが連続して現れたという情報をデータ'b_c'で表現し、それを右隣へ送っている。一方、別の考え方も可能で、cは自分の名前を右へ送っているのであるから、それを受け取ったプロセスdが、cとdが連続して現れたことを示すプロセス'c_d'を生成してもよい。このような考え方が[佐藤 90][山崎 90]によって提案されている。つまり、前者ではプロセスの並びをデータで表現しているのに対し、後者ではそれをもプロセスで表現しているのである。後者の本質的な部分のプログラムを(2)に示す。

1.2.3 指定方式

優先度の指定は，荘園単位におおまかに指定する方法と，各ゴールごとに細かく指定する方法の両者を用意した．メタレベルプログラム的な意味でおおまかに制御する場合 (たとえばオペレーティングシステムがユーザプログラムの優先度を管理する場合) には荘園単位の指定，オブジェクトレベルの知識を元に細かい制御をしたい場合 (たとえばアルファベータ探索プログラムで，深さ優先探索をしたい場合) にはゴール単位の細かい制御ができるようにするためである．

ゴール単位の指定はそのゴールそのものの実行の優先度を指定する．荘園に対する指定は，荘園内のゴールの優先度の最大 / 最小値を指定することによって行う．荘園生成のための述語にはこの優先度範囲指定のための引数を追加する．

指定は荘園単位の指定・ゴール単位の指定のいずれも，優先度の絶対値ではなく，実行中の荘園の優先度範囲や指定を行う計算自身の優先度と比べての相対的な値として与える．具体的には以下のいずれかの方法を用いる．

1. 指定を行う計算の属している荘園内での相対値．属している荘園の優先度範囲内のどのあたりで実行するかを，割合で指定する．
2. 指定を行う計算自身の優先度と，属している荘園の最大 (または最小) 優先度との間のどのあたりで実行するかを，割合で指定する．

絶対値ではなく相対値を用いることには，以下のような利点がある．

- 優先度の絶対値の範囲は処理系ごとに異なることが考えられるが，相対記述ならプログラムには処理系独立な優先度指定記述ができる．
- 局所的に相対優先度指定を付したプログラムを，大局的に荘園全体の優先度範囲を変えて呼び出すことができる．

1.2.4 暴走の停止

優先度の概念がなかったり，オブジェクトレベルの優先度がメタレベルの優先度以上になったりすると，オブジェクトレベルプログラムの暴走をメタプログラムから制止できなくなる．たとえオブジェクトレベルの実行を停止するためのメタプログラムのゴールを実行しようとしても，オブジェクトレベルの暴走のためにメタレベルはたったひとつのリダクションも実行されないことがあり得る．

この問題を解決する方法のひとつに，スケジューリングに公平さを導入する方法がある．公平なスケジューリングでは，実行可能状態にあるどのプログラム部分も有限時間内には少なくとも少しは実行されることを保証する．このためには，幅優先や制限つき深さ優先のスケジューリングを行なえばよい．

一方，優先度の機構を用いてもこの問題は解決できる．オブジェクトレベルが決してメタレベルよりも高い優先度を持たないように，荘園を生成する際の優先度範囲指定を行えばよい．逆に，優先度機構があれば，プログラムの暴走防止の目的で公平なスケジューリングを行う必要はない．

スケジューリングが不公平でよければ，プログラムの局所性を最大限に生かすような，あるいは実装が最も容易なスケジューリング方式を自由に選択できる．極端な例では，公平なスケジューリング方式で最も簡単である幅優先スケジューリングより，公平さを保証しない最も簡単なスケジューリング方式である深さ優先スケジューリングの方が，通常時間的な局所性をはるかに高くでき，ワーキングセットを小さくできる．また，実装も簡単である．

暴走の停止を保証するためには処理系がある程度は優先度を守る保証が必要である．それは，他に低優先度の計算がいくらあろうとも，有限時間内に終了できるような最高優先度の計算は有限時間内に終了する，ということである．

1.3 クローズ間の優先度

GHC では，複数の候補節のどれを用いてもリデュース可能な場合，どの節を用いるかは，言語仕様としては定めないことになっている．これは言語仕様を簡潔に保つことが本来の動機である．処理系の作成

謝辞

後半の例として用いた構文解析のプログラミングの色々な可能性については、ICOT の瀧和男氏、三菱の佐藤裕幸氏、富士通の山崎重一郎氏との議論に負うところが大きかった。感謝いたします。また、日頃貴重な意見を賜わる ICOT 並列プログラムワーキンググループの諸氏に感謝したします。

参考文献

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

[Clark & Gregory 86] Clark, K.L. and Gregory, S., "PARLOG: Parallel Programming in Logic," ACM Trans. Programming Languages and Systems, Vol.8, No.1, pp.1-49, 1986.

[Ueda 86] Ueda, K., "Making Exhaustive Search Programs Deterministic," Proc. 3rd ICLP, '*Lecture Notes in Computer Science 225*', pp.270-282, 1986.

[Tamaki 87] Tamaki, H., "Stream-based Compilation of Ground I/O Prolog into Committed-Choice Languages," Proc. 4th ICLP, The MIT Press, pp.376-393, 1987.

[Matumoto 86] Matsumoto, Y., "A Parallel Parsing System for Natural Language Analysis," Proc. 3rd ICLP, '*Lecture Notes in Computer Science 225*', pp.396-409, 1986.

[Okumura & Matsumoto 87] Okumura, A. and Matsumoto, Y., "Parallel Programming with Layered Streams," Proc. 1987 International Symposium on Logic Programming, pp.224-232, San Francisco, September 1987.

[奥村 & 松本 87] 奥村晃、松本裕治、「レイヤードストリームを用いた並列構文解析」、情報処理学会第 35 回 (昭和 62 年後期) 全国大会予稿集、pp.861-862, 1987.

[松本 & 奥村 88] 松本裕治、奥村晃、「Dynamic Layered Stream Programming」、日本ソフトウェア科学会第 5 回大会論文集、pp.377-380, 1988.

[佐藤 90] 佐藤裕幸、「並列自然言語構文解析システム PAX の改良」、KL1 プログラミング・ワークショップ、ICOT, May 1990.

[山崎 90] 山崎重一郎、「並列自然言語解析システム LaPuta について」、KL1 プログラミング・ワークショップ、ICOT, May 1990.

NewGoal: 例外を起こしたゴールの代わりに実行すべきゴールを指定するための変数.

例外の原因別に適当なタグを割当てる. 例外の報告は, 例外を起こした荘園, その親荘園, そのまた親, という先祖荘園の中で, 例外原因タグにマッチするタグを持つ最も近い先祖荘園の報告ストリームに流す. 荘園生成時に指定するタグはビットマスクで, 例外原因タグとビットごとの論理和を取って, ゼロでない荘園に報告する. こうすることによって, 特定の例外だけを取り扱い, 他はより外側の荘園に任せるような記述が可能になる.

例外の原因 例外の原因としては, ボディ部の組込み述語に対して誤った型の引数を与えた場合, 処理できない値の引数を与えた場合 (演算あふれやゼロ除算など), そしてゴールが失敗した場合などがある. KL1 ではユニフィケーションの失敗や, ガード部の成功する候補節がないことによる失敗は, 例外として扱う. したがって, 荘園の実行が失敗によって終了することはない. 失敗した際に荘園の実行を放棄したければ, 制御ストリームから放棄メッセージを送れば良い.

ガード部では例外が生じることはない. ガード部の組込み述語の型が述語の引数型にあっていない場合 (たとえば加算の引数にアトムを与えたような場合) は, 例外とせず単に失敗とする. これは, 例外による中断 / 再開の機構を簡素化するためである.

積極的な例外の生起 ユーザが積極的に例外を起こすことができるように, 以下のような組込み述語を用意する.

raise(Info, Data, Tag)

ここで, 各引数は以下のようなものである.

Info: 例外に関する情報. この引数が完全に (構造体の場合はその要素も, 要素が構造体の場合はそのまた要素と, 構造体全体が) 具体化されるまで例外の生起は遅延される.

Data: 任意のデータ. この引数は具体化されていてもいなくてもよい.

Tag: 例外のタグを指定する.

例外を取り扱うプログラムは例外を出すプログラムのメタプログラムである. このメタプログラムがデッドロックすることを防ぐためには, 具体化されていることが保証された例外情報が必要である. 組込み述語 raise の引数 Info はこのような目的に用いるためのものである. 一方, 引数 Data は, メタプログラムが中身を読まなくても良いデータ (たとえば, 後述の実行継続に使うデータ) を渡すのに用いる.

例外後の実行の継続 荘園内では例外を起こしたゴールの実行は中断され, NewGoal の具体化を待って, 元のゴールの代わりに NewGoal を実行する. この際, 荘園には NewGoal が具体化されたらそれを実行しようとするゴールがあると考えて良い. このゴールがあるため, NewGoal を具体化しないうちに荘園全体が先に成功裏に終了してしまうことはない.

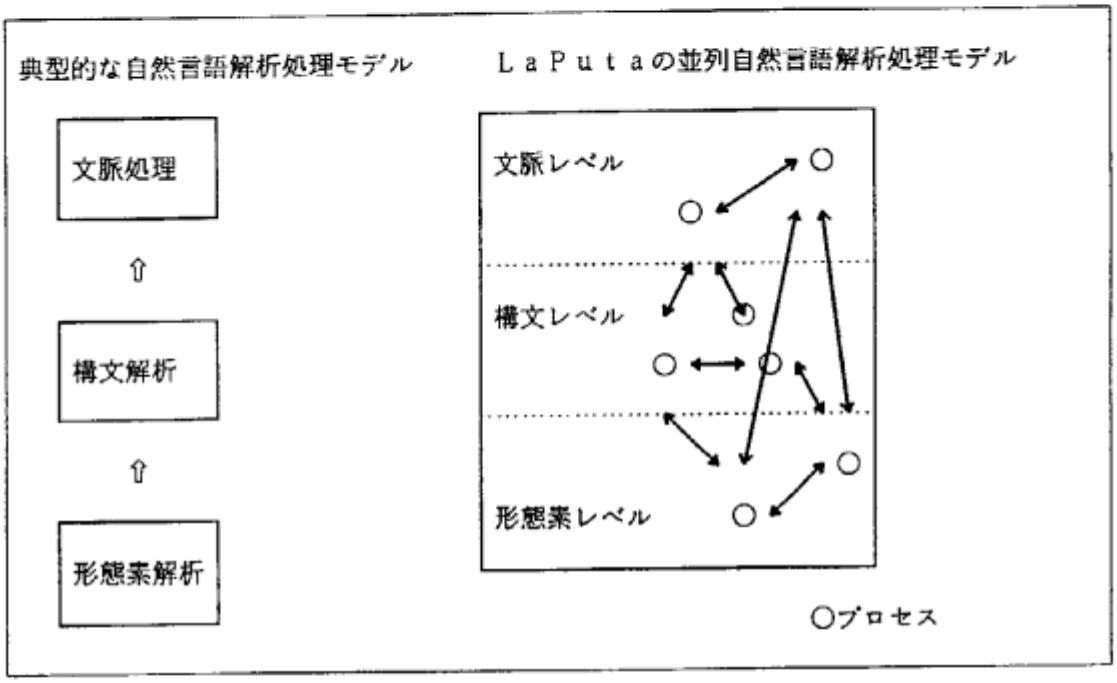
例外処理を行う側で NewGoal を適当に与えることによって, 言語仕様を拡張することができる. たとえばあるパターンのユニフィケーションの失敗に対する処理を記述することによって, ボディ部でのユニフィケーションの仕様を拡張することもできる. このように積極的に例外を利用する場合, 荘園の実行が自動的に中断されてしまうような仕様は効率面から好ましくない. そこで, 同じ荘園内でも, 例外を起こしたゴール以外のゴールは正常に実行が継続されるものとする.

例外機構の意味 KL1 の例外機構は, すべてのゴールが暗黙のうちに報告ストリームにつながるようなストリームを持っていると考えることで, GHC の範囲内で説明できる. つまり

```
p(X,Y) :- true | q(X,Z), r(Z, Y).
```

のような節ひとつからなる述語定義は

レベルの違いはここでは単にデータの違いであって処理方法の違いではない．そして様々なレベルのデータが同時並行的に処理されることによって並列協調処理が実現される．また，開発の観点から見ると，各レベルの文法はそのレベルの視点で開発することが可能であり，異なるレベルの情報を利用することも可能になる．また，言語現象ごとに個別に開発することも可能である．そして，このようにして開発された文法は，最終的にはマージされて並列的に実行されることになる．



処理モデルの比較

2．基本原理

このような処理モデルを考えると，別の問題が生ずる．処理過程で他の処理プロセスと動的に協調させようとすると，プロセス間の呼出関係が組み合わせ的に増加し，プロセスの構造が爆発的に複雑化するために開発が非常に困難になるという問題である．これを解決するために，我々は形態素解析や構文解析といった色付けの無い汎用的な処理機構を中心とした自然言語解析システムを構築することを考えた．

そして，この汎用的な処理機構の原理として，型推論［Reynolds 85,Hindley 86］と部分項の単一化に基づく制約解消［向井 89 ］を採用することにした．

型推論は与えられた表現に型宣言と呼ばれる環境の下で型を与えるための推論である．表現を単語の列とし，型を品詞もしくは句や文と考え，型宣言を辞書および句構造文法と考えると，入力として与えられた単語の列に対して，辞書に基づいて品詞を割当てたり，句構造文法に基づいて句や文としてまとめあげる構文解析は型推論に対応している．また，入力を文字列とし，形態

ジェクトの階層が重なることもある．仮想機械オペレーティングシステムでは，オペレーティングシステムの中にもメタ / オブジェクトの階層があることになる．

メタプログラムの機能を実現するにはインタプリタを用いるのが簡明である．しかし，インタプリタを用いると実行効率の面で通常一桁程度以上の損失が生じる．部分評価技術などを用いてこの損失を軽減はできるが，現在の技術水準では数倍程度の損失はまぬがれない．ことに，メタ / オブジェクトの階層が重なると，この損失は階層一段ごとに乗ぜられ，最終的な効率は指数関数的に低くなっていく．

この効率上の問題は KL1 に翻訳されるような言語階層を積み重ねることでは解決できず，メタレベルプログラムとオブジェクトレベルプログラムを同じ土俵 (機械語) で動かすことが必要である．そのためには，メタプログラミング機能を言語自身に導入することが不可欠である．以下に，そのようなメタプログラムの持つべき機能を上げる．

失敗の伝播の制限　FGHC では，すべてのゴールは論理積関係にある．このため，ひとつのゴールが失敗すると，全体が失敗することになる．このような言語でメタレベルプログラムを記述すると，オブジェクトレベルプログラムに何らかの問題があって実行に失敗すると，メタレベルプログラムを含めたシステム全体が失敗することになってしまう．そこで，失敗の伝播範囲を制限する機構が必要になる．

メタ制御機能　メタレベルプログラムには，たとえば，暴走したオブジェクトレベルプログラムを強制的に停止させるなど，オブジェクトレベルプログラムの実行の制御を行えるメタ制御機能が必要である．

監視機能　メタレベルプログラムはオブジェクトレベルプログラムの実行の様子を何らかの意味で監視できなくてはならない．

例外処理機能　メタレベルプログラムはオブジェクトレベルプログラムの実行中に生じた例外事象を検出し，例外事象に応じた適当な処理を行えなくてはならない．

1.1.2　荘園機能

概要　メタプログラミング機能を導入するためには，すべてが平板に論理積となっている FGHC に，なんらかの構造を持ち込む必要がある．これを実現するために導入したのが荘園の機能である．

荘園は以下のようなプリミティブを用いて生成する．[1]

execute(Goal, Control, Result, Tag)

ここで，各引数は以下のようなものである．

Goal: 荘園の中で実行すべきゴール．

Control: 荘園内の実行を制御するためのコマンドを荘園の外部から送るストリーム．

Result: 荘園内の実行の結果によるさまざまな情報を，荘園外部に伝達するためのストリーム．

Tag: 荘園がネストしている場合，それらを識別するためのタグ．

各々の引数の詳細については後述する．

荘園の実行　上述の方法で生成した荘園内のゴール群は，荘園外とは独立した論理積を成す．すなわち，荘園内での失敗は荘園内に閉じたものであり，荘園外のゴールを巻き添えにすることはない．また，荘園内のゴールの実行の終了は実行結果のストリームに通知される (後述)．概念上，荘園はインタプリタを機械語レベルで実現したものであると考えて良い．

[1] 実際の仕様はもう少し複雑だが，ここでは詳細を捨象して述べる．

いる．

⑤ 2 階の型

s : {X | X!subj=NP, X=VP }

この表記は，s という表現の型が，集合の抽象化の表記で表された型の集合（型）すなわち2 階の型をもつ型である．この型を経由して制約解消系と型推論系が融合する．

（2） 推論規則

基本的なものに対する型割当の集合を型宣言と呼び，Γで表記する．Γは辞書や文法規則や意味モデルにあたる

① トートロジー

$$e{:}t \in \Gamma \quad \Rightarrow \quad \Gamma \vdash e{:}t$$

この推論規則は，辞書のアクセスや文法規則のアクセスなどに対応する．

② 関数適用

$$\frac{\Gamma \vdash e{:}t \qquad \Gamma \vdash f{:}t \rightarrow s}{\Gamma \vdash ef{:}s}$$

この推論規則は，句構造規則がe という表現によって充足されることなどに対応する．

③ 和型削除

$$\frac{\Gamma \vdash e{:}\ [t_1,\ldots,t_n]}{\begin{array}{l}\Gamma \vdash e{:}t_1 \\ \quad\vdots \\ \Gamma \vdash e{:}t_n\end{array}}$$

この推論規則は，型に曖昧性があるときに探索空間を分岐させることに対応している．

④ 対称関数適用

$$\frac{\Gamma \vdash e{:}t_j \qquad \Gamma \vdash f{:}\ [t_1,\ldots t_{j-1}, t_j, t_{j+1},\ldots t_n] \rightarrow t}{\Gamma \vdash ef{:}\ [t_1,\ldots t_{j-1}, t_{j+1},\ldots t_n] \rightarrow t}$$

この推論規則は，引数をとる順序が自由な関数の適用を表している．

⑤ 対象化

$$\frac{\Gamma \vdash e : t \qquad \Gamma \vdash t : s}{\Gamma \vdash e : s}$$

この推論規則は，型がものとしても型宣言されているならば，型をものとして扱ってよいということを主張するものであるが，このような推論規則が型推論としてはあまり適切ではない．

現在の時点では，対象化が可能な型は原子型に限っている．

この推論規則の存在によって，ものと型との境界は不明確になる．

を並列実行させたからといって飛躍的な台数効果が得られるとは限らない. それは, 並列メタインタプリタでは, 実行終了条件 (総てのゴールキューが空である状態) やデッドロック条件 (総てのゴールがサスペンドしている状態) を 1 リダクション毎に同期をとって判定する必要があるからである. 図 4. にその処理イメージを示す.

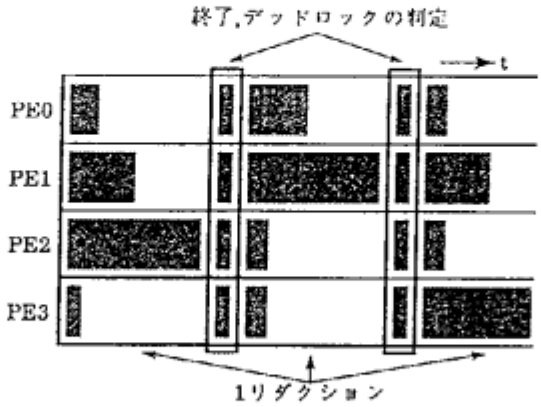


図 4. 情報収集のための同期処理

4.2 パフォーマンスモニタの拡張

パフォーマンスモニタによる負荷状況の推移を用いて, 詳細なプラグマ設定支援を行うためには, トレース情報を参照できる機能が必要である.

この, トレース情報の参照機能は, パフォーマンスモニタで選択したある時点におけるゴールキューの内容, あるいは, その時点の負荷を発生させるきっかけとなったリダクションを示すことを目的とする. そこで, 負荷情報とメタインタプリタから出力させたトレース情報を対応させることにより, プラグマ設定支援を行えるような機能の拡張を考えている.

5 まとめ

KL1 プログラムの設計支援システムにおいて, プラグマ設定支援を目的として開発した, メタインタプリタを利用したパフォーマンスモニタを紹介した. 現時点におけるメタインタプリタの実行速度は, 実用的とは言い難いものであり, この, 実行速度の向上を目指した並列メタインタプリタを現在開発中である. 今後は, さらにトレース情報と負荷情報のリンクおよび, プロセッサ間の通信量を示す機能等の開発を目指す.

参考文献

[Tanaka 88] 田中, 的場 : 変数管理をする *GHC* の自己記述, *ICOT Technical Report:TR-374, 1988*

[Tanaka 87] 田中, 太田, 的場, 神田 : *GHC* による仮想ハードウェアの構築とリフレクト機能について, 日本ソフトウェア科学会第 *4* 回大会論文集, *B-5-3, 1987*

[Sato 84] 佐藤義雄 : 実習グラフィックス, アスキー出版局, *1984*

3．構文解析手法

ＬａＰｕｔａの型推論機構を本章では構文解析機構として説明する．ＬａＰｕｔａの構文解析手法は，基本的にはＰＡＸ［松本 86］をもとにＩＣＯＴの瀧氏によって最初に考案され［佐藤 90］，我々が再発見したものである．ＰＡＸとの違いは，プロセスとストリームの関係が反転している点で，ＰＡＸが，ストリームの構造で探索空間を表現していたのに対して，ここで提案する方式はプロセスの親子関係で探索空間を表現するものである．

3．1．ＬａＰｕｔａとＰＡＸの得失

この方式は，ＰＡＸのように適用可能な全ての文法規則をプロセス間の通信ストリームに流すのではなく，文法規則が実際に完全に充足されたときに構成が成功した非終端記号のみをストリームに流すことによって，プロセス間の通信量を減少させている．

適用可能な全ての文法規則と実際に構成が成功した非終端記号とでは，非終端記号の方が量的に少ないのは明らかなので，通信量の観点からはＬａＰｕｔａの方式の方が有利である．通信量を削減したことのトレードオフとして処理過程で生成されるプロセスの数はＬａＰｕｔａの方が増える．

3．2．ＬａＰｕｔａの辞書／文法

ＬａＰｕｔａでは，終端記号と非終端記号の区別は無い，辞書および文法規則は次のように記述される．

人間 ： 体言語基．	(DCG で書くと	体言語基 --> 〔人間〕．)
が ： (体言語基 -> 名詞句)	(DCG で書くと	名詞句 --> 体言語基，〔が〕．)
述語 ： (名詞句 -> 文)	(DCG で書くと	文 --> 名詞句，述語 ．)

この記述法で，「 :」の左を表現，右を型と呼ぶことにする．

3．3．構文解析の基本動作

ＬａＰｕｔａ構文解析の基本動作は，大きく分けると，文法規則の充足のための処理と非終端記号が構成された時の処理の２つの動きから構成されている．

```
            VarID1,BindTBL1,Reports1).
pes1(fail,_,_,_,_,_,_,_,Reports) :- true |
    Reports = [].
otherwise.
pes1(_,ProcID,NumOfProc,Clauses,Goals,GoalQueue,
    VarID,BindTBL,Reports) :- true |
    pes(~(ProcID + 1),NumOfProc,Clauses,Goals,GoalQueue,
        VarID,BindTBL,Reports).
```

述語 pes/8 は, 前述の述語 mi から pes(NumOfProc,NumOfProc,Clauses,Goals, GoalQueue, VarID,BindTBL,Reports) という形で呼ばれる. ここで NumOfProc は全プロセッサ数である. 呼ばれた直後は GoalQueue は空ではないので 2 番目の節がコミットされ, ゴールキューの中身, すなわち起動ゴールの代入例が eval_term/3 で求められ, 報告ストリーム Reports に出力される. この出力の後, プロセッサ識別子 0 番から NumOfProc−1 番までが順に実行される. すなわち, あるプロセッサ識別子 ProcID を持つプロセッサのリダクションを試みたならば, ProcID を 1 インクリメントし, その ProcID を持つプロセッサの行なうリダクションとして実行を試みるのである. ProcID がインクリメントされてゆき, 全プロセッサ数と等しくなったならば, ゴールキューの代入例を報告ストリームに流し, 再びプロセッサ識別子 0 番よりリダクションを試みる.

前述のメタインタプリタと大きく異なるのは, ゴールキューである. このゴールキューではキュー内のゴールに, タグとしてプロセッサ識別子が取り付けられている. 述語 get_goal(ProcID,GoalQueue, GoalQueue1,BindTBL,G) は get_goal/4 に対して, 変数 ProcID で示されたプロセッサ識別子と同じ識別子のタグの付いたゴールのみを返すよう変更されている. またトレース情報の出力においてもどのプロセッサでのリダクションであるかを示すため, 出力情報にタグとして ProcID を付加している.

3 パフォーマンスモニタによる動作情報表示

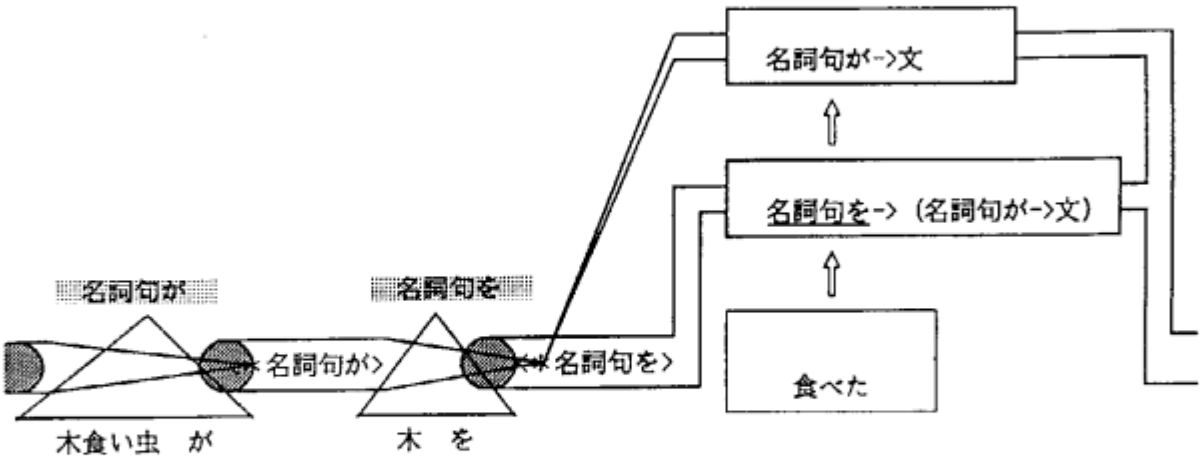
パフォーマンスモニタにおける動作情報表示部は, Multi-PSI FEP 上で ESP 言語を用いて開発したものである. この表示部は, メタインタプリタから出力される動作情報を一定の形式に変換し, 対象プログラム実行時における, プロセッサ負荷状況の時間的推移を視覚的に表現することを目的としている. ここでメタインタプリタが負荷として扱っているのは, ゴールキュー中に存在するゴール数である.

動作情報の表示は, 全プロセッサの負荷状況を同時に表示するために, 時間 − 負荷 − プロセッサの三次元グラフで示した. また, 目的に合わせた表示を得るために, 三次元コンピュータ・グラフィックス [Sato 84] を用いて表示した. 動作情報を任意の視点で表示し, 変更することを可能とした. さらに, 任意の時間あるいはプロセッサに注目した表示を行うために, 時間 − 負荷及び, プロセッサ − 負荷の各二次元グラフ表示を可能とした.

3.1 三次元表示機能

三次元表示では, 座標原点側をプロセッサ 0 としており, 負荷の最も大きい点を 100% に合わせてスケールを決定している. また, グラフ上の時間は, プロセッサが一つでも立ち上がった時点を 0 とし, 全てのプロセッサに負荷がなくなるまでを表している.

するストリームである.



この実装方法では，ストリームは左から右にデータを流すので，文法規則の右隅の要素に対応する表現をもとに型にプロセスを生成し，その左側の表現と結合して構文木が構成されていくような，右隅解析法になっている.

しかし，右隅解析法の採用は特に本質的ではない，もし我々のシステムで左隅解析法を用いると反対にストリームの中の通信データは右から左に流れることになるというだけの違いである.しかし，日本語では，助詞や述語のような文法的構造を決定する語や句は右隅に現れる傾向があり，それ以外の要素は比較的自由な順序で出現するので，右隅解析法の方が有利であると判断した.

（2）非終端記号が構成されたときの処理

もうひとつの動作は，文法規則の適用の結果，非終端記号が構成されたときの処理である．また，最初に入力された終端記号／非終端記号に対しても基本的にこれと同様の処理を行う．このとき，二つの処理が対になって発生する.

一つは，構成された非終端記号（または終端記号）を現在処理を行っているプロセスの入力ストリームと対にして出力ストリームに流すことである.

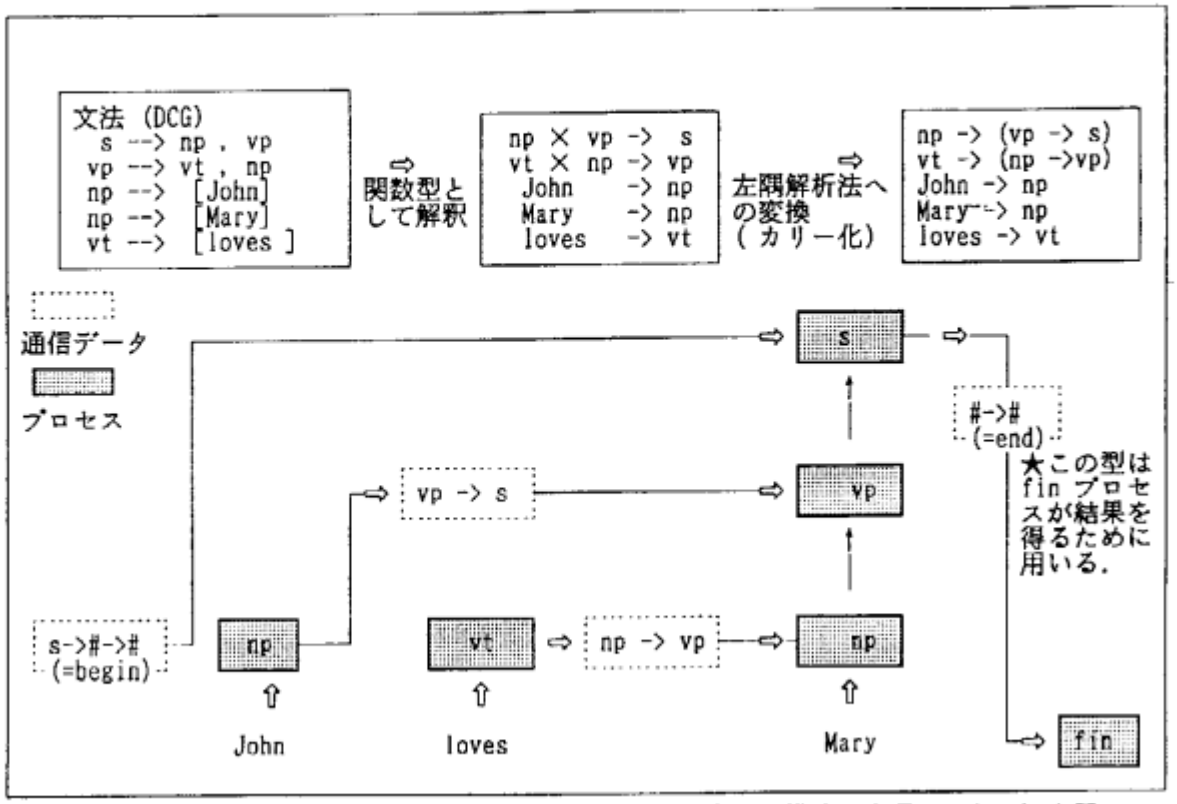
もうひとつは，その非終端記号（または終端記号）を表現とみなして辞書／文法にアクセスし，その表現の型を得て，型が「->」の記号を含む場合にプロセスを生成することである．一般に文法や辞書に曖昧性が含まれる場合，型は和形の形で記述されているが，これに対してＯＲ並列的にプロセスを生成し，入力ストリームも同一のストリームをこれらのプロセスの入力ストリームとする.

```
        freeze_term(Goals,VarID1,VarID2),
        put_goals(Goals,[],GoalQueue),
        pes(Clauses,Goals,GoalQueue,VarID2,[],Reports).

    pes(_,Goals,[],_,BindTBL,Reports) :- true |
        eval_term(Goals,EGoals,BindTBL),
        Reports=[exit(EGoals)].
    pes(Clauses,Goals,GoalQueue,VarID,BindTBL,Reports) :- true |
        get_goal(GoalQueue,GoalQueue1,BindTBL,Goal),
        exec_body(Goal,RGoals,VarID,VarID1,BindTBL,BindTBL1,Clauses,Result),
        eval_term(GoalQueue,EGoalQueue,BindTBL),
        eval_term(Goal,EGoal,BindTBL),
        eval_term(RGoals,ERGoals,BindTBL1),
        Reports = [goal_queue(EGoalQueue),{Result,EGoal,ERGoals} | Reports1],
        put_goals(RGoals,GoalQueue1,GoalQueue2),
        pes1(Result,Clauses,Goals,GoalQueue2,VarID1,BindTBL1,Reports1).

    pes1(fail,_,_,_,_,_,Reports) :- true |
        Reports = [].
    otherwise.
    pes1(_,Clauses,Goals,GoalQueue,VarID,BindTBL,Reports) :- true |
        pes(Clauses,Goals,GoalQueue,VarID,BindTBL,Reports).
```

述語 mi(Clauses,Goals,Reports) は, メタインタプリタのトップレベルのゴールであり, Clauses にプログラム, Goals に起動ゴールを与えて呼び出すと, 報告ストリーム Reports にトレース情報を返す. このメタインタプリタでは全ての変数は, 述語 freeze_term(X, VarID, NewVarID) によりフリーズされた形で扱われる. X はフリーズする入力項であり, freeze_term/3 は X に含まれる未定義変数全てにベクタ $var(ID) をユニファイする. また, フリーズ処理が終了すると, NewVarID に ID を更新した値をユニファイする. ID は変数識別番号 VarID から始まる整数であり, 変数毎にユニークな変数識別子である. 述語 mi/3 では, プログラム Clauses, 及び起動ゴール Goals を freeze_term/3 によりフリーズしている. また述語 put_goals(Goal,GoalQueue, NewGoalQueue) は, ゴールキュー GoalQueue にゴール Goal を格納し, 格納済みのゴールキューを NewGoalQueue にユニファイする. 述語 mi/3 内では, キューの中に起動ゴール Goals を格納している.

述語 pes(Clauses,Goals,GoalQueue,VarID,BindTBL,Reports) は, 実際にゴールの評価を行なう述語である. Clauses,Goals は freeze_term/3 によってフリーズされたプログラム, 起動ゴールであり, GoalQueue はゴールキュー, VarID は変数識別番号, BindTBL は変数の束縛表, Reports は報告ストリームである.

pes/6 は GoalQueue が空リストになるまでゴールの評価を繰り返す. 先ず, get_goal(GoalQueue, GoalQueue1,BindTBL,Goal) により評価すべきゴールをゴールキューより 1 つ取り出し, 変数 Goal にユニファイする. GoalQueue1 には Goal が取り除かれたゴールキューがユニファイされる. 述語 exec_body(Goal,RGoals,VarID,VarID1,BindTBL,BindTBL1, Clauses,Result) は, ゴール Goal のリダクションを GHC の実行規則に従って試みる. すなわち, ゴール Goal とパッシブユニフィケーション可能な節がプログラム Clauses 内にあるならば, これを試み, 成功する節があるならばその節にコミットし, そのボディ部をリダクションの結果生成された子ゴールとして変数 RGoals に, コミット成功を示すアトム succ を変数 Result にユニファイする. この変数 RGoals



ＰＡＸはレイヤードストリーム（探索空間をストリームの入れ子構造で表現したもの）を用いている前の例の lovesと Mary　の間のストリームは（処理が完了した後では）次のような構造になっている.

```
[(np ->vp)* [(vp -> s)* [(s->#->#)* [] ] ] ]
```

これに，表現と型の表示（表現: 型）を明示的に付加すると次のようになる.

```
[  loves :(np->vp)* [John:(vp->s)*  [ {} :(s->#->#)*  [] ] ] ]
```

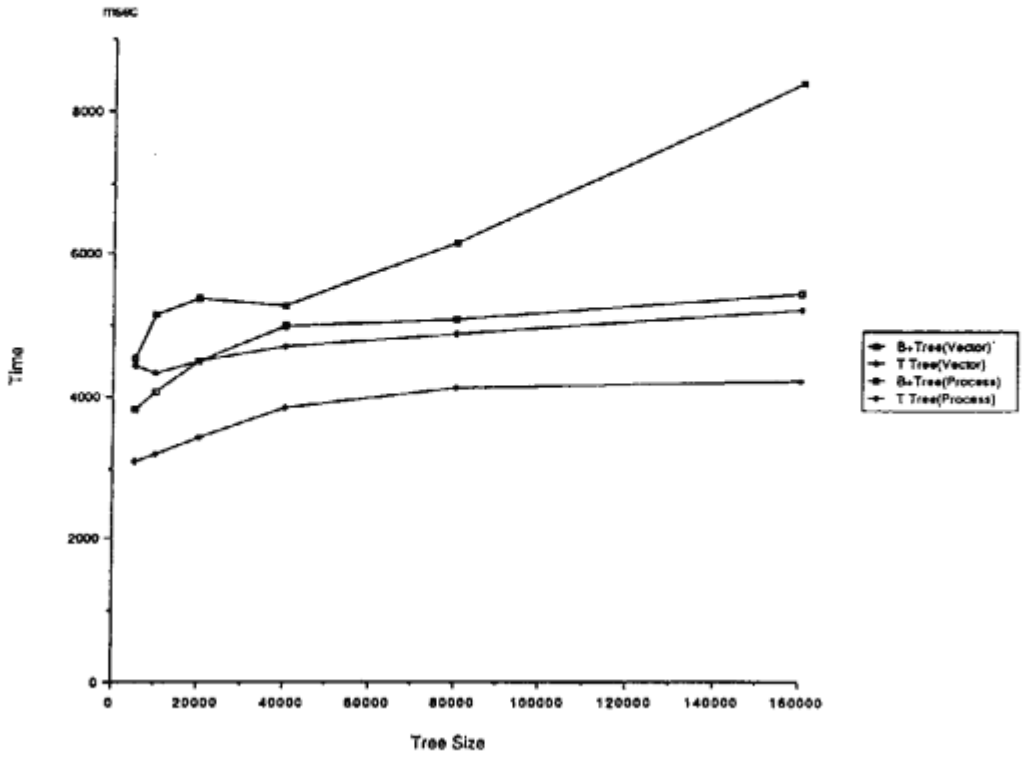
loves と Mary の間のストリームの第1 の階層は，loves で終わる表現列の対する型のバリエーションを表示している．一方，ストリームの階層は，一つ内側の階層は，その外側の階層に対応している表現列の直前でおわる表現列に対する型を表示している．（例えば，loves　に付随しているストリームの中に John の型が現れる）また，レイヤードストリームは，表現列の区切りごとに与えられた型のバリエーションを '*' という演算の分配法則を用いて共有できる部分を共有させている．'*' という演算は次のようなものである.

```
(em+1,...,en :(T1->...->Ti))* (e1 ... em  :(Ti->Ti+1,...->Tj) )
  = e1 ... em1 em+1...en  :(T1->...->Ti-1 -> Ti+1->...->Tj)
```

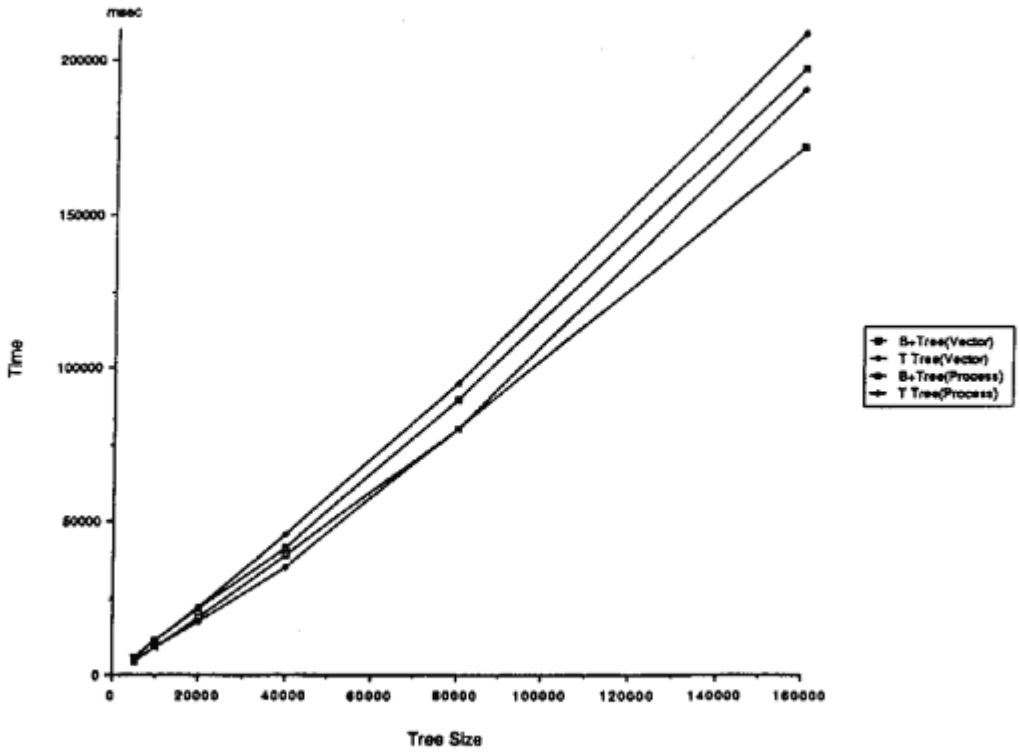
ここで型を命題と解釈すると，この演算は次のような推論規則になる

$$\frac{T1 \supset \ldots \supset Ti \qquad Ti \supset Ti+1, \ldots \supset Tj}{T1 \supset \ldots \supset Ti-1 \supset Ti+1 \supset \ldots \supset Tj}$$

グラフ 3. 削除処理



グラフ 4. 範囲検索処理

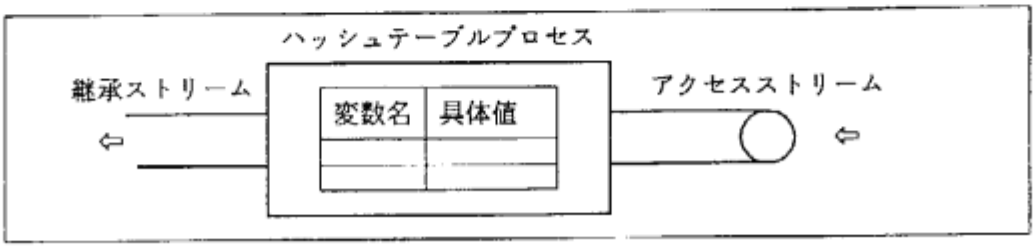


うことによって制約式が解かれる．

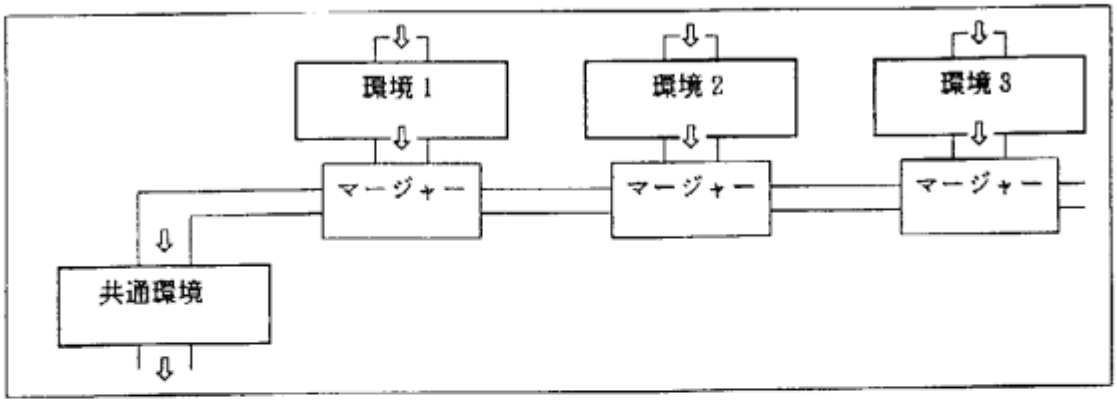
（3） 多重環境の実現方法

現在の実装法では，変数の環境は変数名をキーとするハッシュ表を内部に持つプロセスとして実現している．変数の環境の参照及び更新は，このプロセスのアクセスのためのストリームを介して行う．

ただし，このプロセスは継承のためのストリームも持っていて，自分のテーブルに該当するキーが存在しないときには，継承ストリームに自分が受け取ったアクセス要求を参照の要求に変換して渡す．参照や更新した結果は結局は自分のテーブルに追加する．したがって，継承するストリームの先にあるプロセスのテーブルは参照するだけで更新することはなく，また，一旦参照すると自分の場所に情報を追加するので，2度目からは継承先を探索することはない．



多重環境は，和型によって探索が分岐するとき，および入力ストリームから新たなデータを読み込んだ時に新たなハッシュ表プロセスを生成し，この新しいプロセスがマージャを介して古い環境を継承することによって実現している．この方法は，環境を全て複写するのではなく，環境の差分のみを新規に生成することになっている．



（4）環境の複写

非終端記号までリデュースされた時に，完成した非終端記号のインスタンスとしてその環境が持っている情報の内容は，次にその非終端記号を受け取ったプロセスで，文法との適合検査を行うときに参照され，適合するときにはその環境にマージされなければならない．

このために，非終端記号を構成するために作られた環境は，非終端記号とともに通信データとして送られる．このとき，非終端記号を受け取るプロセスでは，受け取った非終端記号毎に新しい環境を作成し，その環境を複写する必要がある．

全体としては，範囲検索以外ではベクタ版がプロセス版の二割性ど速く，プロセスにした時の遅さが現れた．範囲検索は四つの間にあまり差はない．ここでは1PEで測定したが，プロセス版が逐次的な処理なのに対し，ベクタ版は並列性がかなりあるので，PIMのクラスタで実行する場合，ベクタ版が有利となると予想される．T木とB^+木との比較では[Lehman 86]の結果ほど大きな差にはならなかった．単一代入を基本とするKL1と副作用を許すC言語の違いが原因であると予想している．

4 まとめ

並列データベース管理システムの基本要素として，ローカルDBMSでの処理で基本となる，集合操作，レコード操作，索引操作について試作をおこないマルチPSI上で計測をおこなった．ただし，試作ではローカルDBMS内の主記憶データベース機能に限定し，共有メモリを持つクラスタで実行されることを前提として設計をしている．

評価結果の簡単なまとめをおこなう．

- 集合操作
  ローカルDBMSでの処理でも並列性はかなり存在し，しかもPIMのクラスタでの実行で，それが高速化につながることが明らかになった．
- レコード操作
  ベクタを利用したレコードからは，その記述性と処理速度ともに満足のいく結果がでたが，ストリングを利用したレコードからは，ストリング処理能力の改善が必要であるという結果がでた．
- 索引処理
  並列処理に向いた範囲検索の方法が見つかった．ただし，これは単一参照を守った場合の結果であり，一般的に言えるのかどうかは疑問である．

その他に，ここではデータとして出さなかったが，KL1で記述し1PEで実行しても，ESPで記述したKappa-IIと同等以上の速度が期待できることが分かった．

データベース管理システムをPIM/PIMOSで作るにあたってネックになりそうなのは，レコード操作で明らかになったストリング処理能力である．ストリング処理能力は二次記憶データベースで非常に重要であるため，ストリング処理のファームウエア化が望まれる．詳しくは，ストリングの部分構造へのアクセス(substring,copy_string_elements,move_string_elements)や，ストリング内の要素検索(search_character)，ストリングの比較(less_than,not_less_thanまたはcompare_string)などである．

また，考えがまとまらなかったので本稿では触れなかったが，並列データベース管理システムなどの分散処理指向のシステムで分散トランザクションやデータの複製をおこなうことを考えると，PIM/PIMOSでどんな種類の障害が発生する可能性があって，それに対しどんな対応をPIM/PIMOSが行ない，それをPIM/PIMOS上のシステムがどこまで知ることができるのかということも重要である．

参考文献

[Lehman 86] Tobin J. Lehman and Michael J. Carey, A Study of Index Structures for Main Memory Database Management Systems, Twelfth international conference on very large data bases, 294-303(1986).

並列自然言語構文解析システム PAX の改良

佐藤 裕幸

三菱電機 (株) 情報電子研究所

〒247 鎌倉市大船 5-1-1

Tel: 0467-46-3665, E-Mail: hiroyuki@isl.melco.co.jp

概要

近年、自然言語処理、特に自然言語構文解析の並列化が活発化しており、新世代コンピュータ技術開発機構 (ICOT) でも並列自然言語構文解析システム PAX (Parallel Analizer for syntaX and semantiCS) の研究が行なわれている。

これまでの PAX の研究で、構文解析の処理の過程でプロセサ間の通信量が非常に多いため、複数のプロセサを用いても、処理速度があまり向上しないということが報告されている。そこで、今回、PAX のプロセサ間通信を減少させる改良を行なったので、改良方式及び測定結果を報告する。

PAX のプログラムは、通信量が減少することにより実行時間も減少するという性質を持っている。マルチ PSI 上での測定の結果、改良によって、通信量は $\frac{1}{6}$ に減少し、単一プロセサ上では最高 3 倍強の速度向上が得られた。また、今回の改良は、逐次型の自然言語構文解析システム SAX にも適応できる。

なお、この改良方式の元々のアイデアは、ICOT の淵氏によって考案されたものであり、筆者が実現 / 評価し、その効果を実証した。

1 はじめに

自然言語処理は多大な計算量を必要とし、処理速度の向上が望まれているプログラムの 1 つである。そのため、近年、自然言語処理、特に比較的アルゴリズムが確立している自然言語構文解析の並列化が活発化しており、新世代コンピュータ技術開発機構 (ICOT) でも並列自然言語構文解析システム PAX (Parallel Analizer for syntaX and semantiCS) の研究が行なわれている。

PAX は、逐次型の自然言語構文解析システム SAX (Sequential Analizer for syntaX and semantiCS) [松本 86] の並列版である。SAX の構文解析アルゴリズムは、並列性を持っているだけではなく、バックトラック及び副作用を用いないため、逐次計算機上でも高速に処理できる。

しかし、これまでの PAX の研究で、構文解析の処理の過程でプロセサ間の通信量が非常に多いため、プロセサ間通信のコストが高いマルチ PSI のような疎結合並列計算機上では、複数のプロセサを用いても、処理速度があまり向上しないということが報告されている [寿崎 89]。

そこで、今回、PAX のプロセサ間通信を減少させる改良を行い、オリジナルの PAX との比較した。

まず最初に、オリジナル PAX の構文解析方法と負荷分散方式について解説する。次にそれをどのように改良したかを記述した後で、この改良方式のオリジナル PAX への適応方法について解説する。そして最後に、今回改良した PAX をいくつかの項目についてマルチ PSI 上で測定したので、その結果及びオリジナル PAX との比較を報告する。また、付録として、簡単な文法に対応する PAX のプログラムを添付しておく。

2 オリジナル PAX

2.1 オリジナル PAX での解析方法

PAX は、逐次型の自然言語構文解析システム SAX の並列化版である。SAX のアルゴリズムは、基本的には左隅構文解析法であり、あるまとまった非終端記号 (または、終端記号) が構成されると、それを基にして、より大きな構文木を構成しようとするが、その時、その (非) 終端記号を左隅の要素とする新しい解析木を作ろうとする。つまり、この非終端記号を右辺の先頭要素として持つ文法規則を選び、その文法規則を完成させようとする。このような文法規則の右辺の 2 番目以降の要素は、新しい (非) 終端記号が得られるたびに、埋め合わされ、完全な解析木へと作り上げられていく。

このような処理を (並列) 論理型言語で実現するために、文法規則の右辺に、以下のようにそれぞれの場所を示す識別子を設ける。

```
(1)  s    → np, id1, vp.   (2)  np → a, id2, n.
(3)  np   →  prp, id3, n.
(4)  np   →  np, id4, conj, id5, np.
(5)  vp   → v, id6, a.     (6)  vp → v, id7, av.
(7)  vp   → v, id8, np.    (8)  a  → [failing].
(9)  a    → [hard].        (10) av → [hard].
(11) n    → [students].    (12) v  → [looked].
(13) conj→ [and].          (14) prp→ [failing].
```

id1 から id8 までが識別子であり、文法規則内の特定の位置を表している。例えば、id1 は (1) の文法規則の np までの解析が終ったことを表している。PAX では、解析された終端記号および非終端記号がすべて KL1 のプロセス

```
<ベクタ版B+木> ::= "{" "}" | <ノード> | <リーフ>
<ノード> ::= "{" <キー数> "," <ノードリスト> "}"
<ノードリスト> ::= "[" <ベクタ版B+木> "," <キー>  "," <ベクタ版B+木>
                    { "," <キー>  "," <ベクタ版B+木> } "]"
<リーフ> ::= "leaf(" "{" <キー> "," <データ> "}" { "," "{" <キー> "," <データ> "}" } ")"
<キー> ::= INTEGER
```

T木のデータ構造

T木は主記憶での木構造に適したものとして，二進木であるAVL木の変形として考案された．AVL木との違いは，ノードに複数個のデータ格納を許し，リバランスによるデータの移動を減らすことができるとしている．我々は，木に対する順次アクセスの性能も重視したため，B^+木と同じ理由から，ノードをプロセスにし順次アクセス用のリンクを持つものと，ベクタで表現し順次アクセス用のリンクを持たないものの二種類を試作した．

(1) リーフノード プロセス版T木

リーフノードをプロセスとしたT木のデータ構造を簡略化して示す．これを今後，プロセス版T木と呼ぶ．

```
<プロセス版T木> ::= "{" "}" | "{" <左プロセス版T木> "," <制御情報> ","
      <右プロセス版T木> "," "{" <最小値> "," <最大値> "," <データ保持プロセス> "}" "}"
<制御情報> ::= "leftdown" | "balanced" | "rightdown"
<データ保持プロセス> ::= "{" <左右データ保持プロセス> ","
           "{" <キー> "," <データ> "}" { "," "{" <キー> "," <データ> "}" } "}"
<左右データ保持プロセス> ::= "{" <左データ保持プロセス> ","  <右データ保持プロセス> "}"
<キー> ::= INTEGER
```

(2) リーフノード ベクタ版T木

リーフノードをベクタとしたT木のデータ構造を簡略化して示す．これを今後，ベクタ版T木と呼ぶ．

```
<ベクタ版T木> ::= "{" "}" | "{" <左ベクタ版T木> "," <制御情報> ","
           <右ベクタ版T木> "," "{" <最小値> "," <最大値> "," <データリスト> "}" "}"
<制御情報> ::= "leftdown" | "balanced" | "rightdown"
<データリスト> ::= "{" "{" <キー> "," <データ> "}" { "," "{" <キー> "," <データ> "}" } "}"
<キー> ::= INTEGER
```

3.3.2 演算

木に対する演算として，検索 / 追加 / 削除および範囲検索を実現した．

- 検索処理　　キーを指定しそれに対応するデータを木から取り出す処理
- 追加処理　　キーとデータのペアを木に対して追加する処理
- 削除処理　　キーを指定しそれを木から削除する処理
- 範囲検索　　ある範囲に存在するキーとデータを順に取り出す処理

処理方式に特徴があるのは，範囲検索に対する処理である．以下にその処理概要を示す．

- リーフノードがプロセスで実現されている木
  リーフノード間にパスがあるため，まず最小値を検索しそれ以降はリーフノード間のパスに沿って最大値以上のものが現れるまでを検索結果として返す．比較演算の回数は，最小値にたどり着くまでと，最大値にたどり着くまでのに通過するリーフノード中の最大値との比較と，最大値を含むノード内で最大値にたどり着くまでの比較になる．

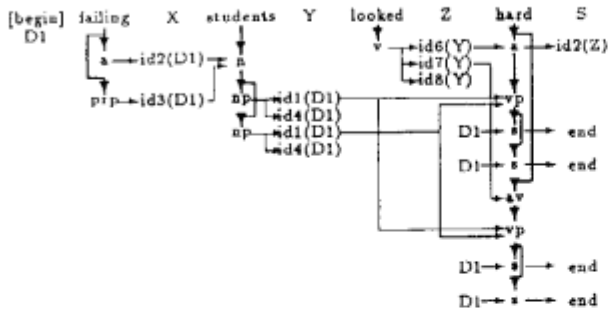


図 2: PAX による文の解析例

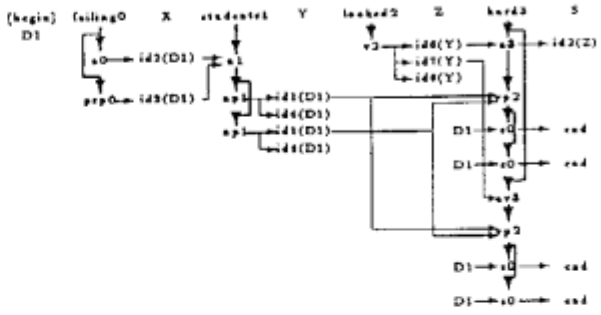


図 3: 負荷分散を行った文の解析例

るので、それぞれの組み合わせで 4 つの解となる。

入力文の単語から始まって、構文解析木を作り上げていく過程が、プロセス間でデータ (識別子) を通信することで行われているのがよくわかる。このように PAX では、最初に単語間に張られた共有変数をストリームとして階層的に使用することで、それぞれの (非) 終端記号に相当するプロセス間で通信しながら解析を進めていく。このようにストリームを階層的に使用する方式を、レイヤード・ストリーム手法 [Okumura87] と呼んでいる。

2.3 負荷分散方式

PAX のアルゴリズムには、次の 2 点に並列性があると考えられる。

and 並列: 独立した非終端記号を生成する過程に並列性がある。

or 並列: 1 つの非終端記号を生成する際に、複数の可能性がある場合、それら複数の可能性同士に並列性がある。

上で述べた 2 つの並列性はそれぞれ次のような特徴を持つ。

and 並列: これは、最初に起動する各単語に相当するプロセスを並列に実行することに相当する。従って、並列度は実行開始時が一番大きく最大 (単語の数) であり、解析が進むにつれて並列度が下がると考えてよい。

or 並列: これは、文法規則に曖昧性があれば、それらを並列に実行することに相当する。従って、並列度は文法規則の曖昧性に依存するが、and 並列より大きいと考えてよい。

次に、プロセサ間通信の観点から負荷分散を考えてみる。PAX のプロセスはタイプ 2 が多く、左の兄弟からのストリームを読んで、右の兄弟へのストリームへデータを送るという動きをする。そのため、左右の兄弟が別プロセサで実行されているとすると、多対多という莫大なプロセサ間通信が起き、並列実行の効果が現れなくなってしまう。従って、上記の and 並列 / or 並列 どちらを取っても、プロセサ間通信が多くなり、並列実行の効果は期待できない。

そこで、プロセサ間通信を抑える負荷分散方式が考えられた。これは、単語間を結ぶストリームにプロセサを割り付け、最初に起動する単語に相当するプロセスをそのプロセスの左のストリームと同じプロセサ上で起動する。そして、各プロセスが、ストリームからのデータを読み込もうとすると、そのストリームに割り付けられたプロセスに移動する方式である。この負荷分散方式では、同じストリームを読むプロセスが 1 つのプロセサに集まるので、多対 1 のプロセサ間通信しか起きない。また、この負荷分散方式では、データの代わりにプロセスをプロセサ間で移動することになるが、データ移動とプロセス移動のコストの比率より 1 つのプロセスが多くのデータを読み込む場合に、この負荷分散方式の長所が現れると言える。

この負荷分散方式を採用するためにタイプ 2 の述語に以下の節を追加する。

```
(f)     node([PE|In], Out) :- integer(PE) |
                    node(In, Out)@processor(PE).
```

また、前に述べた例文を解析する場合は、以下のように各述語を起動する。

```
(g)  ?- failing([0,begin], X)@processor(0),
        merge(X, X1)@processor(1),
        students([1|X1], Y)@processor(1),
        merge(Y, Y1)@processor(2),
        looked([2|Y1], Z)@processor(2),
        merge(Z, Z1)@processor(3),
        hard([3|Z1], S)@processor(3),
        merge(S, S1)@processor(4),
        fin([4|S1])@processor(4).
```

この例文の解析で、各プロセスがどのプロセサで実行されるかを示したのが、図 3 である。プロセス名の横に示した数字が、実行されるプロセサを示している。

3 改良版 PAX

3.1 改良の概要

以上のようにオリジナルの PAX では、プロセス間で通信が多いという特徴を持っている。従って、プロセス間の通信量 (データ量) を減少させる改良が必要となる。

レコードの Rid の集まりがまず求められる．これを集合と呼ぶ．複雑な条件の検索は，単純な索引を利用した検索とその結果である集合間の演算 (Union, Intersection, Difference) により求められる．レコードの構造に木構造を許す非正規関係では Rid は複雑な構造になる場合があるが，ここではレコード全体の選択情報を示す Rid(整数値) のみからなる集合間の演算に限定した評価をおこなった．

3.1.1 データ構造

整数値のみからなる集合を，あらかじめ決められた値の範囲によって分割し，それぞれ独立して演算を行なえるような構造とした．

```
<Set> ::= "[" <Partition> { "," <Partition> } "]"
<Partition> ::= "{" <Quantity> "," "[" <Rid> { "," <Rid> } "]" "}"
<Quantity> ::= INTEGER
<Rid> ::= INTEGER | "{" <FromRid> "," <ToRid> "}"
<FromRid> ::= INTEGER
<ToRid> ::= INTEGER
```

ただし，次の制限がある．

- Rid は前から昇順に並んでいる
- Partition に入る Rid の範囲は静的に決められている

3.1.2 演算

Union, Intersection, Difference の各演算を，マージ アルゴリズムを使って求める．

3.1.3 計測と評価

この集合演算では，Partition 間に渡る演算はなく Partition の数だけの並列度が見込まれる．この演算は共有メモリを仮定しているので，同じ範囲をとる Partition はマルチ PSI の同じプロセッサ上に割り当てられるようにデータを配置して計測した．

10 万の集合要素を含む集合どうしの集合演算 (Union) を 64 台版マルチ PSI で計測した時の台数効果を表 1 に示す．集合要素数と PE 台数変えて測定した結果，一つの PE に 7,000 から 8,000 の集合要素を割り当てた時最も台数効果が現れることがわかった．また，分割数は PE 台数と等しいとき，もっとも台数効果があり，PE 台数の 3 倍程度までは著しい低下は見られない．また，演算の処理速度は，結果として求まる集合の要素数はあまり関係せず，比較の回数に依存する．

これらから，実際の集合の演算には全レコード件数を 7,000 で割った数だけの PE があれば十分であり，分割数は，選択される Rid のばらつきを考えて，集合演算に使用する PE の数の 2 倍ぐらいにすれば良いと想像できる．

表 1 集合演算の台数効果

PE の数	2	4	8	16	32	64
奇数集合同士	1.97	3.86	7.61	14.30	24.46	37.93
乱数集合同士	1.99	3.92	7.63	14.80	27.3	—

3.2 レコード操作

主記憶構造に適したレコードの構造を調べることを目的に，評価をおこなった．ただし，レコード操作は軽い処理であり，関係代数演算にはレコード操作を単位とする並列性がレコード数だけ存在するので，レコード操作に含まれる並列性を引き出すことを第一に考えてはいない．レコード操作については，KL1 で記述することの面白みは特にないため，簡単な記述にとどめる．

評価では，二次記憶データベースとの変換時の手間が少ないレコードのストリング表現と，二次記憶データベースとの変換時の手間が多いがその他のアクセスは効率的と予想されるレコードの構造体表現とを実装し，レコード操作の基本操作 (生成 / 読み / 削除 / 追加) についてマルチ PSI で 1 台の PE に割

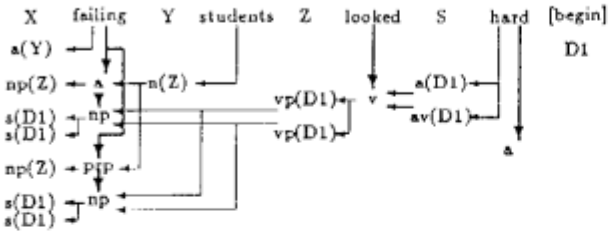


図 5: 改良版 PAX による文の解析例

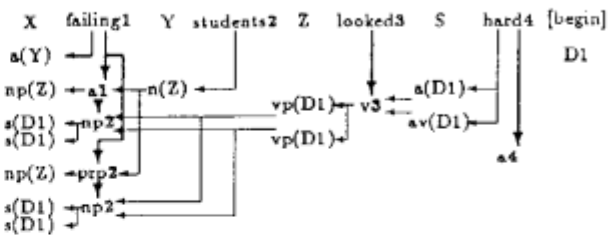


図 6: 改良版 PAX による文の解析例 (負荷分散付き)

ら流れてくる conj(Y) を (h) の第 2 節によって受け付けて np_conj を呼び出し、それが Y から流れてくる np を (i) の最初の節によって受け付けて、新しい np を呼出している様子が示されている。この時、変数 Y が、conj により、変数 Z が np により渡され、最終的に Z が新しい np の入力引数として渡されている。非終端記号が入力引数を運んでいたのは、このように新たな非終端記号が生まれた時に、正しい入力データを渡すためである。

3.3 改良版 PAX での解析例

"Failing students looked hard." という文を構文解析する例を示す。この文を解析するには、以下のようなゴールを呼び出す。

```
(l)  ?- fin(X1), merge(X, X1),
        failing(Y1, X), merge(Y, Y1)
        students(Z1, Y), merge(Z, Z1),
        looked(S1, Z), merge(S, S1),
        hard([begin], S).
```

最終的に求めたいものは、文すなわち s であるから、解析が成功するのは、与えられた文の単語を全て用いて s が構成された時である。これは、ストリーム X から s([begin]) が送られた時であり、(l) の fin という述語は、この s([begin]) を理解するための述語であり、文の解析が成功するのは、fin が s([begin]) を受け付ける時、かつその時に限る。

図 5 に、PAX による構文解析の様子を示す。図中の太い矢印は、プロセスの起動 (述語の呼び出し) に対応し、細い矢印は、(非) 終端記号の通信に対応する。また、X, Y, Z, S は、単語間に置かれた共有変数で、各プロセス間で通信するために、ストリームとして使われている。

改良版の PAX も、オリジナルの PAX と同様にストリームを階層的に使用してそれぞれのプロセスが通信を行う。

3.4 負荷分散方式

改良版の負荷分散方式は、当面オリジナルの負荷分散方式と同じにする。つまり、単語間を結ぶストリームにプロセサを割り付け、最初に起動する単語に相当するプロセスをそのプロセスの右のストリームと同じプロセサ上で起動する。そして、各プロセスが、ストリームからのデータを読み込もうとすると、そのストリームに割り付けられたプロセスに移動する方式である。

この負荷分散方式を採用するために各述語に以下の節を追加する。

```
(m)    inter_node([PE|In], Out) :- integer(PE) |
                inter_node(In, Out)@processor(PE).
```

また、前に述べた例文を解析する場合は、以下のように各述語を起動する。

```
(n)  ?- fin([0|X1])@processor(0),
        merge(X, X1)@processor(0),
        failing([1|Y1], X)@processor(1),
        merge(Y, Y1)@processor(1),
        students([2|Z1], Y)@processor(2),
        merge(Z, Z1)@processor(2),
        looked([3|S1], Z)@processor(3),
        merge(S, S1)@processor(3),
        hard([begin], S)@processor(4).
```

この例文の解析で、各プロセスがどのプロセサで実行されるかを示したのが、図 6 である。プロセス名の横に示した数字が、実行されるプロセサを示している。

改良版の PAX では、プロセス間の通信量がオリジナルの PAX に比べて少ないので、別の負荷分散方式も考えられるかも知れない。

3.5 オリジナル PAX への改良方式の適応

今回の改良方式をオリジナル PAX に適応する方法を考えてみる。

今回の改良では、識別子の代わりに (非) 終端記号をデータとして受渡しすることにより、プロセス間で通信されるデータ量を抑えているが、オリジナル PAX でも、識別子の数 (種類) を減らすことで、プロセス間の通信量を抑えることがある程度可能である。

オリジナル PAX での識別子は、文法規則内の特定の位置を示していた。しかし、識別子の本来の意味は、文法規則の左隅からどういう (非) 終端記号列を解析できたかを示すことである。従って、文法規則の左隅から同じ (非) 終端記号列が続く限り、同じ識別子を割り付けることが可能である。例えば、文法規則 (1) ～ (14) は、以下のように識別子を重複して割り付けることができる。

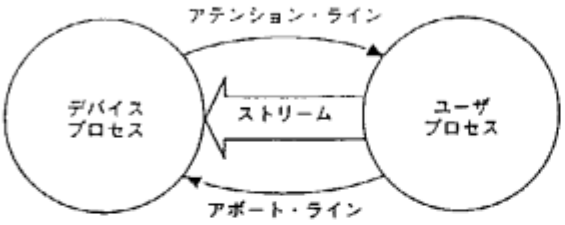


図1：ストリームとライン

デバイス・プロセスは，アボート状態と正常状態があり，デバイスが生成された時点ではアボート状態である．アボート状態では，すべてのIO要求メッセージは無効になり，メッセージのStatus変数をabortedにして処理を完了するため，デバイス・プロセス生成後の最初にresetメッセージをデバイスに送り正常状態にする必要がある．

このようにアボート状態でのメッセージを全て無効にするのは，ストリームにバッファリングされたメッセージをクリアする意味を持つ．また，アボート・ラインが具体化されるとデバイス・プロセスはアボート状態になる．アボート・ラインが具体化された時点で処理中のメッセージがあれば，それは中断されるものとする．ラインが張られていない場合のアテンションやアボートの要求は無視される．

以下に，ウィンドウを例にとり，ユーザからの割り込みと，それによるIO処理の中断を順を追って説明する．

1. ユーザ割り込み

   ウィンドウが割り込みのキー入力を検出すると，その旨をアテンション・ラインを具体化することで，ユーザ・プロセスに通知する．ユーザ側では，アテンション待ちプロセスがアテンション・ラインの具体化を待っている．

2. IOの中止

   ユーザ・プロセスが割り込みを受け取り，その結果としてその時点までのIO処理を中断する必要を生じた場合は，アボート・ラインを具体化する．ウィンドウ・プロセス側では，アボート・ラインの具体化によりその時点でのIO処理を中断し，アボート状態になる．以後，resetメッセージが到着するまでのIO要求は全て無視される．

3. IOの再開

   IOを中断した後で再度IOを再開するには，ユーザ・プロセスからresetメッセージをデバイスに送出する．

このようにresetメッセージをストリームに流すことで，それぞれのアボート・ライン，アテンション・ラインがストリーム中を流れるIO処理メッセージに及ぼす影響の範囲(スコープ)を明確にすることができる．つまり，それぞれのラインのスコープは，そのラインを張るためのresetメッセージから次のresetメッセージまでとなる．

もし，割り込みの結果IO処理を中断しないのであれば，単にアテンション・ラインを張り直すためにresetメッセージをデバイスに送ればよい．

PIMOSでは，アボートによる処理の中断後，アボートされた処理を再開できるようになっている．この機構は，デバイス・プロセスがアボートされたメッセージを内部のバッファに保持しておき，resendメッセージを受け取ることで保持されたメッセージを再度処理する．

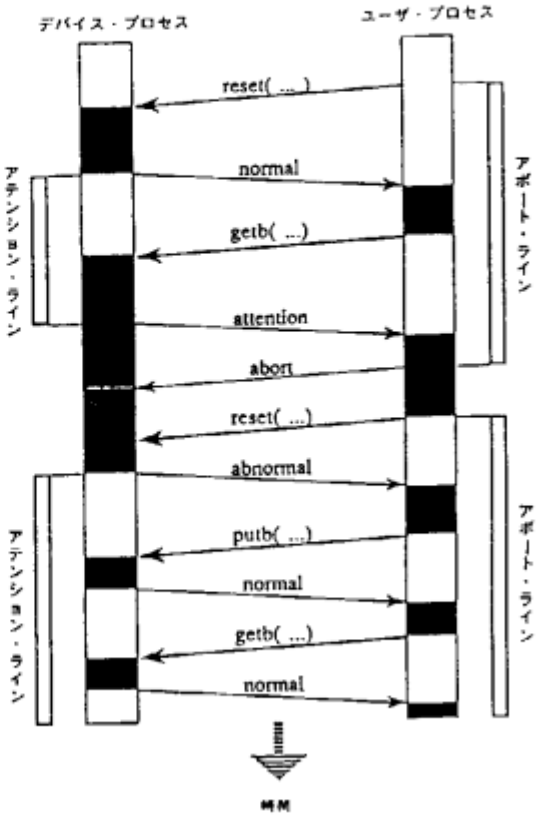


図2：ラインによる非同期処理の例

5 おわりに

1本のストリームにラインと呼ばれる機構を用いることで，ストリームを逆行するアテンションとメッセージを飛び越えた処理の中断を実現する方式について述べた．複数のストリームを用いる場合に比べ，機構が比較的単純でありながら，目的を果たすことが可能になった．この方式は，既にPIMOSにおけるデバイスを制御するストリームに実装されている．

この機構は，デバイスに限らず非同期通信が必要なプロセス間通信に一般的に応用可能である．

6 謝辞

本研究に関して有益な助言を頂いたICOT第2研究室の近山室長をはじめとするPIMOSの開発メンバーに深く感謝する．

7 参考文献

[1] 佐藤他：「並列論理型OS-PIMOS(1)」，第35回情処全国大会，4D-3，1987-9

[2] 堀他：「PIMOSの入出力管理」，第36回情処全国大会，2D-2，1988-3

表 3: 各ストリームのデータ量とプロセス数

ストリーム番号	(i)			(ii)		
	D	P	M	D	P	M
0	1	9	9	1	9	9
1	28	11	308	6	11	66
2	64	7	448	15	7	105
3	16	1	16	4	1	4
4	76	1	76	18	1	18
計	185	29	857	44	29	202

ストリーム番号	(iii)			(iv)		
	D	P	M	D	P	M
0	1	9	9	9	1	9
1	4	11	44	11	4	44
2	11	7	77	7	8	56
3	4	1	4	1	2	2
4	16	1	16	1	12	12
計	36	29	150	29	27	123

D: データ数
P: プロセス数
M: データ数 × プロセス数

表 4: 単一プロセサでのリダクション数

文番号	(i)	(ii)	(iii)	(iv)	(i)/(ii)	(i)/(iii)	(i)/(iv)
1	1,669	969	678	619	1.72	2.46	2.52
2	1,451	810	572	458	1.79	2.54	3.17
3	7,388	3,577	2,919	2,600	2.07	2.53	2.84
4	4,405	2,408	1,736	1,603	1.83	2.54	2.75
5	9,547	4,856	4,348	3,784	1.97	2.20	2.52
6	15,307	7,516	6,135	5,501	2.04	2.50	2.78
7	47,340	18,740	16,682	14,538	2.53	2.84	3.26
8	16,495	8,709	6,810	6,112	1.89	2.42	2.70
9	17,398	9,106	7,834	6,745	1.91	2.22	2.58
10	1,949,802	686,495	611,624	512,706	2.84	3.18	3.80
11	2,844,476	915,663	825,516	736,253	3.11	3.45	3.86

表 5: 単一プロセサでの実行時間 (msec)

文番号	(i)	(ii)	(iii)	(iv)	(i)/(ii)	(i)/(iii)	(i)/(iv)
1	71	61	64	53	1.16	1.11	1.34
2	69	56	52	52	1.23	1.33	1.33
3	163	115	90	88	1.42	1.81	1.85
4	117	87	83	71	1.34	1.41	1.65
5	202	119	113	103	1.70	1.79	1.96
6	302	160	148	131	1.89	2.04	2.31
7	811	344	304	270	2.36	2.67	3.00
8	307	175	149	151	1.75	2.06	2.03
9	332	190	167	150	1.75	1.99	2.21
10	29,068	10,522	9,363	8,634	2.76	3.10	3.37
11	43,104	14,000	12,764	11,955	3.08	3.38	3.61

表 6: 複数プロセサでの実行時間 (msec)

PE 台数	(i')	(i)	(ii)	(iii)	(iv)
1	43,104	43,104	14,000	12,764	11,955
2	110,130	46,996	21,420	20,306	21,003
4	79,078	34,120	16,507	16,098	15,922
8	61,521	27,646	12,228	12,749	12,650
16	72,576	21,131	8,947	8,227	9,913
32	69,251	13,167	6,386	6,151	8,133
$\frac{1}{32}$	0.62	3.27	2.19	2.08	1.47

も、同様の順に減少しており、改良版 PAX では約 $\frac{1}{7}$ になっている。従って、実行速度の向上がかなり期待できる。

改良版 PAX はオリジナル PAX のプロセスとストリームを逆転させているため、各ストリームでの (i), (ii), (iii) のプロセス数と (iv) のデータ数が等しくなっている。一方、オリジナル PAX のデータ数と改良版 PAX のプロセス数は異なっており、前者の方がかなり多くなっている。これは、オリジナル PAX では、ある (非) 終端記号まで解析されると可能な右側の識別子をすべて送るのに対して、改良版 PAX では、二つのとなり合う (非) 終端記号が両方とも解析されないとその間に相当するプロセスが起動されないため、改良版 PAX のプロセス数の方が少なくなっていると考えられる。。

4.3 単一プロセサでの測定

単一プロセサ上でのリダクション数及び実行時間を表 4, 5 に示す。リダクション数の減少及び処理速度の向上とも、(i)< (ii)< (iii)< (iv) の順になっている。特に改良版 PAX (iv) は、リダクション数で、$\frac{1}{2.5} \sim \frac{1}{3}$ になり、実行時間で、$\frac{1}{1.3} \sim \frac{1}{3.6}$ になっている。

前節でのデータ数 × プロセス数の減少比率より、リダクション数や実行時間の減少比率が小さいのは、データ数 × プロセス数には補強項の実行が含まれておらず、実際の解析では補強項の実行比率が大きいからであろう。

4.4 複数プロセサでの測定

例文 11 の複数プロセサでの実行時間を表 6 に示す。オリジナル PAX, 改良版 PAX とも、各ストリーム毎にプロセサを割り付けることで、プロセサ間の通信量を抑える負荷分散方式を採用している。表中の (i') は、この負荷分散

謝辞

本研究に関して有益な助言を頂いたICOT第2研究室長の近山隆博士，三菱電機の佐藤裕幸氏および沖電気工業の宮崎敏彦氏に深く感謝する．

参考文献

[1] Goto, A. et al.: Toward a High performance Parallel Inference Machine -The intermediate State Plan of PIM -, Technical Report TR-201, ICOT (1986).

[2] Taki, K.: The Parallel Software Research and Development Tool: Multi-Psi System, Technical Report TR-237, ICOT (1986).

[3] 松尾他: PIMOS のタスク管理方式 - タスク終了時の資源解放 -, 第36回情報処理学会全国大会論文集 3D-4, pp.293-294(1988).

[4] 藤瀬他: PIMOS の階層的資源管理, 第37回情報処理学会全国大会論文集 5P-3, pp.251-252 (1988).

[5] Chikayama,T. et al.: Overview of the Parallel Inference Machine Operating System (PIMOS), Proc.of FGCS'88, Vol.1, pp.230-251(1988).

[6] 佐藤他: PIMOS の資源管理方式, 情報処理学会論文誌 Vol.30 No.12, pp.1646-1655 (1989).

```
t_students(In, Out0, Tree) :- true |
    n_0(In, Out0, n_0(Tree)).

t_looked(In, Out0, Tree) :- true |
    v_0(In, Out0, v_0(Tree)).

t_and(In, Out0, Tree) :- true |
    conj_0(In, Out0, conj_0(Tree)).

s_0([begin(In1,Tree1)|In], Out0, Tree) :- true |
    Out0 = [end(In1,(Tree1,Tree))|Out1],
    s_0(In, Out1, Tree).
s_0([PE|In], Out0, Tree) :- true |
    s_0(In, Out0, Tree)@processor(PE).
s_0([], Out0, Tree) :- true | Out0 = [].
otherwise.
s_0([_|In], Out0, Tree) :- true |
    s_0(In, Out0, Tree).

np_0(In, Out0, Tree) :- true |
    Out1 = [{0,In,Tree},{1,In,Tree}],
    np_0_zz(In, Out2, Tree),
    Out0 = {Out1,Out2}.

np_0_zz([{2,In1,Tree1}|In], Out0, Tree) :- true |
    np_0(In1, Out1, np_0((Tree1,Tree))),
    Out0 = {Out1,Out2},
    np_0_zz(In, Out2, Tree).
np_0_zz([{7,In1,Tree1}|In], Out0, Tree) :- true |
    vp_0(In1, Out1, vp_0((Tree1,Tree))),
    Out0 = {Out1,Out2},
    np_0_zz(In, Out2, Tree).
np_0_zz([PE|In], Out0, Tree) :- integer(PE) |
    np_0_zz(In, Out0, Tree)@processor(PE).
np_0_zz([], Out0, Tree) :- true | Out0 = [].
otherwise.
np_0_zz([_|In], Out0, Tree) :- true |
    np_0_zz(In, Out0, Tree).

vp_0([{0,In1,Tree1}|In], Out0, Tree) :- true |
    s_0(In1, Out1, s_0((Tree1,Tree))),
    Out0 = {Out1,Out2},
    vp_0(In, Out2, Tree).
vp_0([PE|In], Out0, Tree) :- integer(PE) |
    vp_0(In, Out0, Tree)@processor(PE).
vp_0([], Out0, Tree) :- true | Out0 = [].
otherwise.
vp_0([_|In], Out0, Tree) :- true |
    vp_0(In, Out0, Tree).

conj_0([{1,In1,Tree1}|In], Out0, Tree) :- true |
    Out1 = [{2,In1,(Tree1,Tree)}],
    Out0 = {Out1,Out2},
    conj_0(In, Out2, Tree).
conj_0([PE|In], Out0, Tree) :- integer(PE) |
    conj_0(In, Out0, Tree)@processor(PE).
conj_0([], Out0, Tree) :- true | Out0 = [].
otherwise.
conj_0([_|In], Out0, Tree) :- true |
    conj_0(In, Out0, Tree).

a_0(In, Out0, Tree) :- true |
    Out1 = [{3,In,Tree}],
    a_0_zz(In, Out2, Tree),
    Out0 = {Out1,Out2}.

a_0_zz([{5,In1,Tree1}|In], Out0, Tree) :- true |
    vp_0(In1, Out1, vp_0((Tree1,Tree))),
    Out0 = {Out1,Out2},
    a_0_zz(In, Out2, Tree).
a_0_zz([PE|In], Out0, Tree) :- integer(PE) |
    a_0_zz(In, Out0, Tree)@processor(PE).
a_0_zz([], Out0, Tree) :- true | Out0 = [].
otherwise.
a_0_zz([_|In], Out0, Tree) :- true |
    a_0_zz(In, Out0, Tree).

n_0([{3,In1,Tree1}|In], Out0, Tree) :- true |
    np_0(In1, Out1, np_0((Tree1,Tree))),
    Out0 = {Out1,Out2},
    n_0(In, Out2, Tree).
n_0([{4,In1,Tree1}|In], Out0, Tree) :- true |
    np_0(In1, Out1, np_0((Tree1,Tree))),
    Out0 = {Out1,Out2},
    n_0(In, Out2, Tree).
n_0([PE|In], Out0, Tree) :- integer(PE) |
    n_0(In, Out0, Tree)@processor(PE).
n_0([], Out0, Tree) :- true | Out0 = [].
otherwise.
n_0([_|In], Out0, Tree) :- true |
    n_0(In, Out0, Tree).

prp_0(In, Out0, Tree) :- true |
    Out0 = [{4,In,Tree}].

v_0(In, Out0, Tree) :- true |
    Out0 = [{5,In,Tree},{6,In,Tree},{7,In,Tree}].

av_0([{6,In1,Tree1}|In], Out0, Tree) :- true |
    vp_0(In1, Out1, vp_0((Tree1,Tree))),
    Out0 = {Out1,Out2},
    av_0(In, Out2, Tree).
av_0([PE|In], Out0, Tree) :- integer(PE) |
    av_0(In, Out0, Tree)@processor(PE).
av_0([], Out0, Tree) :- true | Out0 = [].
otherwise.
av_0([_|In], Out0, Tree) :- true |
    av_0(In, Out0, Tree).
```

6.3 改良版 PAX のサンプル・プログラム

```
:- module test.
:- public call/3.

call(failing, In, Out) :- true |
```

```
        t_failing(In, Out, failing).
    call(hard, In, Out) :- true |
        t_hard(In, Out, hard).
    call(students, In, Out) :- true |
        t_students(In, Out, students).
    call(looked, In, Out) :- true |
        t_looked(In, Out, looked).
    call(and, In, Out) :- true |
        t_and(In, Out, and).

    t_failing(In, Out0, Tree) :- true |
        a_0(In, Out1, a_0(Tree)),
        prp_0(In, Out2, prp_0(Tree)),
        Out0 = [a_0(In,a_0(Tree))|{Out1,Out2}].

    t_hard(In, Out0, Tree) :- true |
        Out0 = [a_0(In,a_0(Tree)),
                av_0(In,av_0(Tree))|Out1],
        a_0(In, Out1, a_0(Tree)).

    t_students(In, Out0, Tree) :- true |
        Out0 = [n_0(In,n_0(Tree))].

    t_looked(In, Out0, Tree) :- true |
        v_0(In, Out0, v_0(Tree)).

    t_and(In, Out0, Tree) :- true |
        Out0 = [conj_0(In,conj_0(Tree))].

    np_0([vp_0(In1,Tree1)|In], Out0, Tree) :- true |
        Out1 = [s_0(In1,s_0((Tree,Tree1)))],
        Out0 = {Out1,Outz},
        np_0(In, Outz, Tree).
    np_0([conj_0(In1,Tree1)|In], Out0, Tree) :- true |
        np_0__0(In1, Out1, (Tree,Tree1)),
        Out0 = {Out1,Outz},
        np_0(In, Outz, Tree).
    np_0([PE|In], Out0, Tree) :- integer(PE) |
        np_0(In, Out0, Tree)@processor(PE).
    np_0([], Out0, Tree) :- true | Out0 = [].
    otherwise.
    np_0([_|In], Out0, Tree) :- true |
        np_0(In, Out0, Tree).

    np_0__0([np_0(In1,Tree1)|In], Out0, Tree) :- true |
        Out1 = [np_0(In1,np_0((Tree,Tree1)))|Out2],
        np_0(In1, Out2, np_0((Tree,Tree1))),
        Out0 = {Out1,Outz},
        np_0__0(In, Outz, Tree).
    np_0__0([PE|In], Out0, Tree) :- integer(PE) |
        np_0__0(In, Out0, Tree)@processor(PE).
    np_0__0([], Out0, Tree) :- true | Out0 = [].
    otherwise.
    np_0__0([_|In], Out0, Tree) :- true |
        np_0__0(In, Out0, Tree).

    a_0([n_0(In1,Tree1)|In], Out0, Tree) :- true |
        Out1 = [np_0(In1,np_0((Tree,Tree1)))|Out2],
        np_0(In1, Out2, np_0((Tree,Tree1))),
        Out0 = {Out1,Outz},
        a_0(In, Outz, Tree).
    a_0([PE|In], Out0, Tree) :- integer(PE) |
        a_0(In, Out0, Tree)@processor(PE).
    a_0([], Out0, Tree) :- true | Out0 = [].
    otherwise.
    a_0([_|In], Out0, Tree) :- true |
        a_0(In, Out0, Tree).

    prp_0([n_0(In1,Tree1)|In], Out0, Tree) :- true |
        Out1 = [np_0(In1,np_0((Tree,Tree1)))|Out2],
        np_0(In1, Out2, np_0((Tree,Tree1))),
        Out0 = {Out1,Outz},
        prp_0(In, Outz, Tree).
    prp_0([PE|In], Out0, Tree) :- integer(PE) |
        prp_0(In, Out0, Tree)@processor(PE).
    prp_0([], Out0, Tree) :- true | Out0 = [].
    otherwise.
    prp_0([_|In], Out0, Tree) :- true |
        prp_0(In, Out0, Tree).

    v_0([a_0(In1,Tree1)|In], Out0, Tree) :- true |
        Out1 = [vp_0(In1,vp_0((Tree,Tree1)))],
        Out0 = {Out1,Outz},
        v_0(In, Outz, Tree).
    v_0([av_0(In1,Tree1)|In], Out0, Tree) :- true |
        Out1 = [vp_0(In1,vp_0((Tree,Tree1)))],
        Out0 = {Out1,Outz},
        v_0(In, Outz, Tree).
    v_0([np_0(In1,Tree1)|In], Out0, Tree) :- true |
        Out1 = [vp_0(In1,vp_0((Tree,Tree1)))],
        Out0 = {Out1,Outz},
        v_0(In, Outz, Tree).
    v_0([PE|In], Out0, Tree) :- integer(PE) |
        v_0(In, Out0, Tree)@processor(PE).
    v_0([], Out0, Tree) :- true | Out0 = [].
    otherwise.
    v_0([_|In], Out0, Tree) :- true |
        v_0(In, Out0, Tree).
```

並列一般化LRパーザの負荷分散の検討

沼崎 浩明 田中 穂積
Hiroaki NUMAZAKI Hozumi TANAKA

東京工業大学 工学部
Tokyo Institute of Technology
〒152 目黒区大岡山 2-12-1
03-726-1111 (内線 4175)
E-mail : numazaki@cs.titech.ac.jp

概要

我々は，LR構文解析法[Knuth 65]に基づく並列パーザを並列論理型言語KL1で実現した．この並列LRパーザは文の構文的解釈の曖昧性を並列に計算する．ただし，複数のプロセスの解析過程が重複する場合は，これを一つのプロセスに統合する．我々の方法はこの点で富田法[Tomita 85]に基づいている．本報告では，我々の並列LRパーザをマルチPSI実機で実行するための負荷分散の方法を検討する．そこでは実際に，CFGで記述した英語の文法規則を用いて実験を行ない，解析時間の台数効果を測定した．

1 まえがき

本報告では，KL1で記述した並列LRパーザ[沼崎 89]をマルチPSI実機で実行するための負荷分散の方法について検討する．LR構文解析法を一般の文脈自由文法に適用すると，パーザを駆動するLRパーズ表にコンフリクトを生じ，そこで解析が非決定的となる．我々の並列LRパーザは，コンフリクトの生じたエントリが指定する複数の処理を，それぞれ異なったプロセッサ上で実行する．これにより，自然言語の構文的な曖昧性が並列に処理される．

本報告では並列一般化LRパーザのKL1による実現方法を簡単に示し，これをマルチPSI実機で実行するための負荷分散の方法を示す．マルチPSIのような疎結合型の並列計算機では，プロセッサ間の通信のオーバヘッドが問題となる．我々のLRパーザは複数のプロセスの解析が重複する場合，これを一つに統合する．そのためには，一方のプロセッサの持つスタックの情報をもう一方のプロセッサに転送する必要がある．各スタックには，それが得られるまでの解析の情報が全て含まれているため，通信のオーバヘッドは解析の効率の点から無視できない大きさとなる．そこで我々は，プロセッサ間の通信量を抑えるために，一旦負荷分散したプロセスは，他のプロセッサ上にあるプロセスとは通信しないようなインプリメントを行なった．そこでは，異なるプロセッサ上のプロセスの重複を許す．しかし，一つのプロセッサが複数のパージングプロセスを扱う必要が生じた場合は，スタックの木構造化によって重複した解析を防ぐ．

(1) S → NP, VP.
(2) S → S, PP.
(3) NP → NP, PP.
(4) NP → det, noun.
(5) NP → pron.
(6) VP → v, NP.
(7) PP → p, NP.

図 1: 曖昧な英語の文法

	det	noun	pron	v	p	$	NP	PP	VP	S
0	sh1		sh2				4			3
1		sh5								
2				re5	re5	re5				
3					sh6	acc		7		
4				sh8	sh6			10	9	
5				re4	re4	re4				
6	sh1		sh2				11			
7					re2	re2				
8	sh1		sh2				12			
9					re1	re1				
10				re3	re3	re3				
11				re7	sh6/re7	re7		10		
12					sh6/re6	re6		10		

図 2: LR パーズ表

本方式をマルチ PSI実機で実行し，その場合の台数効果を測定するために，規則数 123 の文脈自由文法を用いて，使用するプロセッサの台数に対する解析時間の推移を測定した．その結果，最良値として，2倍程度の台数効果が得られた．この値は寿崎らの報告[寿崎 89]を参考にすれば，初期の実験結果としては妥当な値であると言える．

今回実験した負荷分散の方法は，一台のプロセッサに一回しかプロセスを投げないため，そのプロセスの実行が終了すると，そのプロセッサは稼働しなくなる．この点は今後改善する必要がある．

2 KL1による並列一般化LRパーザの記述

本節では，並列LRパーザのKL1による記述を示す．これを説明するために簡単な英語のCFG文法と，それから得られるLRパーズ表を図1, 図2に示す．図2のpの列の11,12の行にはコンフリクトが生じている．このようなエントリをシフト・レデュースコンフリクトと呼ぶ．パージングプロセスはこのエントリに到達すると，二つに分かれてそれぞれの解析を行なう．

並列一般化LRパーザのKL1による実現の特徴は，LRパーズ表の各エントリをパージングプロセスの記述に置き換え，そこでスタックを操作する点にある．パージングプロセスは以下の5つのプロセスから構成される．

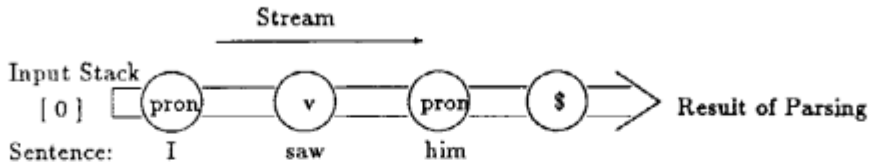


図 3: 親プロセスとストリーム

1. 親プロセス
   入力文中の各入力語に対して起動するプロセス．このプロセス間にストリームを張り，そこにスタックを流す．各親プロセスは入力ストリームにスタックを一つ受けとると，エントリプロセスを呼び出してそのスタックを処理する．その結果得られたスタックは直ちに出力ストリームに送られ，それが次の親プロセスの入力となる (図3参照)．親プロセスは文法規則中の任意の前終端記号 (preterminal) に対して記述する．

2. エントリプロセス
   LR パーズ表の各エントリのスタックの操作を実行するプロセス．このプロセスは，親プロセスから先読み語のカテゴリと，スタックの先頭の状態番号によって定められる．各エントリプロセスに対して一つずつスタックが渡される．

3. reduce プロセス
   スタックを必要な数だけ reduce するプロセス．reduce が木構造化スタックの分岐点に及ぶと，その先のそれぞれの枝に対して別々の reduce 操作を実行する．

4. condition プロセス
   規則が DCG 文法で与えられている場合，reduce 操作の際にその補強項を実行するプロセス．補強項の成功／失敗によってパージングプロセスの継続／終了を判定する．

5. merge_stack プロセス
   複数のスタックの先頭の要素が等しい場合，その部分を統合して一つの木構造化スタックを生成するプロセス．以下に木構造化したスタックの例を示す．

```
[ 6,p, [12,np,8,v,4,np,0], [3,s,0] ]
```

- 親プロセスの記述例
  例えば，カテゴリ p に対する親プロセスは，

```
p( [], _, Out ):- true |
   Out = [].
p( [Stack|Stream], Cat, Out ):- true |
   p_( Stack, Cat, Out1 ),
   p( Stream, Cat, Out2 ),
   merge_stack( Out1, Out2, Out ).
```

と記述する．ここで，P_はエントリプロセスであり，各エントリに対して以下のように記述する．

- シフトエントリの記述例
状態1，先読み語がnounのエントリ'sh 5'は，

```
noun_( [1|Stack], Cat, Out ):- true |
    Out = [ [ 5,Cat,1 | Stack ] ].
```

と記述する．

- レデュースエントリの記述例
状態2，先読み語が'v'のエントリ're 5'は，

```
v_( [2,T1|Stack], Cat, Out ):- true |
    cond( 5, Stack, [T], Result ),
    v(Result, Cat, Out ).
```

と記述する．

- シフト・レデュースコンフリクトのあるエントリの記述例
状態11，先読語がpのエントリ'sh 6/re 7'は，

```
p_( [11,T|Stack], Cat, Out ):- true |
    reduce( 7, 1, Stack, [T], Result ),
    p( Result, Cat, Out1 ),
    merge_stack( [ [6,Cat,11,T|Stack] ], Out1, Out ).
```

と記述する．reduceプロセスは第二引数に与えられた数だけスタックの要素をポップし，それをconditionプロセスに渡す．

- パーザの起動
例えば，入力文の各単語のカテゴリがpron，v，nで，それに付随する情報がPron，V，Nの時，次のような親プロセスの列を呼ぶことによってパーザを起動する．

```
?- pron( [ [0] ], Pron, Stream1 ),
   v( Stream1, V, Stream2 ),
   n( Stream2, N, Stream3 ),
   $( Stream3, [], Result ).
```

最初のゴールpronに一つのスタック［0］を与える．

3 負荷分散

この節では，並列LRパーザの負荷分散について図4を用いて説明する．図4(a)(b)(c)の各円はパージングプロセスのある時点での状態を示し，プロセスは左から右へと推移していくものとする．最初は一つのプロセスで入力文を解析していき，解析が曖昧になると複数のプロセスに分かれる．(図(a)参照)．それぞれのプロセスには重複する部分が存在するため，これを削減すると全体の計算量が減る(図(b)参照)．しかし，マルチPSIのような疎結合型の計算機では，各プロセスをそのまま負荷分散すると通信コストが膨大になる．なぜなら，図(b)の全ての斜め線の部分でプロセスの間の通信が起こるためである．

そこで本報告では，異なるプロセッサ上に移動したプロセスは，他のプロセッサ上にあるプロセスとは通信しないような負荷分散を実現した(図(c)参照)．プロセッサ数の制限から一つのプロセッサが複数のプロセスを扱う場合は，スタックの木構造化によって解析の重複を避ける．図(c)は使用するプロセッサの台数が4台の場合のプロセスの推移を示している．この図中の3箇所の矢印で示された部分でプロセッサ間の通信が起こる．

負荷分散は以下のようにして行なう．

- 各プロセスは，自分がこれから負荷分散を行なうことができるプロセッサの台数を持つ．
- 負荷分散を行なう際に，プロセス数が使用可能なプロセッサ数よりも多い場合は，各プロセッサにできるだけ均等にプロセスを割り当てる．この場合，分散された各プロセスはそれ以上負荷分散を行なわない．
- 負荷分散を行なう際に，プロセス数が使用可能なプロセッサ数よりも少ない場合は，使用可能なプロセッサ数をできるだけ均等に分割して，各プロセスに割り当てる．

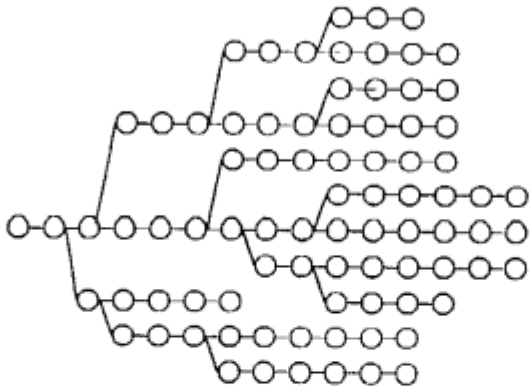
図(c)中の数字はプロセスに割り当てられたプロセッサの台数を示している．ここでは，最初に使用可能なプロセッサの台数は4台であり，プロセスはPE0で実行される．そのプロセスが二つに分岐する時，一方のプロセスはPE0で実行を継続し，もう一方のプロセスはPE2へ移動する．その際，各プロセスに割り当てられるプロセッサ数は2台となる．この図のように，異なるプロセッサ上のプロセス間では，通信を行なわないようにインプリメントした．これによって，通信のコストがある程度大きくても，各プロセッサ上の処理量がそれ以上に大きければ，台数効果を期待できる．ただし，この負荷分散の方法は次の二つの問題点がある．

- あるプロセッサ上の全てのプロセスが処理を終えると，そのプロセッサは二度と稼働しない．
- あるパージングプロセスが，自分に割り当てられたプロセッサの全てを使い尽くさない場合，最初から全く稼働しないプロセッサが生ずる．

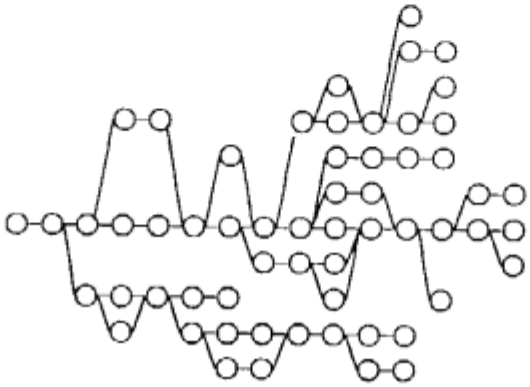
この二点は今後，動的負荷分散などの方法を導入することによって改善する必要がある．

4 実験

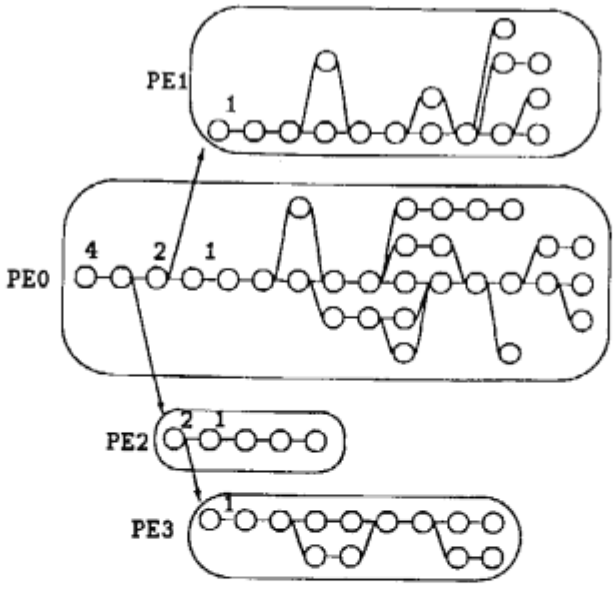
今回の実験では，規則数123の文脈自由文法を用いて，本方式を実現する並列LRパーザを構築し，以下の5つの入力文に対する解析時間を，使用するプロセッサの台数を変えて測定した．使用した計算機は16台のプロセッサを有するマルチPSI実機である．



(a) プロセスを統合しない場合



(b) 統合可能なプロセスを全て統合した場合



(c) 同一プロセッサ上のプロセスのみ統合する本方式
(プロセッサ数4台の場合の例)

図 4: プロセスの推移

文	木の数	プロセッサの台数	Reduction 数	解析時間(ms)
1	30	1	7550	671
		2	7776	522
		4	8306	483
		6	8358	416
2	56	1	14605	1521
		2	15140	1196
		4	15213	1018
		8	16739	938
		10	17543	854
		12	17535	913
3	192	1	88291	7765
		2	92504	6302
		4	92409	6241
		8	102335	5457
		10	111435	5223
		12	113222	5960
4	200	1	142567	11392
		2	135209	6005
		4	175675	7102
		8	199789	7832
		12	203661	7007
5	186	1	16972	1388
		2	16981	1331
		4	20492	1250
		8	23169	1230
		12	24445	943
		16	24506	885

表 1: 解析時間の推移

1. Diagram analyzes all of the basic kinds of phrases and sentences.

2. This paper presents an explanatory overview of a large and complex grammar that is used in a sentence.

3. The annotations provide important information for other parts of the system that interpret the expression in the context of a dialogue.

4. For every expression it analyzes, diagram provides an annotated description of the structural relations holding among its constituents.

5. Procedures can also assign scores to an analysis, rating some applications of a rule as probable or as unlikely.

表 1に解析時間と reduction 数を示す.

5 おわりに

今回の実験では最良で 2 倍程度の台数効果による解析効率の向上を得ることができた. 寿崎らの報告 [寿崎 89] を参考にすれば, この値は初期の実験として妥当であると思われる.

今後，負荷分散の方式の改善や通信する情報の圧縮などを行なえば，さらに良い結果が得られると期待できる．

謝辞

本研究を進めるにあたり，日頃から暖かいご支援をいただいた田中研究室のみなさんに感謝致します．また，本研究に対し貴重な御意見をいただいたICOTのKL1タスクグループの方々に感謝致します．

参考文献

[寿崎 89] 寿崎かすみ，他:マルチPSIにおける並列構文解析プログラムPAXの実現および評価，並列処理シンポジウムJSPP '89, PP.343-350 (1989)

[沼崎 89] 沼崎浩明，田村直良，田中穂積:並列論理型言語による一般化LR構文解析アルゴリズムの実現，自然言語処理 74-5, PP.33-40 (1989)

[沼崎 90] 沼崎浩明，田中穂積:LR法に基づく新しい並列構文解析アルゴリズム，情報処理学会第40回全国大会, 4F-3, pp.456-457 (1990)

[峯 89] 峯 恒憲，谷口倫一郎，雨宮真人:文脈自由文法の並列構文解析，情報処理学会自然言語処理研究会研究報告，73-1, pp.1-8 (1989)

[Aho 72] Aho,A.V.and Ulman,J.D.: *The Theory of Parsing,Translation,and Compiling,* Prentice-Hall,Englewood Cliffs,New Jersey (1972)

[Henry 89] Henry S. Thompson:*Chart Parsing for Loosely Coupled Parallel Systems,* Proc. of International Workshop on Parsing Technologies pp.320-328 (1989)

[Knuth 65] Knuth,D.E.: *On the translation of languages from left to right,*Information and Control 8:6,pp.607-639

[Matsumoto 88] Matsumoto,Y.:*Natural Language Parsing System based on Logic Programming,* Doctoral Thesis，Kyoto University (1988)

[Nijholt 89] Anton Nijholt:*Parallel Parsing Strategies in Natural Language Processing,* Proc. of International Workshop on Parsing Technologies pp.240-253 (1989)

[Tomita 85] Tomita,M.:*Efficient Parsing for Natural Language,* Kluwer Academic Publishers (1985)

[Tomita 87] Tomita,M.: *An Efficien Augmented-Context-Free Parsing Algorithm,* Computational Linguistics, Vol.13, Numbers 1-2, pp.31-46 (1987)

[Ueda 85] Ueda,K:*Guarded Horn Clauses,* Proc. The Logic Programming Conference, Lecture Notes in Computer Science, 221 (1985)

ＫＬ１プログラミング雑感－－ｐｒｏｖｅｒの並列化の体験より－－

１９９０．４．２３ 渕

１．いきさつ

ＫＬ１はこのプロジェクトの基本言語である。考え方については聞いているし了承しているものであるが、実作業については皆さんにお委せしてきた。今でもそれで良いと思っている。

ところで、昨年末、ＰＳＩをやっと手元において貰うことになった。しばらくＳＩＭＰＯＳ／ＥＳＰほかで遊んでいたが、２／１６（というのは実は私の誕生日）を期してＫＬ１を試みることにした。基本的には、老化度チェックの頭の体操のつもり。

題材としては、ｐｒｏｖｅｒを選んだ。正月の拡大所議で古川次長が紹介したＳａｔｃｈｍｏが大変コンパクトなｐｒｏｖｅｒ（の一種）で、私程度に手ごろかと思ったことが一つ。ｐｒｏｖｅｒ自体についての議論はここでは省くが、本来ＦＧＣＳの基本技術（の一つ）であるはずのもので、知識処理的テーマ（いわゆる山側テーマ）に直接、間接に関わっている。

ＫＬ１利用については、ＰＩＭＯＳなどシステム・プログラミングでは着実に進展しているようであるが、山側では（頑張ってくれている人達ももちろんいるが）私が期待したほどではないようである。ウワサでは、ＫＬ１は使い難いらしい。そういえば、昨年正月の拡大所議で、ＫＬ１のユニフィケーションについての議論（不用論？）が少しあった。議論を煮詰めて欲しいと要望してあったのだが（してくれたのかもしれないが）、その後が判然としない。ウワサは本当なのか。自分でも実感を若干掴んでおきたいという気もあった。

２．やってみたこと（その１）

やったことの実体は、Ｓａｔｃｈｍｏ的動作のＫＬ１によるサンプル・コーディングである。（ここで本当はＳａｔｃｈｍｏの説明が必要だが、省略） 始める前の見通しとしては、

（１）ｏｒ並列的動作は、ａｎｄ並列で代用できる。（あまり問題はないはず。但し（その２）を参照のこと）

（２）ユニフィケーションにはなんらかの工夫を要する。それを丸々再コーディングするのでは遅くなりすぎるだろう。これは皆さん体験済み。出来ればＫＬ１の機能を流用したい。コンパイル方式の方が可能性がありそう。

ということで、「問題」をＫＬ１にコンパイルしたとして、想定すべきオブジェクト・コードを工夫することにした。

ところが幸いなことに、Ｓａｔｃｈｍｏには ｒａｎｇｅ－ｒｅｓｔｒｉｃｔｅｄとい

う前提がある。その中で生成されるモデルは基底項だけで構成される。

そこで、実際に必要なのは、ユニフィケーションというより、マッチングである。また、コンパイル方式だから、新変数の導入が楽である。これで変数の依存関係を後送りに出来、また、ｏｒ並列の実現も簡単化できる。ということで第一の関門は抜けることが出来た。

速度は、現段階では、例えば、原論文　９．５秒（Ｓｅｔｈｅｏでは　４．５秒）の問題（それまでは分オーダ）が　２３０ｍｓほど。最適化をも少し入れると　１００ｍｓをきる。

コンパイラはまだ作っていない。本当の並列実行もこれからである。インタプリタは無理かと思っていたが、長谷川／藤田が巧みに実現した。

（３）やったこと（その２）

ところで、長谷川／藤田のアイデアに近いものとして、ＳｈａｐｉｒｏのＦＣＰによるｏｒ並列全解探索ｐｒｏｌｏｇインタプリタがある。少し制限がつくようだが、ＫＬ１にも移植できる。実効はともかく、ｏｒ並列＋ａｎｄ並列の実現もできる。ということでＫＬ１版を作った。

そのアイデアの基本は、ｔｒａｉｌを保存して、ｏｒ分岐のとき変数のコピーを作りなおすというものである。

（４）雑感

１．ＫＬ１は面白い。

並列実行はまだ実際には試みていないが、疑似でも面白い。横型探索の本格的言語の実体験は初めてなので大いに楽しめた。（皆さん、新しい玩具は楽しくないですか）

２．ｐｒｏｖｅｒの実現から見て。

横型探索（並列）は、ｆａｉｒｎｅｓｓを（ある程度）保証しているのでｐｒｏｖｅｒには有利である。

ＫＬ１変数（ユニフィケーション）も結構使える。使いこなす工夫を色々して、その上で拡張や改良の提案が出てくるとよいと思う。

これはもっと考えなければならないが、変数自体を扱う機能がやはり欲しい気がする。それは、すごく軽い（簡単だが有効な）ｆｒｅｅｚｅ－ｍｅｌｔのようなものだろうか。

３．もっとマクロな表現はないか。

各種のブルーバ、問題解決器を書くための上位言語（マクロ表現）はやはり欲しい気がする。ＫＬ１から素直に上昇するやり方があると良いと思うのだが。

```
%%% Satchmo a la natural deduction

macro_bank mg has
(A => C)  => false(S)
      where eval(A, S),
            not(eval(C, S)),
            unsatisfy(C, S, S), !;
unsatisfy((C1;C2), X, S)  => unsatisfy(C1, X, S),
                             unsatisfy(C2, X, S);
unsatisfy((C1, C2), X, S)  => unfatisfy(C2, [C1|X], S);
unsatisfy(false, X, S)     => true;
unsatisfy(C, X, S)  =>        false([C|X]), !;

eval((X, Y), S)  => eval(X, S), eval(Y, S);
eval((X;Y), S)  => (eval(X, S);eval(Y, S));
eval(:X, S)   => X;
eval(true, S)   => true;
eval(false, S)  => fail;
eval(X, S)      => true(X, S);

:- inserta(1, (
      true(X, [X|S]);
      true(X, [Y|S]):-true(X, S)));
:- inserta(c, (:do(C, R):-false([]))));
end.

include("mg.esp").
class mg with_macro mg has
local

true=>  (p(a);q(b));
q(X) =>  s(f(X));
r(X) =>  s(X);
p(X) =>  (q(X);r(X));
q(X), s(Y) => false;
p(X), s(X) => false;

end.
```

```
:- module mg_s1.
:- public do/1.
%%  problem s1
do(N) :-N>0|false([],T),N1:=N-1,do(N1).  otherwise.   do(_).
false(F,T) :-true|c11(F,F,T),c21(F,F,T),c31(F,F,T),
                  c41(F,F,T),c51(F,F,T),c61(F,F,T).
%%      true=>  (p(a);q(b));
c11(F1,F,t).   alternatively.
c11(F1,F,T) :-true| (F1=[]    ->c12(F,F,T);
                F1=[p(a)|F2]  ->true ;
 otherwise;     F1=[Z|F2]     ->c11(F2,F,T)).
c12(F1,F,t).   alternatively.
c12(F1,F,T) :-true| (F1=[]    ->c13(F,T);
                F1=[q(b)|F2]  ->true ;
 otherwise;     F1=[Z|F2]     ->c12(F2,F,T)).
c13(F,T)    :-true|false([p(a)|F],T1),false([q(b)|F],T2),
                both(T1,T2,T).
both(T1,T2,T) :-T1=t,T2=t        |T=t.
otherwise.   both(T1,T2,T) :-true|T=abort.
%%      p(X) =>  (q(X);r(X));
c21(F1,F,t).   alternatively.
c21(F1,F,T) :-true| (F1=[]    ->true ;
                F1=[p(X)|F2]  ->c22(X,F,F,T),c21(F2,F,T);
 otherwise;     F1=[Z|F2]     ->c21(F2,F,T)).
c22(X,F1,F,t).   alternatively.
c22(X,F1,F,T) :-true| (F1=[]->c23(X,F,F,T);
                F1=[q(X)|F2]  ->true ;
 otherwise;     F1=[Z|F2]     ->c22(X,F2,F,T)).
c23(X,F1,F,t).   alternatively.
c23(X,F1,F,T) :-true| (F1=[]->c24(X,F,T);
                F1=[r(X)|F2]  ->true ;
 otherwise;     F1=[Z|F2]     ->c23(X,F2,F,T)).
c24(X,F,T)  :-true|false([q(X)|F],T1),false([r(X)|F],T2),
                both(T1,T2,T).
%%      q(X) => s(f(X));
c31(F1,F,t).   alternatively.
c31(F1,F,T) :-true| (F1=[]    ->true ;
                F1=[q(X)|F2]  ->c32(X,F,F,T),c31(F2,F,T);
 otherwise;     F1=[Z|F2]     ->c31(F2,F,T)).
c32(X,F1,F,t).   alternatively.
c32(X,F1,F,T) :-true| (F1=[]->c33(X,F,T);
                F1=[s(f(X))|F2]->true ;
 otherwise;     F1=[Z|F2]     ->c32(X,F2,F,T)).
c33(X,F,T)   :-true           |false([s(f(X))|F],T).
%%      r(X) => s(X);
c41(F1,F,t).   alternatively.
c41(F1,F,T) :-true| (F1=[]    ->true ;
                F1=[r(X)|F2]  ->c42(X,F,F,T),c41(F2,F,T);
 otherwise;     F1=[Z|F2]     ->c41(F2,F,T)).
c42(X,F1,F,t).   alternatively.
c42(X,F1,F,T) :-true| (F1=[]->c43(X,F,T);
                F1=[s(X)|F2]  ->true ;
 otherwise;     F1=[Z|F2]     ->c42(X,F2,F,T)).
c43(X,F,T)   :-true|false([s(X)|F],T).
%%      q(X),s(Y) => false;
c51(F1,F,t).   alternatively.
c51(F1,F,T) :-true| (F1=[]    ->true ;
                F1=[q(X)|F2]  ->c52(F,F,T),c51(F2,F,T);
 otherwise;     F1=[Z|F2]     ->c51(F2,F,T)).
c52(F1,F,t).   alternatively.
c52(F1,F,T) :-true| (F1=[]    ->true ;
                F1=[s(Y)|F2]  ->T=t;
 otherwise;     F1=[Z|F2]     ->c52(F2,F,T)).
%%      p(X),s(X) => false;
c61(F1,F,t).   alternatively.
```

```
c61 (F1, F, T) :-true¦ (F1= []   ->true  :
                  F1= [p (X) ¦F2]  ->c62 (X, F, F, T), c61 (F2, F, T) :
 otherwise:   F1= [Z¦F2]       ->c61 (F2, F, T) ).
c62 (X, F1, F, t).   alternatively.
c62 (X, F1, F, T) :-true¦ (F1= [] ->true  :
                  F1= [s (X) ¦F2]  ->T=t;
 otherwise:   F1= [Z¦F2]       ->c62 (X, F2, F, T) ).
```

```
%% problem s3      possiblely or parallel
%%  with complement splittig
%%
%%      dom(a):dom(b):dom(c):dom(d):
%%                p(a, b) => false;
%%                q(c, d) => false;
%%       p(X, Y), p(Y, Z)  => p(X, Z);
%%       q(X, Y), q(Y, Z)  => q(X, Z);
%%              q(X, Y)  => q(Y, X);
%%      :dom(X), :dom(Y) =>  (p(X, Y);q(X, Y));
%%
:-module mg_s3.
:-public do/1.
do(N):-N>0|false([], [], T), N1:=N-1, do1(T, N1). otherwise. do(_).
do1(t, N):-true|do(N).   otherwise.   do1(_, _).
%%      dom(a):dom(b):dom(c):dom(d):
dom(D):-true|D=[a, b, c, d].
both(T1, T2, T):-T1=t, T2=t|T=t.
 otherwise. both(T1, T2, T):-true|T=abort.
false(G, F, T):-true|c11(G, F, F, T), c21(G, F, F, T), c31(G, F, F, T),
                     c41(G, F, F, T), c51(G, F, F, T), c6(G, F, T).
%%                p(a, b) => false;
c11(G, F1, F, t).   alternatively.
c11(G, F1, F, T):-true|(F1=[]    ->true;
                F1=[p(a, b)|F2]->T=t;
 otherwise:     F1=[Z|F2]       ->c11(G, F2, F, T)).
%%                q(c, d) => false;
c21(G, F1, F, t).   alternatively.
c21(G, F1, F, T):-true|(F1=[]    ->true;
                F1=[q(c, d)|F2]->T=t;
 otherwise:     F1=[Z|F2]       ->c21(G, F2, F, T)).
%%                q(X, Y)  => q(Y, X);
c31(G, F1, F, t).   alternatively.
c31(G, F1, F, T):-true|(F1=[]    ->true;
                F1=[q(X, Y)|F2]->c32(X, Y, G, F, F, T), c31(G, F2, F, T);
 otherwise;     F1=[Z|F2]       ->c31(G, F2, F, T)).
c32(X, Y, G, F1, F, t).   alternatively.
c32(X, Y, G, F1, F, T):-true|(F1=[]->c33(X, Y, G, G, F, T);
                F1=[q(Y, X)|F2]->true;
 otherwise;     F1=[Z|F2]       ->c32(X, Y, G, F2, F, T)).
c33(X, Y, G, F1, F, t).   alternatively.
c33(X, Y, G1, G, F, T):-true|(G1=[]->false(G, [q(Y, X)|F], T);
                G1=[q(Y, X)|G2]   ->T=t;
 otherwise;     G1=[E|G2]         ->c33(X, Y, G2, G, F, T)).
%%       q(X, Y), q(Y, Z)  => q(X, Z);
c41(G, F1, F, t).   alternatively.
c41(G, F1, F, T):-true|(F1=[]      ->true;
                F1=[q(X, Y)|F2]   ->c42(X, Y, G, F, F, T), c41(G, F2, F, T);
 otherwise;     F1=[E|F2]         ->c41(G, F2, F, T)).
c42(X, Y, G, F1, F, t).   alternatively.
c42(X, Y, G, F1, F, T):-true|(F1=[]->true;
                F1=[q(Y, Z)|F2]   ->c43(X, Z, G, F, F, T), c42(X, Y, G, F2, F, T);
 otherwise;     F1=[E|F2]         ->c42(X, Y, G, F2, F, T)).
c43(X, Z, G, F1, F, t).   alternatively.
c43(X, Z, G, F1, F, T):-true|(F1=[]->c44(X, Z, G, G, F, T);
                F1=[q(X, Z)|F2]   ->true;
 otherwise;     F1=[E|F2]         ->c43(X, Z, G, F2, F, T)).
c44(X, Z, G1, G, F, t).   alternatively.
c44(X, Z, G1, G, F, T):-true|(G1=[]->false(G, [q(X, Z)|F], T);
                G1=[q(X, Z)|G2]   ->T=t;
 otherwise;     G1=[E|G2]         ->c44(X, Z, G2, G, F, T)).
%%       p(X, Y), p(Y, Z)  => p(X, Z);
c51(G, F1, F, t).   alternatively.
c51(G, F1, F, T):-true|(F1=[]      ->true;
                F1=[p(X, Y)|F2]   ->c52(X, Y, G, F, F, T), c51(G, F2, F, T);
```

```
 otherwise:    F1=[E|F2]          ->c51(G, F2, F, T)).
c52(X, Y, G, F1, F, t).   alternatively.
c52(X, Y, G, F1, F, T) :-true| (F1=[] ->true;
                F1=[p(Y, Z)|F2]   ->c53(X, Z, G, F, F, T), c52(X, Y, G, F2, F, T) :
 otherwise:    F1=[E|F2]          ->c52(X, Y, G, F2, F, T)).
c53(X, Z, G, F1, F, t).   alternatively.
c53(X, Z, G, F1, F, T) :-true| (F1=[] ->c54(X, Z, G, G, F, T) :
                F1=[p(X, Z)|F2]   ->true:
 otherwise:    F1=[E|F2]          ->c53(X, Z, G, F2, F, T)).
c54(X, Z, G1, G, F, t).   alternatively.
c54(X, Z, G1, G, F, T) :-true| (G1=[] ->false(G, [p(X, Z)|F], T) :
                G1=[p(X, Z)|G2]   ->T=t;
 otherwise:    G1=[E|G2]          ->c54(X, Z, G2, G, F, T)).
%%      :dom(X), :dom(Y) =>  (p(X, Y) ;q(X, Y)) :
c6(G, F, T) :-true|dom(L), c61(G, L, L, F, T).
c61(G, L1, L, F, t).   alternatively.
c61(G, L1, L, F, T) :-true| (L1=[]    ->true;
                L1=[X|L2]          ->c62(X, G, L, L, F, T), c61(G, L2, L, F, T)).
c62(X, G, L1, L, F, t).   alternatively.
c62(X, G, L1, L, F, T) :-true| (L1=[] ->true;
                L1=[Y|L2]   ->c63(X, Y, G, F, F, T), c62(X, G, L2, L, F, T)).
c63(X, Y, G, F1, F, t).   alternatively.
c63(X, Y, G, F1, F, T) :-true| (F1=[] ->c64(X, Y, G, F, F, T) :
                F1=[p(X, Y)|F2]   ->true;
 otherwise:    F1=[E|F2]          ->c63(X, Y, G, F2, F, T)).
c64(X, Y, G, F1, F, t).   alternatively.
c64(X, Y, G, F1, F, T) :-true| (F1=[] ->c65(X, Y, G, G, F, T) :
                F1=[q(X, Y)|F2]   ->true:
 otherwise:    F1=[E|F2]          ->c64(X, Y, G, F2, F, T)).
c65(X, Y, G1, G, F, t).   alternatively.
c65(X, Y, G1, G, F, T)  :-true| (G1=[] ->false([q(X, Y)|G], [p(X, Y)|F], T1),
                                  c66(X, Y, T1, G, G, F, T) :
                    G1=[p(X, Y)|G2] ->c66(X, Y, t, G, G, F, T) ;
 otherwise:         G1=[E|G2]       ->c65(X, Y, G2, G, F, T)).
c66(X, Y, T1, G1, G, F, t).   alternatively.
c66(X, Y, T1, G1, G, F, T) :-true| (G1=[] ->false(G, [q(X, Y)|F], T2),
                                     both(T1, T2, T) :
                      G1=[q(X, Y)|G2] ->both(T1, t, T) ;
 otherwise:           G1=[E|G2]       ->c66(X, Y, T1, G2, G, F, T)).
```

```
:-module orp.
:-public do1/1, do2/1, solve/2.

%%%%%   prolog interpreter/or_parallel and and_parallel
solve(FG, Sols) :-true|merge({Sols1, Sols2}, Sols),
      melt(FG, G),
      explore([G], [], Trail, FG, G, Sols1, T, Abort),
      solution(T, G, Sols2).
solution(t, G, Sols) :-true|Sols=[G].
solution(f, G, Sols) :-true|Sols=[].
explore(_, Tr0, Trail, _, _, Sols, T, abort) :-true|
      T=f, Sols=[], Tr0=Trail.
alternatively.
explore(fail, Tr0, Trail, FG, G, Sols, T, Abort) :-true|
      Sols=[], T=f, Abort=abort, Tr0=Trail.
explore([], Tr0, Trail, FG, G, Sols, T, Abort) :-true|
      Sols=[], Trail=Tr0, T=t.
explore([A|As], Tr0, Trail, FG, G, Sols, T, Abort) :-wait(G)|
      Sols={Sols1, Sols2}, both(T1, T2, T),
      clauses(A, Cs),
      explore1(Cs, A, Tr0, Trail1, FG, G, Sols1, T1, Abort),
      explore(As, Trail1, Trail, FG, G, Sols2, T2, Abort).
%explore1(_, _, Tr0, Trail, _, _, Sols, T, abort) :-true|
%      T=f, Sols=[], Tr0=Trail.
%alternatively.
explore1([], A, Tr0, Trail, FG, G, Sols, T, Abort) :-true|
      Sols=[], T=f, Abort=abort, Tr0=Trail.
explore1([C|Cs], A, Tr0, Trail, FG, G, Sols, T, Abort) :-true|
      Sols={Sols1, Sols2},
      clause(C, A, Bs/[]),
      explore(Bs, [C|Tr0], Trail, FG, G, Sols1, T, Abort),
      explore_rest(Cs, Tr0, FG, Sols2).
explore_rest([], Trail, FG, Sols) :-true|Sols=[].
explore_rest([C|Cs], Trail, FG, Sols) :-true|Sols={Sols1, Sols2},
      melt(FG, G),
      reverse(Trail, [C], Path),
      trace(Path, [G], [C|Trail], FG, G, Sols1),
      explore_rest(Cs, Trail, FG, Sols2).
trace([C|Cs], [A|As], Trail, FG, G, Sols) :-wait(G)|
      clause(C, A, Bs/As),
      trace(Cs, Bs, Trail, FG, G, Sols).
trace([], As, Trail, FG, G, Sols) :-wait(G)|
      explore(As, Trail, Tr1, FG, G, Sols1, T, Abort),
      solution(T, G, Sols2), Sols={Sols1, Sols2}.
reverse([X|Xs], Ys, Zs) :-true|reverse(Xs, [X|Ys], Zs).
reverse([], Ys, Zs) :-true|Zs=Ys.
both(t, t, T) :-true|T=t.
both(f, _, T) :-true|T=f.
both(_, f, T) :-true|T=f.

%%%%
do1(N) :-N>0|solve(perm([a, b, c, d, e, f], Ys), Sols), N1:=N-1, do1(N1).
otherwise.  do1(_).
do2(N) :-N>0|solve(append(X, Y, [
1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
21, 22, 23, 24, 25, 26, 27, 28, 29, 30
]), Sols), N1:=N-1, do2(N1).
otherwise.  do2(_).

%%%%   test programs
clauses(perm(     [], _), Cs) :-true|Cs=[2].
clauses(perm([_|_], _), Cs) :-true|Cs=[1].
clauses(select(_, [_|_], _), Cs) :-true|Cs=[3, 4].
clauses(select(_,     [], _), Cs) :-true|Cs=[].
```

```
clauses(append(_, _,      []), Cs) :-true|Cs=[5].
clauses(append(_, _, [_|_]), Cs) :-true|Cs=[6, 5].
alternatively.
clauses(append([], _,     _), Cs) :-true|Cs=[5].
clauses(append([_|_], _, _), Cs) :-true|Cs=[6].

clause(1, A, Bs) :-true|A=perm(Xs, [Y|Ys]),
                        Bs=[select(Y, Xs, Xs1), perm(Xs1, Ys)|As]/As.
clause(2, A, Bs) :-true|A=perm([], []),
                        Bs=As/As.
clause(3, A, Bs) :-true|A=select(X, [X|Xs], Xs),
                        Bs=As/As.
clause(4, A, Bs) :-true|A=select(X, [X1|Xs], [X1|Ys]),
                        Bs=[select(X, Xs, Ys)|As]/As.
clause(5, A, Bs) :-true|A=append([], Z, Z),
                        Bs=As/As.
clause(6, A, Bs) :-true|A=append([U|V], Y, [U|W]),
                        Bs=[append(V, Y, W)|As]/As.
otherwise.  clause(_, A, Bs) :-true|Bs=fail/As.

melt(perm(L, _), MG) :-true|MG=perm(L, _).
melt(append(_, _, L), MG) :-list(L)|MG=append(_, _, L).
melt(append(_, _, []), MG) :-true|MG=append(_, _, []).
alternatively.
melt(append(X, Y, _), MG) :-true|MG=append(X, Y, _).
```

KL1による定理証明プログラム

藤田博, 長谷川隆三
ICOT・第5研究室

1. はじめに

我々はKL1をベースにした一階述語論理の高速な定理証明システムの実現を目指している。一階述語論理の定理証明の方式としては、タブロー法、レゾリューション法[Wos 88]、コネクション法[Bibel 86]など幾つかあるが、いずれが我々の目的に最も適しているか現在検討中である。これまでに、タブロー法に属すると考えられる、Prologをベースとした二つの既存の定理証明プログラムの検討を行い、KL1による試作を進めている。一つはStickelのPTTP(Prolog Technology Theorem Prover)[Stickel 88]、もう一つはManthey&BryのSATCHMO[Bry 88]である。いずれもPrologをベースに、その長所を活かしつつ最小限の拡張を行うことによってフルファーストオーダーの定理証明を行えるようにしたものである。

ここで主要な問題となるのは、証明対象となる論理式に現れる変数の扱い方である。これをKL1変数で表現することができるか否か、あるいはこれをKL1の基底項で表現した場合に、変数束縛の伝搬、ユニフィケーション、節のコピーや環境の管理はどうするか、といった問題が生じる。PTTPではオカーチェック付きユニフィケーションが不可欠であるが、SATCHMOの方は前向き推論に限定すれば照合で済む。この点、変数の扱いが簡単になるSATCHMOの方式がKL1インプリメンテーションに適していると思われる。実際、証明対象となる論理式に現れる変数をKL1変数で表現できたために、KL1版SATCHMOはインタプリタ方式でありながら、PSI-II上の擬似並列実行においてすらProlog版に優る速度が得られた。本稿では、このSATCHMOのKL1インプリメンテーションを中心に報告する。

2. SATCHMO

SATCHMOは、任意の一階述語論理式(の否定)をrange-restrictedな節集合に変換したものを対象とする。各節は、前件 → 後件 の形をしており、前件はtrueか原子論理式の連言、後件はfalseか原子論理式の選言である。以下では前件がtrueの節をtrue節、後件がfalseの節をfalse節と呼び、単に節と言うときは前件がtrueでもなく後件がfalseでもない節のことを指す。range-restrictedの条件とは、「後件中に変数が現れていれば、それらはすべて前件中にも現れていなければならない」というものである。SATCHMOの証明方式(逐次)を図1に示す。

ここで、次の例題を考えてみよう。

問題S1:

```
(C1)    p(X), s(X) ---> false.
(C2)    q(X), s(Y) ---> false.
(C3)    q(X) ---> s(g(X)).
(C4)    r(X) ---> s(X).
(C5)    p(X) ---> q(X) ; r(X).
(C6)    true ---> p(a) ; q(b).
```

(1) モデル候補の生成
　(a) true 節:
　　true → F_1 ; F_2 ; ⋯ ; F_n.
　から、今のモデル候補に F_i を追加(拡張)する。
　(F_1, F_2, ⋯, F_n はグランド)
　(b) ある節:
　　A_1, A_2, ⋯ ,A_m → C_1 ; C_2 ; ⋯ ; C_n.
　においてある代入のもとに、今のモデル候補が A_1, A_2, ⋯, A_m の全てを含み、C_1, C_2, ⋯, C_n のいずれも含まなければ、C_j を今のモデル候補に追加(拡張)する。
(2) モデル生成の成功(節集合は充足可能)
　全ての節:
　　A_1, A_2, ⋯ A_m → C_1 ; C_2 ; ⋯ C_n.
　においてある代入のもとに、今のモデル候補が A_1, A_2, ⋯, A_m の全てを含み、C_1, C_2, ⋯, C_n のいずかを含んでいれば、それはこの節集合の一つのモデルである。
(3) モデル生成の失敗
　ある false 節:
　　A_1, A_2, ⋯ A_m → false
　においてある代入のもとに、今のモデル候補が A_1, A_2, ⋯, A_m の全てを含むとき、それはこの節集合のモデルではない。
(4) モデル生成の再試行(backtrack)
　1) でモデル生成に失敗したとき、
　1) の F_i ／ C_j を除く残りの F_{i+1}, F_{i+2}, ⋯, F_n ／ C_{j+1}, C_{j+2}, ⋯, C_n の中からモデル拡張の候補を選びなおす。
(5) モデル生成の全失敗(節集合は充足不能)
　1) で F_1, F_2, ⋯, F_n ／ C_1, C_2, ⋯, C_n の全てのモデル拡張の候補が 3) のモデル生成の失敗を導くとき、この節集合にモデルはない。

図1　SATCHMO の証明方式

C1 から C6 まで全て range-restricted な節である。何故ならば、C1,C2,C6 では後件に変数は現れないし、C3,C4,C5 で後件に現れる変数 X は前件にも現れている。

図1に従えば、まず C6 によって p(a) がモデル候補に加えられる。次に C5 によって q(a) が追加される。さらに C3 によって s(g(a)) が追加される。しかし C2 によってこのモデル候補 {p(a),q(a),s(g(a))} は失敗であることがわかる。そこで C5 で r(a) を選び直す。すると C4 によって s(a) が追加されるが、C1 によって今度のモデル候補 {p(a),r(a),s(a)} も失敗であることがわかる。

そこで C6 に戻って q(b) を選び直す。すると C3 によって s(g(b)) が追加される。しかし、またしても C2 によってモデル候補 {q(b),s(g(b))} が失敗であることがわかる。C5,C6 での選択は全て尽くしたから、モデル生成は全て失敗したことになる。即ち、問題 S1 は充足不能であることがわかる。

以下、特に断らない限り、単にモデルと言う場合は、生成過程の途上にあるモデル候補のことを指すことにする。

3. KL1 インプリメンテーション

SATCHMO を並列化しようとするとき、次の二つの可能性が考えられる。

(1) モデル生成において、モデルに加えるアトムの候補が複数ある場合に並列化 (OR 並列) できる。この場合、モデル生成の再試行 (backtrack) が並列実行で置き換えられる。

(2) 前件の充足を調べるステップにおいて、複数のアトムからなる前件に対するチェックを並列化 (AND 並列) できる。

しかし、簡単のためここでは図1の逐次的アルゴリズムを忠実に KL1 コーディングすることを考える。

3.1 問題点

並列性を如何に引き出すかをさておいたとしても、SATCHMO を KL1 でインプリメントしようとすると大きな困難に直面する (この問題は、Prolog や GHC のメタインタプリタ、Knuth-Bendix の完備化手続きなどを GHC や KL1 でインプリメントしたときの問題と全く同じである)。即ち、問題に現れる変数の扱いである。

Prolog でインプリメントするときには、問題に現れる変数を Prolog 変数で都合よく表現することができた。そこでは、var 述語、組み込みのユニフィケーションが便利に利用できた。厳密にいえばこのような方法はメタとオブジェクトの相違を曖昧にする好ましからぬ用法かもしれないが、プログラムの規模、効率の良さなど実利は確かに大きい。

しかし、KL1 ではこの手は通用しない。言語仕様上、原理的に不可能である。というのは、var 述語のように変数の非束縛を判定することは並列実行下では意味をなさないし、ヘッド側で行われるユニフィケーションは呼び側の変数を具体化しない一方向 (照合) に制限され、ボディ側で行われるユニフィケーションは失敗が許されないからである。

3.2 解決策

上述の問題点に関して、考えられる解決策には二通りある。

(1) 問題に現れる変数は KL1 の基底項で表現する。

(2) 問題に現れる変数をそのまま KL1 変数で表現する。

(1) はメタプログラミングの正道かもしれない。しかし、ユニフィケーション、サブスティテューション、リネーミングといった変数にまつわるあらゆる操作を全て詳細にプログラムする必要がある。そのプログラムの規模、複雑さはメインアルゴリズムの規模、複雑さに比べて無視できないものとなる。当然、そのままではオーダーを数桁も劣化させるようなオーバヘッドのために実行には耐えられない。これを改善する一つの手段は部分計算であろうが、残念なことに現在のところ KL1 において自己適用可能な部分計算が実現されていないため、メタプログラムの一つである部分計算が上述の場合と同種のオーバヘッドを伴い実用的でない。

(2) は (1) とは逆に、ユニフィケーション、サブスティテューション、リネーミングといった変数にまつわる操作をなるべくプログラムせずに、インプリメント言語の処理系に肩代わりさせようというアプローチである。ただし、KL1 では Prolog の場合のようにはうまくいかないので工夫が要る。

```
:- module satchmo_problem.
:- public model/1, nc/1, c/4.

model(M) :- true | M=[].
nc(NC) :- true | NC=6.

c(1,p(X),[],     R) :- true | R=cont.
c(1,s(X),[p(X)],R) :- true | R=false.        %  (C1)  p(X), s(X) ---> false.
c(2,q(X),[],     R) :- true | R=cont.
c(2,s(Y),[q(X)],R) :- true | R=false.        %  (C2)  q(X), s(Y) ---> false.
c(3,q(X),[],     R) :- true | R=[s(g(X))].    %  (C3)  q(X) ---> s(g(X)).
c(4,r(X),[],     R) :- true | R=[s(X)].       %  (C4)  r(X) ---> s(X).
c(5,p(X),[],     R) :- true | R=[q(X),r(X)].  %  (C5)  p(X) ---> q(X) ; r(X).
c(6,true,[],     R) :- true | R=[p(a),q(b)].  %  (C6)  true ---> p(a) ; q(b).
otherwise.
c(_,_,_,R) :- true | R=fail.
```

図2 KL1の節形式に変換された問題S1

SATCHMOの場合、好都合なことにフルユニフィケーションは必要ない。何故ならば、節をモデルにつきあわせて充足性を判定する際、変数は一方の節のみに現れ、他方のモデル側には現れないからである(range-restrictedな節集合から生成されるモデルは常にグランドアトムのみから成る)。即ち、変数の束縛は照合だけで行われるから、この操作はKL1のヘッドユニフィケーションで実現できることがわかる。残る問題は、一つの節の異なるアトムに出現する同名の変数への値の伝搬の方法であるが、一旦照合が済んだアトムはもはや変数を含まない(グランドである)ことを考え合わせると、これもKL1のヘッドユニフィケーションで実現できることがわかるのである。

具体的な例でこれを見てみよう。図2は、先の問題S1をKL1の節形式に変換したものである。c(N,P,GS,R)のNは問題の節の番号、Pは前件中のアトム、GSは前件中でPより左に現れるアトムの列、Rはこの節の呼び出しが成功した場合に返される値である。ここで、C1は二本のKL1節で表現されていることに注意しよう。1番めの節では、C1の最初のアトムp(X)がモデル中の要素(グランドアトム)と照合をとられる。この照合が成功すれば、呼び手(SATCHMOインタプリタ)は対応するp(X)のインスタンスを保持しておき、二つめのアトムs(X)の照合に進む。このとき、呼び手はp(X)のインスタンスを第2引き数としてcの2番めの節を呼ぶことになり、ここでs(X)のXが先のp(X)のXと同じ値をもつことになる。

3.3 インタプリタ

図3にKL1コーディングされたSATCHMOインタプリタを示す。do(A)は、問題に指定された初期モデルMでfalseプロセスを起動し、結果Aを得る。

false(M,A)は、satisfy_clausesを起動して、問題として与えられたNC本の節をモデルMの下で充足させることを試み、その結果をAに与える。Aの値は2通りで、節集合が充足可能な場合は充足した一つのモデルをsat(M1)の形で返し、充足不能の場合は調べた全てのモデルのリストをunsat(Ms)の形で返す。

satisfy_clauses(Cn,NC,M,A2,A)は、逐次的に1番めからNC番めまでの節の充足可能性を順に調べ、その結果をAに返す。Cnは現在調べている節の番号で、A2はそれ以前の節の結果である。各節からの結果はcl_sat、sat(M1)、unsat(Ms)の3通りある。

cl_satは、現在調べているCn節がMで充足されたことを示し、この場合には次の節が調べられる。現在調べているCn節からsat(M1)かunsat(Ms)が返されたときは、以後の節を調べることなくその結果をsatisfy_clauses全体の結果としてAに返す。

```
:- module satchmo.
:- public do/1, false/2.

do(A) :- true | satchmo_problem:model(M), false(M,A).

false(M,A) :- true |
    satchmo_problem:nc(NC),
    satisfy_clauses(0,NC,M,cl_sat,A).

satisfy_clauses(Cn,NC,M,A2,A) :- Cn < NC |
    Cn1 := Cn + 1,
    satisfy_ante(Cn1,[],[true|M],M,A2,A1),
    satisfy_clauses(Cn1,NC,M,A1,A).
satisfy_clauses(NC,NC,_,sat(M1),  A) :- true | A = sat(M1).
satisfy_clauses(NC,NC,M,cl_sat,   A) :- true | A = sat(M).
satisfy_clauses(NC,NC,M,unsat(Ms),A) :- true | A = unsat(Ms).

satisfy_ante(Cn,GS,[P|M2],M,cl_sat,A) :- true |
    satchmo_problem:c(Cn,P,GS,R),
    satisfy_ante1(Cn,R,P,GS,M2,M,A).
satisfy_ante(Cn,_,[],_,cl_sat,A) :- true | A=cl_sat.
otherwise.
satisfy_ante(_,_,_,_,A1,A) :- true | A=A1.

satisfy_ante1(Cn,fail,P,GS,M2,M,A) :- true |
    satisfy_ante(Cn,GS,M2,M,cl_sat,A).
satisfy_ante1(Cn,cont,P,GS,M2,M,A) :- true |
    satisfy_ante(Cn,[P|GS],M,M,cl_sat,A1),
    satisfy_ante(Cn,GS,M2,M,A1,A).
satisfy_ante1(Cn,false,P,GS,M2,M,A) :- true | A=unsat(M).
satisfy_ante1(Cn,R,P,GS,M2,M,A) :- R=[_|_] |
    satisfy_cnsq(R,R,M,A1),
    satisfy_ante(Cn,GS,M2,M,A1,A).

satisfy_cnsq([Fact|R2],R,M,A) :- true | check_cnsq(Fact,R2,R,M,M,A).
satisfy_cnsq([],R,M,A) :- true | extend_model(R,M,A).

check_cnsq(Fact,R2,R,[Fact|M2],M,A) :- true | A=cl_sat.
check_cnsq(Fact,R2,R,[],M,A) :- true | satisfy_cnsq(R2,R,M,A).
otherwise.
check_cnsq(Fact,R2,R,[_|M2],M,A) :- true | check_cnsq(Fact,R2,R,M2,M,A).

extend_model([Fact|R2],M,A) :- true |
    false([Fact|M],A1),
    extend_model(R2,M,A2),
    both(A1,A2,A).
extend_model([],M,A) :- true | A=unsat([]).

both(unsat(M1),unsat(M2),A) :- true | A=unsat([M1|M2]).
both(A1,_,A) :- A1=sat(_) | A=A1.
both(_,A2,A) :- A2=sat(_) | A=A2.
```

図3 KL1によるSATCHMOインタプリタ

1番めからNC番めまでの全ての節がcl_satを返した場合、問題の節集合はMで充足可能とわかるので、sat(M)を返す。

Cn節からsat(M1)が返されるのは、その節がモデルMの下で充足されていなくて、そのモデルMに可能な拡張を施して得られたある一つのモデルM1について節集合の充足可能がわかった場合(M1に対して起動された最下段のあるfalseプロセスがsat(M1)を返す場合)である。

unsat(Ms)が返されるのは2通りあって、Cn節がfalse節で、前件がMで充足された場合(Ms=M)、およびCn節がfalse節でなく、モデルMに可能な拡張を施して得ら

れた全てのモデルMsについて節集合の充足不能がわかった場合(Msに対して起動された最下段の各falseプロセスがunsat(Mi)を返す場合)である。

satisfy_ante(Cn,GS,[P|M2],M,A2,A)は、Cn節の前件中に出現するアトムを左から右へ順に調べ、Cn節の充足可能性の判定結果をAに返す。GSは、現時点までに既にMのいずれかの要素と照合に成功したアトムのリストである。satisfy_anteでは、今調べようとしているアトムとモデルMとの照合を行うため、ワーキング(探索中の)モデルリスト[P|M2]から先頭の要素Pを取り出し、Cn、PおよびGSに照合するKL1節cを呼び出す。cの呼び出し側のPおよびGSには変数は含まれないことに注意。また、このときcの定義節側に含まれる変数に対して自動的に新変数が確保される。

次にsatisfy_ante1(Cn,R,P,GS,M2,M,A)は、cの呼び出しの結果であるRの値に応じて以下の処理を進める。

(1) R=failの場合(現在のアトムが今のモデル要素Pで充足不能): satisfy_anteを再帰的に起動して残りのモデルM2と照合を試みる。

(2) R=contの場合:現在のアトムとモデル要素Pとの照合によって生じた変数束縛情報を以後の照合に伝搬させるため、GSに今のモデル要素Pを追加して、前件中の次のアトムに対するsatisfy_anteを起動する。それと同時に、残りのモデルM2に対して新たなsatisfy_anteを起動する。

(3) R=falseの場合(前件の最後のアトムが今のモデル要素Pで充足可能): unsat(M)を返す。

(4) R=[_|_](後件を表すアトムのリスト)の場合(前件の最後のアトムが今のモデル要素Pで充足可能): satisfy_cnsqを起動して後件の充足性を調べると同時に、残りのモデルM2に対して新たなsatisfy_anteを起動する。

satisfy_cnsqは、後件がfalseでない節でsatisfy_anteで前件がMの下で充足されたものについて、Mで後件が充足されているかどうかをcheck_cnsqによって調べる。充足されている場合は、その節全体がMで充足されているので結果cl_satを返す。そうでない場合は、extend_modelによって後件部の選言に現れる各アトムに対してそのアトムを一つだけMに追加したモデルを作り、拡張されたそれぞれのモデルについてfalseの再帰プロセスを起動する。これらのfalseプロセスの結果はbothによって合成されて親のfalseプロセスの結果を決定する。即ち、拡張されたそれぞれのモデルの下で、子供のfalseプロセスの結果が全て充足不能ならば、親のfalseプロセスはunsat(Ms)を返す。他方、拡張されたあるモデルの下で、子供のfalseプロセスの結果が充足可能ならば、親のfalseプロセスはsat(M1)を返す。

4. 性能評価

表1にPTTP、SATCHMO (Prolog版)、SATCHMO (KL1版)の性能比較を示す。S1～S3はいずれも有限領域を扱う問題であるが、これらについてはPTTPに比べてSATCHMO (Prolog版)がはるかに高速であることがわかる(問題S3についてはPTTPでは30分を経過しても解が得られなかった)。一方、SATCHMOのKL1版はProlog版に比べて3～4倍高速であることがわかる。ただし、問題S2についてはProlog版の方が2倍弱速い。これは、Prolog版においては図1に示したボトムアップなモデル生成手続きに加えて、falseゴールから逆向きに推論を行うことによって不要なモデル生成を回避するトップダウンの評価機構が組み込まれており、この効果が現れているためであろうと考えられる。

表1 性能評価

問題	S1	S2*	S3*
PTTP†	86msec	24sec	?(>30min)
SATCHMO in Prolog†	16msec	68msc	6.3sec
SATCHMO in KL1‡ (リダクション数)	5.4msec (390)	107msec (8,495)	1.5sec (118,237)

† (Sicstus Prolog V0.6 on SUN3/260)
‡ (pseudo-Multi-PSI single-PE on PSI-II)
* 問題S2, S3は付録を参照

ここで示したKL1版SATCHMOはインタプリタ方式によるものであったが、問題の節集合とその解法をKL1プログラムに直接ハンドコンパイルする方式[Fuchi 90]も可能である。我々のインタプリタ方式をこれと比較した場合、約3倍ほどの定数オーバヘッドに収まっていることがわかった。インタプリタ方式のオーバヘッドとしては驚くほど小さいものと思えるが、両方式ともコントロールは本質的に同じである他、我々の方式でも既に問題の節をKL1節に変換していることを考え合わせれば、妥当な数字であるといえよう。また、この程度のオーバヘッドであれば部分計算技法を適用することによって直接ハンドコンパイル方式に匹敵する性能を得ることが可能であろう。

5. 議論

PTTPや、レゾリューションに基づくフルファーストオーダーの定理証明システムにはオカーチェック付きユニフィケーションが必要である。

オカーチェック付きユニフィケーションをKL1でプログラムするとき、論理式中の変数を例えばvar(x)のような基底項を用いて表す。また、ユニフィケーションゴールunify(X,Y,S)はXとYをユニファイした結果をSに返すようなものとする。Sの値はユニファイに失敗した場合はfailで、成功した場合は変数の束縛のリストである。このようにして書いたunify(X,Y,S)の実行時間はKL1の組み込みユニフィケーションX=Yの場合の150～200倍ほどであった。このようにオーバヘッドが大きいとやはり実用には耐え難いといわざるを得ない。従って、現在のKL1の機能ではPTTPやレゾリューション方式よりもSATCHMOの方が実現が容易な証明方式であると思われる。

図3のSATCHMOインタプリタを基本型として、幾つかの拡張が考えられる。

(1) OR並列

まず、選言を後件にもつ節によって拡張されるモデルを同時に処理するOR並列処理を考えることができる。この場合、資源はあくまで有限であること、分岐の数が大きいような問題ではたちまち資源が食い潰されることなどを考慮して、資源数に見合うだけOR並列展開し、それ以後はAND並列処理を行う(restricted-OR)など、何らかの制限戦略が必要になるであろう。

(2) クローズインデクシング

問題の節集合が大きくなるにつれ、節の選択に要するコストが増大する。図3に示したインタプリタでは、falseの再帰プロセスが起動される度にいつもNC本の節全部を調べるが、モデルを拡張した節の後件と他の節の前件のコネクションを考えれ

ば、この再帰プロセスで調べるべき節の数を絞ることができる。このような修正は構文的解析に基づいて比較的容易に行うことができる。また、PrologやKL1処理系で行っているようなクローズインデクシングも自動的に利用できる。

(3) モデル拡張ステップの縮約

また、今のモデルに対して充足されておらずモデルの拡張を要する節が複数ある場合、その全ての拡張の組み合わせを1ステップで計算することにより、ステップ数を縮めることが考えられる。例えば、次の問題を考えよう。

```
(C0)   true ---> p(a).
(C1)   p(X) ---> r1(X).
       ...
(Cn)   p(X) ---> rn(X).
(Cx)   r1(X), r2(X), ..., rn(X) ---> false.
```

この問題に図1の手続きを適用すると、n+2ステップ要する。しかし、C0でモデル{p(a)}が生成された後、C1からCnまでがモデル拡張の候補となっていることがわかる。そこで、r1(a)からrn(a)までを一度にモデルに加えることができて、この場合3ステップで終了する。このようにして、一つのモデル生成の失敗が即座にわかる可能性がある。

しかし、一般には後件が幾つかのアトムの選言であるから、モデル拡張の一つの候補はそれぞれの節の選言から一つずつ選んだものとなり、この組み合わせのための計算量の増大がステップ数の短縮に見合わない程大きい場合もあり得る。

(4) モデルの共有

他に、SATCHMOのモデル拡張のステップにおいて、選ばれた節が実はホーン節で、生成されるモデル要素はすべてのモデルに共通である場合がある。この場合、共通に参照できるモデル部分は、選言を後件にもつ節によって拡張される他のモデルとは別の領域に置くのがよい。こうして領域を節約することが可能である。

(5) 消去戦略その他

さらに、レゾリューションにおけるファクタリングやサブサンプションに相当する消去戦略を導入して計算の重複を減らすことが考えられる。その際、余分な記憶領域と照合の手間が導入されるが、これが計算の重複の削減に見合うかどうかは問題に大きく依存している。実際的なプルーバーはこれらの戦略をパラメタ化されたオプションとして装備し、これらのパラメタの選択は結局ユーザの試行錯誤の上決定されるものと考えることになるであろう。(適正パラメタの抽出にニューロネットの手法に基づく学習を適用するという研究[Ertel 90]がある。)

6. 結論

SATCHMOがユニフィケーション(+オカーチェック)を必要としないことはKL1でインプリメントするのに極めて好都合であった。問題の中の変数はKL1変数で表現し、その束縛はKL1のヘッドユニフィケーション(即ち照合)で行うことができるからである。また、節のコピー(異なる節インスタンスを作ること)とそれに付随する新変数の導入(名前替え)は、KL1節の呼びだしの機構によって自動的に行われる。この利点を活用するためには問題の節を適切なKL1の節集合に変換する必要があるが、この変換は極めて機械的にかつ安価に行えるので障害にはならない。

SATCHMOは、有限領域については極めて効率のよい証明システムであり、ここで開発したKL1プログラミング技法は、データベース、TMS等の有限領域を扱う様々

な応用の実現技術として有用である。一方、無限領域についてはdom述語を導入して対処することになるが、これはそのままでは全くのブルートフォース的解決策である。ある種のヒューリスティックスを入れて探索空間を適正に抑えない限り、たちまち組み合わせ的爆発のために資源を消費し尽くしてしまい、証明結果を得るまでには至らない。一つの解決策としてノングランドのモデルを許すことにし、レゾリューション法におけるサブサンプションに相当する消去戦略を導入することを現在試みているところである。

今後、レゾリューション法、コネクション法などについても検討を進めて行く予定であるが、そこでもやはり証明対象となる論理式に現れる変数の扱い方がキーになるであろう。即ち、高速な定理証明システムの実現可能性はSATCHMOのようにフルユニフィケーションからうまく逃れ得るか、あるいは効率のよいフルユニフィケーションがKL1コーディングで実現され得るかにかかっていると思われる。しかし、もしKL1処理系側で基底項表現された変数とそれを含む項の間の効率のよいフルユニフィケーションが組み込みないしユーティリティとしてサポートされれば、この問題は一挙に解決するに違いない。

謝辞

本研究は、PTTPならびにSATCHMOのPrologにおける成果に端を発している。古川次長にはこれらのシステムの紹介とKL1版インタプリタ作成の機会を与えて頂き、様々な御助言を頂いた。渕所長には本研究の当初から数々の貴重な示唆を頂いた。実際、ここで報告したKL1版SATCHMOインタプリタの作成技法は、渕所長の開発されたSATCHMO問題および解法をESPやKL1プログラムへハンドコンパイルする技法に負う所が多い。また、藤田正幸氏には他の定理証明技法との比較検討等に関する有意義な議論を頂いた。最後に御指導、御討論頂いた上記の方々に謝意を表します。

参考文献

[Bibel 86] Wolfgang Bibel, Automated Theorem Proving, Vieweg, 1986.

[Wos 88] Larry Wos, Automated Reasoning – 33 Basic Research Problems –, Prentice-Hall, 1988.

[Stickel 88] Mark E. Stickel, A Prolog Technology Theorem Prover: Implementation by an Extended Prolog Compiler, in *Journal of Automated Reasoning* 4 pp.353-380, 1988.

[Bry 88] Rainer manthey and François Bry, SATCHMO: a theorem prover implemented in Prolog, in *Proc. of CADE 88, Argonne, illinois,* 1988.

[Fuchi 90] Kazuhiro Fuchi, experimental programs in ESP/KL1 for a SATCHMO prover, 1990.

[Ertel 90] Christian Suttner and Wolfgang Ertel, Using Connectionist Networks for Guiding the Search of a Theorem, TUM-19011 SFB-Bericht Nr.342/8/90 A, Institute fur Informatik, Technische Universitat Munchen, 1990.

付録: SATCHMO(KL1 版) による問題記述 (S2, S3)

```
%       S2 : Shubert's Steamroller problem

:- module satchmo_problem.  :- public model/1, nc/1, c/4.
model(M) :- true | M =
    [likes(s,i(s)),likes(c,h(c)),likes(b,c),                  %  likes(s,i(s)).
                                                              %  likes(c,h(c)).
                                                              %  likes(b,c).
     smaller(f,w),smaller(b,f),smaller(s,b),smaller(c,b),     %  smaller(f,w).
                                                              %  smaller(b,f).
                                                              %  smaller(s,b).
                                                              %  smaller(c,b).
     plant(g),plant(i(s)),plant(h(c)),                        %  plant(g).
                                                              %  plant(i(s)).
                                                              %  plant(h(c)).
     animal(w),animal(f),animal(b),animal(s),animal(c)].      %  animal(w).
                                                              %  animal(f).
                                                              %  animal(b).
                                                              %  animal(s).
                                                              %  animal(c).

nc(NC) :- true | NC=6.
c(1,likes(w,f),[],R) :- true | R=false.     %  likes(w,f) ---> false.
c(2,likes(w,g),[],R) :- true | R=false.     %  likes(w,g) ---> false.
c(3,likes(b,s),[],R) :- true | R=false.     %  likes(b,s) ---> false.
c(4,animal(X), [],R) :- true | R=cont.                        %  animal(X),
c(4,animal(Y), [animal(X)],R) :- true | R=cont.               %  animal(Y),
c(4,likes(X,Y),[animal(Y),animal(X)],R) :- true | R=cont.     %  likes(X,Y),
c(4,likes(Y,g),[likes(X,Y),animal(Y),animal(X)],R) :-         %  likes(Y,g)
        true | R=false.                                       %  ---> false.
c(5,animal(X),    [],R) :- true | R=cont.                     %  animal(X),
c(5,animal(Y),    [animal(X)],R) :- true | R=cont.            %  animal(Y),
c(5,smaller(Y,X),[animal(Y),animal(X)],R) :- true | R=cont.   %  smaller(Y,X)
c(5,plant(W),     [smaller(Y,X),animal(Y),animal(X)],R) :-    %  plant(W),
        true | R=cont.
c(5,likes(Y,W),   [plant(W),smaller(Y,X),animal(Y),animal(X)],R) :-
        true | R=cont.                                        %  likes(Y,W),
c(5,plant(Z),     [likes(Y,W),plant(W),smaller(Y,X),animal(Y),animal(X)],R) :-
        true | R=[likes(X,Y),likes(X,Z)].                     %  plant(Z)
                                          %  ---> likes(X,Y) ; likes(X,Z).
otherwise.
c(_,_,_,R) :- true | R=fail.
```

```
%       S3 : Pelletier & Rudnicki's problem

:- module satchmo_problem.  :- public model/1, nc/1, c/4.
model(M) :- true | M = [dom(a),dom(b),dom(c),dom(d)].   %  dom(a).
                                                        %  dom(b).
                                                        %  dom(c).
                                                        %  dom(d).

nc(NC) :- true | NC = 6.
c(1,p(a,b),[],R) :- true | R=false.          %  p(a,b) ---> false.
c(2,q(c,d),[],R) :- true | R=false.          %  q(c,d) ---> false.
c(3,p(X,Y),[],R) :- true | R=cont.           %  p(X,Y),
c(3,p(Y,Z),[p(X,Y)],R) :- true | R=[p(X,Z)]. %          p(Y,Z) ---> p(X,Z).
c(4,q(X,Y),[],R) :- true | R=cont.           %  q(X,Y),
c(4,q(Y,Z),[q(X,Y)],R) :- true | R=[q(X,Z)]. %          q(Y,Z) ---> q(X,Z).
c(5,p(X,Y),[],R) :- true | R=[p(Y,X)].       %  p(X,Y) ---> p(Y,X).
c(6,dom(X),[],R) :- true | R=cont.           %  dom(X),
c(6,dom(Y),[dom(X)],R) :- true |             %          dom(Y)
        R=[p(X,Y),q(X,Y)].                   %           ---> p(X,Y) ; q(X,Y).
otherwise.
c(_,_,_,R) :- true | R=fail.
```

並列推論マシン上のＬＳＩレイアウトシステム co-ＨＬＥＸ の概要

（株）日立製作所 システム開発研究所
第５部 渡辺 俊典, 小松 啓子
☎ (044)966-9111

ＩＣＯＴ後期研究で開発される予定の大規模並列推論マシンの機能実証ソフトウェアの一環としてＬＳＩレイアウトシステム co-ＨＬＥＸ を開発している。co-ＨＬＥＸ の現況につき概要を紹介する。co-ＨＬＥＸ のシステム構成、その基礎を成す階層再帰協調算法 ＨＲＣＴＬ の概要、特徴、負荷分散法、実験結果等を紹介する。ＫＬ１ 使用経験についても簡単に紹介する。

１．はじめに

ＩＣＯＴの活動は、前期、中期、後期に分けられ、中期において主に逐次形推論マシンの開発、後期において並列推論マシンの開発が実施されている。開発されるハードウェアの計算パワーは、並列動作するＰＥ（プロセッサエレメント）数でみて後期は中期までの１０^{2}～１０^{3}倍となる。ソフトウェア的には、逐次処理から並列処理へ重点が移されている。

本研究は、これらの研究成果のレイアウト問題解決、より広くは設計ＣＡＤへの適用性を実証する目的で実施してきており、推論マシンや推論ソフトウェアの計算力や表現力に相応する対象を取り上げて下記の様に開発をすすめている。

（Ｔ．１）前・中期： 計算機室のレイアウト問題を取り上げ、１０～１０^{2}部品から成る小中規模のレイアウトを逐次型推論マシン（ＰＳＩ）によって実現した。ＥＳＰ言語のレイアウト問題の表現力や求解力及びオブジェクト指向性によるプログラムの開発、変更の容易性等を実証した。並列処理については、擬似並列処理レベルでの実証に留めた。

（Ｔ．２）後期： 電子回路のレイアウト問題を取り上げ、１０^{2}～１０^{3}部品からなる回路のレイアウトを並列推論マシン（マルチＰＳＩ）によって実現する。計算パワーの大きさに対応した問題規模とするために電子回路を対象とすることとし、かつ並列処理の本格的実現のために並列レイアウトアルゴリズムを新たに開発する。

後期の初年度である平成元年度においては、ＩＣＯＴ後期研究における並列推論マシンや並列論理型言語等の基本技術のレイアウト問題解決への適合性検証を部分的に実施し、以降の年度での本格的実証の基礎作りを行うことを目的として、下記を実施した。

並列論理型言語による電子回路レイアウト問題解決システムの試作。

並列論理型言語による汎用レイアウト問題解決機構の方式検討。

前者については、並列論理型言語ＫＬ１を用いた電子回路レイアウトシステムの第１版をマルチＰＳＩ上に試作し、電子回路を実験対象として、並列処理による配置・配線の基本機能を確認した。後者については、前者を開発する内で併せて検討し、回路系のレイアウトという枠内で前者が汎用なものとなるよう、前者の基本アルゴリズムやソフトウェア構成に反映させた。すなわち、本システムはアルゴリズム的には階層レイアウト方式を採用しており、配置・配線問題双方を階層処理する。階層処理全体を制御する問題解決基本機能（①）と、階層の非終端、終端部において実質的にレイアウトを生成する上、下位レイアウト機能から成るレイアウト基本機能（②）、入出力機能（③）とが基本構成要素である。②内の下位レイアウト機能の一部である回路素子の幾何形状ライブラリの差し替えによって各種回路系に汎用的に適用し得る可能性を有する。

２．開発現況

２．１　co-ＨＬＥＸの構成と基本アルゴリズム　ＨＲＣＴＬ

マルチＰＳＩ上に並列論理言語ＫＬ１を用いた電子回路レイアウト問題解決システム第１版（cc-ＨＬＥＸ：Co-operative Hierarchical Layout EXpert）を実現した。本システムの構成と動作概要を述べる。図２－１参照。

（Ｓ．１）原データ：　本システムへの最初の入力として回路のネットデータすなわち素子（以下オブジェクトと呼ぶ）と素子間の接続関係（以下ノード情報と呼ぶ）、及びオブジェクトとノードの配置・配線上の制約条件のリストから成る。

（Ｓ．２）状態メモリ：　レイアウト対象回路を階層化したもの（回路木）や、これを平面上に配置・配線した結果の階層化記述（レイアウト木）、配線用コネクタなどが記憶される。高速アクセスを保障するために、ＫＬ１のハッシュプールで実現している。

（Ｓ．３）入出力プロセス：　入力機能は原データ内の非階層回路ネットを回路木に変換して状態メモリに記憶する。出力機能は状態メモリ内のレイアウト木等の情報を出力描画する。

（Ｓ．４）問題解決推論プロセス：　回路木をレイアウト木に変換処理することによってレイアウトを生成する。配置処理のあと配線処理を行う。配置処理においては配線の概略を計画し、配線を無視した配置とならぬようにする。回路木に配置データが追加されレイアウト木が作られる。配線処理では、このレイアウト木に配線データを追加する。配置・配線いずれの処理も、回路及びレイアウト木の非終端部を下降方向にたどる問題分割フェイズ、木の終端部でのセル処理フェイズ、木を上昇方向にたどる統合フェイズを備えた階層問題解決を行う。両木とも木の非終端ノードはブロック、終端ノードはセルと呼んでいる。この部分を、その処理の特徴から ＨＲＣＴＬ(Hierarchical Recursive Concurrent

Theorem prover for Layout)と略称している。

各フェイズでの処理の主内容を以下にまとめる。

（Ｓ．４．１）配置・問題分割フェイズ・ブロックレベル：　回路木の親ノードに対応する回路の平面形状（レイアウト木中に記入してある）を子ノード数だけの部分平面に予定的に分割し、各分画に子ノードを割り当てる。結果はレイアウト木に記入する。木の終端に至るまで同様に再帰処理する。最上位ノードについては予定形状が与えられている。再帰において、各子ノードは並列に処理される。

（Ｓ．４．２）配置・問題分割・統合フェイズ・セルレベル：　回路木の終端ノードに対応する回路のレイアウト形状を、後述するレイアウト基本プロセスより呼び出してレイアウト木の終端ノードに記入する。

（Ｓ．４．３）配置・統合フェイズ・ブロックレベル：　レイアウト木の子ノードの平面形状を統合して親ノードの平面形状を作成し、親ノードに記入する。木の上端に至るまで同様に再帰する。

（Ｓ．４．４）配線・問題分割フェイズ・ブロックレベル：　レイアウト木の親ノードの平面形状の分割区間内に配置された子ノード間の配線計画を行う。配線が各分画境界を通過する点にコネクタを作成し、レイアウトデータの一部として記憶すると共に、子ノードにコネクタ名称を知らせる。各子ノードは、与えられた周辺コネクタを用いて、互いに並列に同様の再帰処理を行う。最上位ノードにおいては外周コネクタの予定位置があらかじめ与えられている。

（Ｓ．４．５）配線・問題分割・統合フェイズ・セルレベル：　各セルは、ブロックレベル処理によって与えられた周辺コネクタと自己の内部コネクタとの間を配線し、配線データをレイアウトデータの一部として記憶する。内部コネクタを分画の一つとする詳細さにまでレイアウト木を伸長させ（局所レイアウト木）、局所レイアウト木に対してブロックレベルと類似の問題分割処理を行う。統合フェイズでは区画内配線データの生成を行う。

（Ｓ．４．６）配線・統合フェイズ・ブロックレベル：　レイアウト木の子ノード間の配線状況を親ノードで解析し、特性面で不具合があれば改良する。

（Ｓ．５）レイアウト基本プロセス：　問題解決推論プロセスが使用する各種のプログラム節から成る。配線幅、間隔等の制限事項等のプロセス制約を記述したレイアウトルール、ブロックあるいはセルレベルでの配置形状や配線形状のテンプレート群であるレイアウトフレーム、及び形状テンプレートを用いてレイアウト木の各ノードにおける問題分割や統合処理を行うプランフレーム群とから成る。

（Ｓ．６）双方向ストリーム：　状態メモリと各プロセスとの間、及び問題解決推論プロセスとレイアウト基本プロセス間に設けられた通信路である。双方向とはデータの送受信双方が可能であることを示す。

試作システムによるレイアウト実験を実施し、部分問題間の協調を行いつつ並列モード

で配置配線が可能であることを確認した。

２．２　co-ＨＬＥＸ の特徴

周知の如く、ＬＳＩ開発において、レイアウト自動化は古くから重要な問題であり、数多くの方式が開発、実用化されてきた。従来技術に対する本方式の特徴を次にまとめる。

（Ｆ．１）レイアウト方式としての特徴：　著名なものとして、配置・配線が不要であるＰＬＡ、配線のみを必要とするゲートアレイ、固定かつ形状のほぼ揃った部品の配置・配線を行うスタンダードセル、異形状物を配置・配線するビルディング・ブロック方式等がある。本方式はこの最後の方式に属する。

（Ｆ．２）複雑度の処理能力：　階層処理を実施しているため処理可能な問題サイズについては従来と同等である。多くの並列プロセスの走行を保障するために大規模なメモリや並列プロセッサが必要となるが、応分の処理スピード向上は期待できる。レイアウト面積の縮小、配線率の向上、配置・配線における電気特性の考慮等の問題については、問題分割フェイズに種々の制約を考慮する機能を、統合フェイズに不具合改良機能を各々設けることで対応できる能力を内包している。

（Ｆ．３）汎用性：　レイアウトルールやレイアウトフレームの差し替えによって原理的には種々のデバイスやプロセスに適用できるソフトウェア構成となっている。ＫＬ１言語が元来持っているモジュラリティも、このことを容易化してくれる。

（Ｆ．４）ソフトウェア開発性：　回路木をレイアウト木に変換する上記の過程は、再帰アルゴリズムとして実現している。同一のプログラムの繰り返し利用によって大規模問題を処理しているため、レイアウトプログラムの規模は、従来の方式（１０^{4}～１０^{5}オーダ）に比べて小さい（１０^{3}～１０^{4}オーダ）。

（Ｆ．５）並列処理と逐次処理：　大規模問題解決においては、古来からの分割統治原理に従って階層処理が行われる。階層処理においては本来ならば分割せずに解きたい原問題（本例では原回路ネットリスト）を無理に分割処理する。分割結果生じる各部分問題が相互関係の調整（本例では隣り合う子回路のレイアウト形状の整合や、子回路間境界線上の配線コネクタ位置の整合）を要する場合が一般的である。逐次処理によってこの問題を解く場合には、禁止的な時間を要するバックトラックによるか、収束計算によって解を漸近的に改良する方式をとる。並列処理によって解く場合には、各プロセスに、変数の値を操作するための情報が充分に集まるまで待つという遅延機能を持たせることで効率的に解を求め得る可能性がある。ＫＬ１言語にはこの機能が備えてあり、階層処理を部分問題間の相互関係を自動調整しつつ並列モードで高速処理することが可能である。この機能を持たない通常の並列処理では、部分問題間でのコミュニケーション不足のためにブロック形状や境界上の配線コネクタ位置のずれのため、ブロック間にデッドスペースや配線用チャネルが必要となり、その分だけチップ面積が大きくなってしまう。他方、並列処理のデメリットとしては、莫大なメモリーや計算パワーを必要とする点が否定できない。

並列推論マシン（マルチＰＳＩ）上に、ＫＬ１言語を用いて以上の特徴を有するレイアウトシステムの第１版を試作し、並列処理動作を実験を通じて確認できたことにより、並列推論マシンのレイアウト問題解決への適合性の検証を部分的に実施でき、次ステップの足がかりを得たと考える。

２．３　co-ＨＬＥＸ における負荷分散法の概要

ＨＲＣＴＬ はチップの再帰的分割操作をその基本としている。配置物である素子やそれらの配線はもともと、できるだけすき間なくチップ上にレイアウトされる。よって、チップ上の任意の区画を取った場合、区画面積が等しければほぼ同一オーダの素子や配線を含むと期待できる。言い換えると配置・配線の手間（プロセス量やそのライフタイム）は同一オーダとなると期待できる。この視点に立って、ＨＲＣＴＬ で木のノード処理を行う際に、「親ノードにおいて、上位より伝えられた並列稼働可能なＰＥの集合（Matrix）を子ノード個数分に面積依存で分割し（[SM1|Tail]）、各部分マトリクス上の代表格子点（左上端点部）のＰＥに、該当する子ノードを割り付けると共に、各子ノードに個々の部分マトリクスを伝達する」という負荷分散方式を用いることとした。Matrix が分割不能になって以降は、分割は自動抑止され、子ノードは全て同一ＰＥ上に割りつけられる。すなわち、各子供は擬似並列モードで処理される。以上のように ＨＲＣＴＬ での現在負荷分散方式は、チップの平面構造をＰＥ格子（Matrix）上にマッピングするという単純なものであるが、前述した理由から、各時点でのＰＥへの負荷は最適配分される傾向を持つ（各ＰＥはほぼ足並みを揃えて多忙となり、足並みを揃えて遊休となる）。co-ＨＬＥＸ の現在のプログラムアーキテクチャは集中メモリ系である。図２－１の双方向ストリームバス線上の通信ネックの発生が本負荷分散法の効率を低下させることも予想される。この問題への対応は今後の課題の一つと考えている。

２．４　実験結果

（１）実験内容

簡単な回路を用いてco-ＨＬＥＸ試作第１版の基本機能を確認した。

（Ｅ．１）使用回路：　図２－２（Ａ）に示す、素子数１６個のバイポーラアナログ回路（tr=８個、res=８個、capa=０個、信号入出力端子=各２個、vcc,gnd端子各１個）。本回路は入力（in1,in2）を差動アンプ（q1,q2系）で増幅し、端子（out1,out2）に出力する。電圧源（q5,r1,r5）はダイオードq5の固定ベース電位をq6,q7,q8等のベースに伝えて、それらのコレクタ電流を一定化し、回路の動作点を固定化する。差動アンプは電圧入力電圧出力増幅器、出力tr（q3,q4）はレベルシフタであり、継続段の差動アンプのベース入力に直接接続できるよう電圧レベルを下降させている。

（Ｅ．２）レイアウト時の考慮制約：　電子回路の並列処理による配置配線機能を実現することが第１次試作の目標であるため、各素子の形状はほぼ同一化した。配置配線時のペア条件、素子のアイソレーション、配線層、レイアウトルール等については、次の条件の

み考慮した。
配置：　素子間の配線長をできるだけ短くすると共に、面積を少なくする。
配線：　ネットの各端点（コネクタ）を単に結線する。異ネットがショートしてもかまわない。配線層も特に考慮しない。電源配線と信号配線の引き回し方の区別もしない。
（Ｅ．３）入力データ作成：　図２－２（Ａ）のアンプ回路を人手で階層化して回路木とした。階層図を図中（Ｂ）に示す。
（Ｅ．４）確認事項：
配置機能：　並列処理によって配置が作成されること。制約条件として考慮した配線長が短くなること。面積の少ない配置が得られること。
配線機能：　並列処理によって配線が作成されること。ネット端点が接続されること。
（２）実験結果
　チップの左辺に入力コネクタ、右辺に出力コネクタを指定して自動レイアウトした結果が図中（Ｃ－１）に、入出力コネクタをこれと左右逆にした時の結果を（Ｃ－２）に示す。マルチＰＳＩ上で約２０秒を要した。
（３）実験結果の評価
　全体動作：並列処理によって配置・配線が実現できることが確認できた。但し、配線時の部分問題間でのコネクタ位置整合処理時に、デッドロックが発生する場合がある。
　配置機能：素子形状がほぼ同一である回路を、デッドスペース無く配置できたことで、面積縮小機能が確認できた。配線長をできるだけ短くする配置を作る機能は、入出力端子位置変更に対して素子配置が自動的に適応したことで確認できた。
　配線機能：一部の配線が混み合っているケースもあるが、配線すべきコネクタ間をできるだけ直線で結ぶ機能があることが目視確認できる。内部では階層的に分割して処理しているにもかかわらず素子間の配線が不要に屈曲せず、従って配線専用のチャネルを必要としていない。このことからＡＮＤ並列による協調機能が確認できた。

３．ＫＬ１の使用経験と要望事項

　システム開発の諸相でのＫＬ１使用経験と要望事項を簡単にまとめる。
（１）機能設計フェイズ：本格的な並列処理記述が可能であり、新算法の実現に有用であると考えられる。開発中のシステムにおいても、階層異ノード間の配線端点の相互調整等、従来は実現不可能であった機能が一種の並列緩和算法によって実現し得た。但し、複数個の条件を満たす解をバックトラックによって簡単に取り出すことが逐次Ｐｒｏｌｏｇほど簡単に実現できない点は不便である。
（２）詳細設計フェイズ：ＫＬ１はフラットＧＨＣであり、ユーザ定義のガード述語を自由に記述できない。このことがＰｒｏｌｏｇの最大の特徴であった仕様記述性を低下させている。せめて、ガード部に既定義組込述語の論理式を記述できるよう拡張すべきである。

（３）プログラミング ＆ デバッグフェイズ：現在のＫＬ１コンパイラでは、述語の引数サイズにかなり強い制限があり、コンパイラを通すために、述語をわざわざ分割する必要が生じる。この点は改良すべきである。走行時ガベージコレクションを目的としてポインタの白黒管理がＫＬ１では行われている。現状では白黒管理はプログラマ責任であるが、コンパイラ側でやるとか、せめてアラーム出力する機能が欲しい。
（４）実行フェイズ：ＫＬ１の現状ではグラフィックスのようなマンマシン系が弱い。充実が望まれる。

４．まとめ

並列推論マシン（マルチＰＳＩ及びＰＩＭ）の機能実証用ソフトウェアの一環として開発中のＬＳＩレイアウトシステム co-ＨＬＥＸ とその中核アルゴリズム ＨＲＣＴＬ、及びＫＬ１利用経験や要望事項について簡単に紹介した。本システムはＨ０１年の８月よりマルチＰＳＩ上にインプリメントを開始したものであり、著者等にとっても並列処理プログラミングの経験は浅いが、残された２年余の中で、co-ＨＬＥＸ の機能・性能両面の改良を通じて並列推論の今後の可能性をできるだけ深く見極めてみたいと思っている。

謝辞

研究推進にあたり、日頃御指導いただくＩＣＯＴ ５研 生駒室長、４研 瀧主任研究員をはじめ、関連する ＫＡＳ、ＰＩＣ ワーキンググループの方々に、また、研究開発機会を与えていただいている当社システム開発研究所、石原孝一郎主管研究員、森文彦第５部長に深謝します。

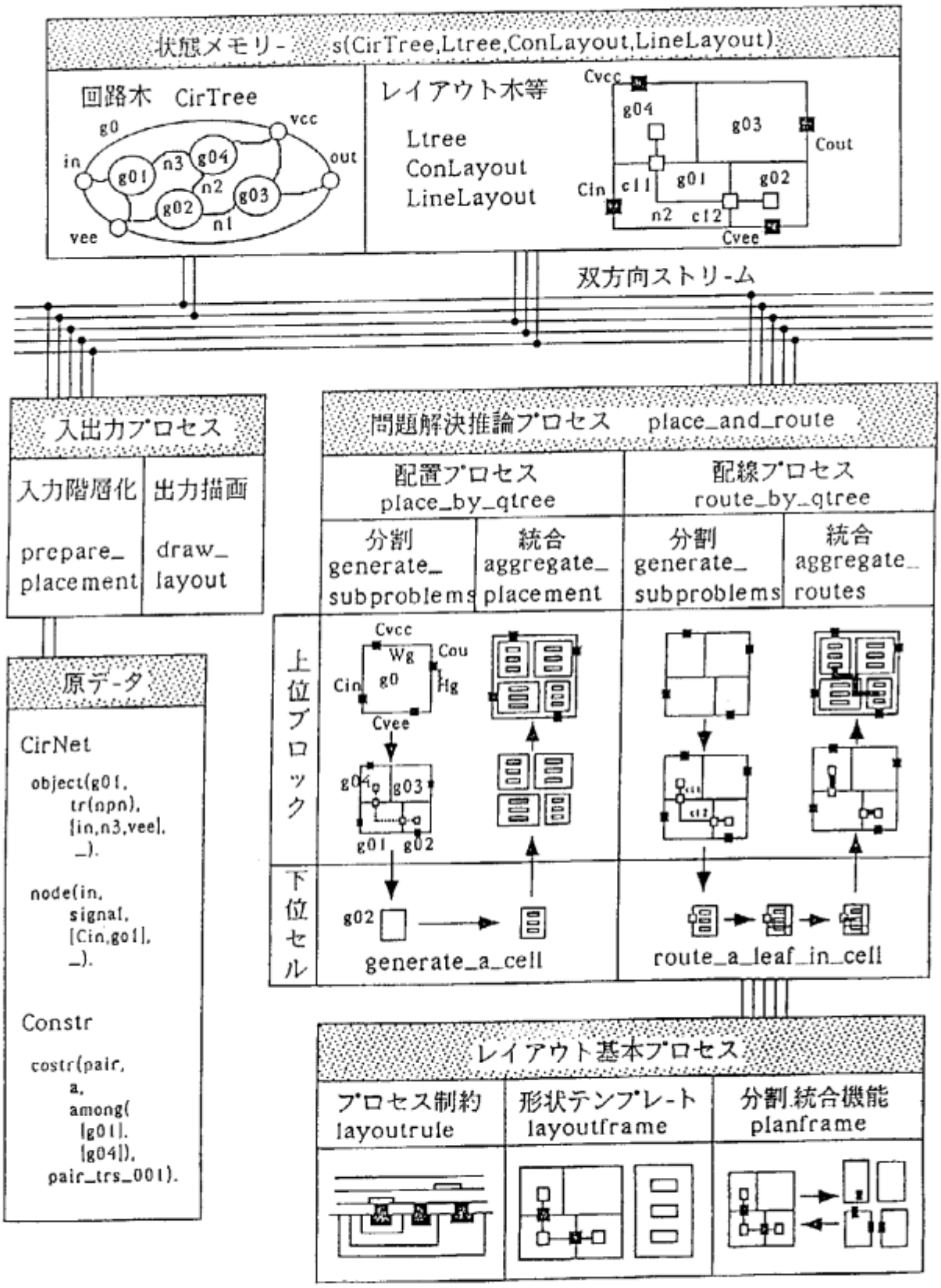
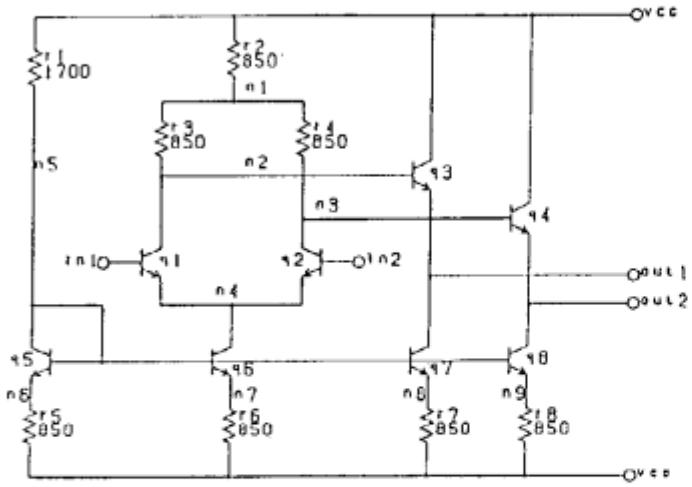
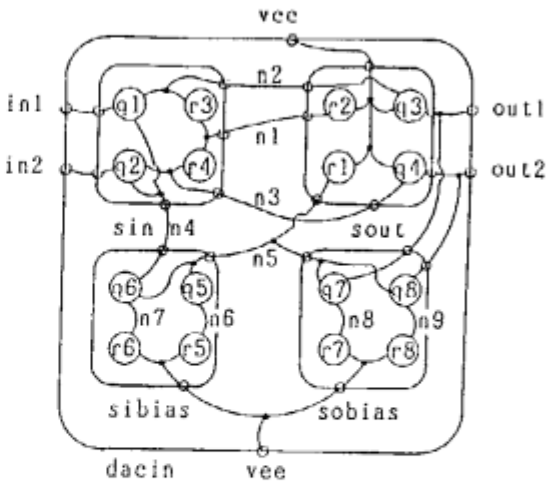


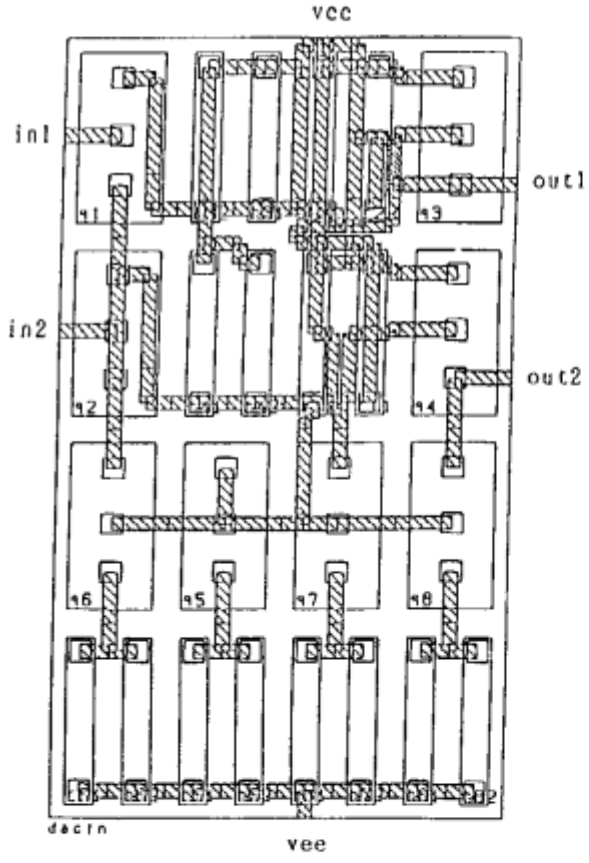
図２－１　並列レイアウトシステム（co-ＨＬＥＸ）の構成



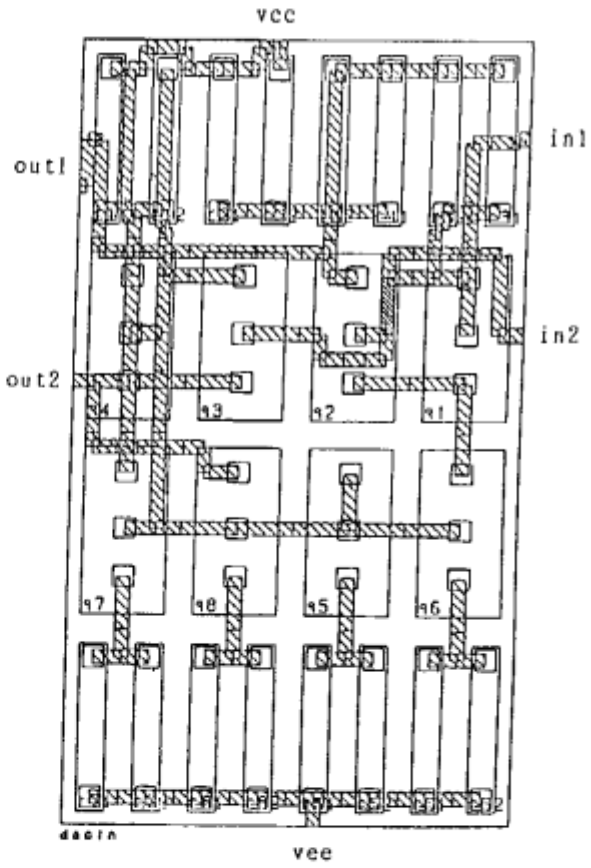
（A）入力回路ネットA



（B）Aの回路木



（C－1）レイアウト図（左辺入力）



（C－2）レイアウト図（右辺入力）

図2－2　実験結果

KL1による帰納的学習システムの構築

坂本忠昭　高橋和子　竹内彰一
三菱電機(株) 中央研究所

1 はじめに

筆者らは、並列問題解決システムANDOR-II [Takeuchi 88][Takeuchi 89]の開発を進めているが、このANDOR-IIの応用問題の1つとして、帰納的学習システムの構築を取り上げた。帰納的学習システムとは、外部から与えられたデータや例をもとに一般化・特殊化等の帰納推論を行なうことによって、入力された全ての例を満足し、かつ、新しい入力例を正しく判定できる一般的記述を得るシステムである。帰納的学習方法としては現在まで数多くのものが提案されているが、本システムでは、帰納的学習システムの1つであるCIGOL [Muggleton 88]が用いている学習方法を採用した。CIGOLは一階述語を獲得対象とした対話型の帰納的学習システムである。CIGOLで用いられている学習方法は、教師から与えられた例に対してtruncation, absorption, intra-constructionと呼ばれる3つのオペレーションを施すことによって一般的記述を求めるものである。ここでは、ANDOR-IIによる実現に先立ち機能検討を目的として行なった、KL1による帰納的学習システムの構築について述べる。

以下、第2章でシステムの構成を示し、第3章で3つの基本オペレーションを含めた学習アルゴリズムについて述べる。続いて第4章で現在のシステムの評価及び検討を行なう。

2 システム構成

図2-1に帰納的学習システムの構成を示す。システムは大きく分けて制御部・帰納推論部・インタフェース部の3つのモジュールから構成される。

以下、各モジュールについて説明する。

1. 制御部
   制御部はシステム全体の動作を制御するモジュールであり、大きく2つの働きを持つ。1つは、帰納推論部の3つの一般化オペレーション及び定理証明器を制御することにより、学習アルゴリズムを実行することである。帰納推論部の4つのサブモジュールは各々独立しており、学習の際のそれらの制御は全て制御部で行なわれている。制御部の第2の働きは、インタフェース部の制御すなわち入出力管理である。これによって、具体例の入力・学習結果の出力・ユーザとの対話等を行なうことができる。

2. 帰納推論部
   帰納推論部は帰納的学習に用いられる3つの基本オペレーション及び定理証明器からなるサブモジュール群である。truncationモジュール・absorptionモジュール・intra-constructionモジュールはそれぞれ各オペレーションを実行するサブモジュー

ルであり、定理証明モジュールは節集合から冗長な節を除くために用いられるサブモジュールである。

3. インタフェース部

インタフェース部は、ウインドウ管理モジュールとファイル管理モジュールの2つのサブモジュールから構成される。各サブモジュールは各々ウインドウ、ファイルの生成及びそれに対する入出力管理を行なう。

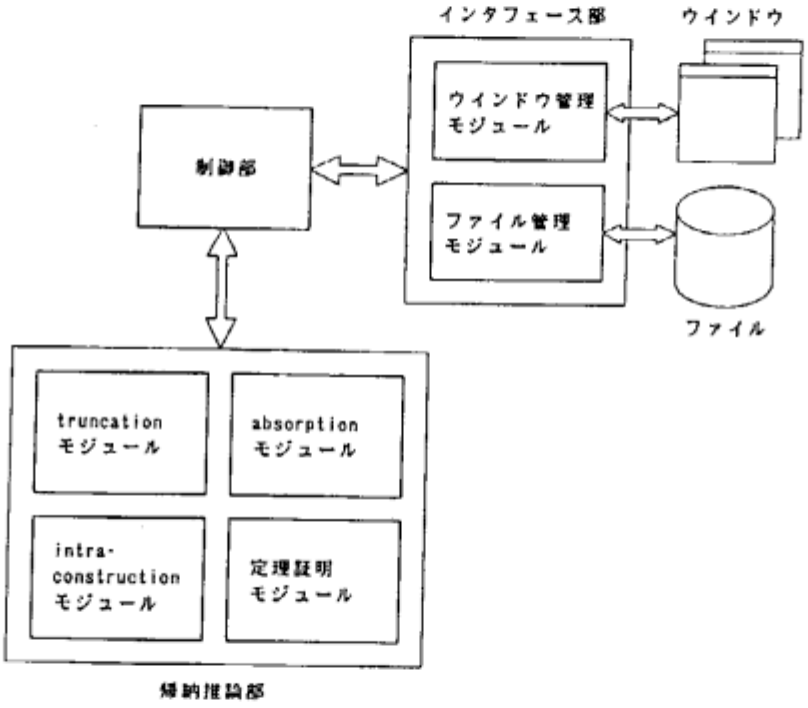


図 2-1 帰納的学習システムの構成

3 学習アルゴリズム

3.1 基本オペレーション

ここでは、システムが学習の際に用いている3つの一般化オペレーション truncation, absorption, intra-construction について簡単に説明する。

3.1.1 truncation

truncation とは、単一節 $C_1, C_2, ...C_m$ が与えられたとき、それら全てを満足する単一節 C を求めるオペレーションである。実際には、単一節 C は $C_1, C_2, ...C_m$ の最小一般化によって求められる。truncation の例を以下に示す。

C_1 = member(blue, [blue]).
C_2 = member(eye, [eye, nose, throat]).
⇓
C = member(A, [A|B]).

3.1.2 absorption

2つの節 C_1, C_2 から節 C が導出されるという導出ステップにおいて、まず C_1 と C が与えられ、これらから C_2 を導出するオペレーションを absorption と呼ぶ。ただし、C_1 は正の節、C_2 は負の節である。このオペレーションは、通常の導出の方向とは逆方向の導出を行なうため、inverse resolution と呼ばれる。absorption の例を以下に示す。

C_1 = member(A, [A|B]).
C = member(A, [B, A|C]).
⇓
C_2 = member(A, [B|C]) :- member(A,C).

3.1.3 intra-construction

intra-construction は、複数個の節 $B_1, B_2, ..., B_n$ が与えられたとき、新しい述語を導入してそれらを一般化するオペレーションである。なお、このオペレーションも absorption と同様に通常の導出とは逆方向であり、inverse resolution と呼ばれるものである。intra-construction の例を以下に示す。

B_1 = arch(([], beam, [])).
B_2 = arch(([block], beam, [block])).
B_3 = arch(([brick], beam, [brick])).
B_4 = arch(([block, brick], beam, [block, brick])).
⇓
A = arch((X, beam, X)) :- p110(X).
C_1 = p110([]).
C_2 = p110([block]).
C_3 = p110([brick]).

$$C_4 = \mathrm{p110}([\mathrm{block}, \mathrm{brick}]).$$

例において、述語 p110 がシステムが新たに生成した述語である。

3.2 学習アルゴリズム

次に、学習アルゴリズムについて述べる。一般に、帰納的学習問題は非決定的要素を含む探索問題に帰着されるため、本質的に並列性を持つ。なぜなら、帰納的学習は初期世界(例集合)に一般化オペレーションを適用することから出発し、各時点における世界に一般化・特殊化オペレーションを施すことによって新しい世界を生成することによって進められるが、ある世界(CIGOL の場合は Horn 節集合)に対するオペレーションの適用は非決定的であり、適用によって新しく生成される世界は1つとは限らない。そして、生成された新しい世界については、これ以降の処理は他の世界とは無関係に個々の世界独自に、つまり並列に実行することができるという点で並列性を持つわけである。しかし、現在の学習システムは逐次処理系上で構築されているため、この並列性を学習アルゴリズムに直接反映することができない。CIGOL の場合には学習アルゴリズムは縦型探索アルゴリズムであり、評価値及び教師との対話をもとに探索制御を行なっている。そこで、本システムでは、KL1 を用いて、問題に内在する並列性を直接反映させた学習アルゴリズムを記述した。これは、先に述べた並列に実行可能な世界の処理を、別々のプロセスに並列に実行させることによって実現している。

4 評価・検討

ここでは、現時点における学習システムに対する評価・検討を行なう。現在、本システムは Multi-PSI 上で稼働しており、文献 [Muggleton 88] に示されている CIGOL の例題 member, arch に対してはそのアルゴリズムが正常に動作することを確認した。しかし、以下に示す幾つかの問題点も残されている。

- 例題が複雑になると探索空間が急激に広くなる。
  これは、現在の学習アルゴリズムが汎用的であり、探索制御をほとんど行なっていないことに起因する。この点に関しては、学習対象に依存する制約や評価値等の導入を行なう必要がある。

- 負荷分散の効果が十分に現れない。
  現在は、各世界に対する処理を各プロセッサにサイクリックに割り付けている。しかし、データ通信量や処理の大きさ等の関係からか、処理速度が上がらないのが現状である。この点については、再度プログラムの見直しを行ない、効率的な割り付けを検討する必要があると思われる。

5 おわりに

本稿では、KL1 による帰納的学習システムの構築について述べた。本システムは、帰納的学習システム CIGOL の持つ学習方法をもとに設計されているが、CIGOL と異なり、学習問題に含まれる並列性を直接反映した学習アルゴリズムを持つ。このアルゴリズムの

実現は、KL1 の持つ記述能力の高さによるところが大きい。なお、簡単な例題について、本システムが正常に動作することを確認したが、問題点として、例題が複雑になると探索空間が急激に広くなることや現在の負荷分散制御方法では処理速度の向上が見られないこと等が残されている。今後の課題としては、これらの問題点についての検討に加えて、並列実行部分を or 並列で実行できる ANDOR-II 上での実現方法の検討を行なっていく予定である。

参考文献

[Muggleton 88] S. Muggleton and W. Buntine, *Machine Invention of First-order Predicates by Inverting Resolution*, Proc. of Machine Learning 88, 1-14(1988).

[Takeuchi 88] A.Takeuchi, K.Takahashi and H.Shimizu, *A Parallel Problem Solving Language for Concurrent Systems*, ICOT TR-418(1988).

[Takeuchi 89] A.Takeuchi and K.Takahashi, *An Operational Semantics of And- Or-Parallel Logic Programming Language, ANDOR-II*, ICOT TR (to appear).

並列協調設計システム評価用プログラム

神矢浩治** 丸山文宏* 吉田健一** 大越隆行** 須田弘美**
箕田依子* 澤田秀穂* 滝沢ユカ*

*富士通研究所 **富士通ＳＳＬ

1. 概要

ＩＣＯＴの委託による中期の研究において，論理設計の支援，自動化を対象として，必要とされる基本的枠組みを追求し，仮説推論や制約処理を導入した論理設計エキスパートシステムの研究を行った．後期ではこの成果を受けて，知識処理による高機能化，及び，大規模な組み合わせ問題に対する並列かつ協調方式による解決方法の研究を目的として，ＫＬ１を用いて並列協調設計システムの研究開発を行っている．

並列協調設計システムでは設計すべき回路を分割した部分回路を各々設計エージェントに割り付け，それらの設計エージェントが互いに協調しながら並列動作する事により，マルチプロセサのパワーを活かしつつより少ない試行回数で制約を満足する回路の設計結果（ＣＭＯＳスタンダードセルのネットリスト）を得ることを目標としている．

今回は，並列協調設計システムの動作アルゴリズムの確認のためのシミュレータとして，並列協調設計システム評価用プログラムを作成したので，シミュレータの協調動作アルゴリズム，及び，ＫＬ１による開発方法等について述べる．

本プログラムは，レジスタ・トランスファ・レベルの動作仕様，データパスを規定するブロック図，及び，面積制約（ゲート数の上限），遅延制約（クロックサイクル）が全て与えられ，動作仕様とブロック図間の整合性チェック，及び，デフォルトＮＪ（ＮＪ：面積／遅延に関する制約と等価な不等式）の設定が完了したところからのシミュレーションを行う．

並列協調設計システムでは各設計エージェントが相互に通信し合う事で同期をとるが，本シミュレータでは各設計エージェントが１ターン（設計，再設計等）実行する毎に同期制御を行う管理部分を設けている．本シミュレータはテーブルアクセス中心のプログラムであるため，テーブルアクセス処理の最適化がポイントになる．

2. 協調動作アルゴリズム

並列協調設計システムでは，設計エージェントが設計を担当する部分回路の設計において，設計，再設計，及び一時的制約を設定した再設計を繰り返すことにより，ＮＪ（制約違反の条件を表現する，不等式あるいは不等式の論理積）を成立させない設計方法を見つけ出すように動く．一時的制約を設定した再設計とは，再設計を行った結果，設計方法を変更することができないループ状態に陥った場合に，制約値を一時的に厳しくして，強制的に設計方法の変更を行う再設計処理である．

図１に，協調動作アルゴリズムの概略を示す（ＮＪに関する詳細＜合成処理など＞は紙数の関係で省略する）．各部分回路の設計とは，どのＮＪも成立させないよう部分回路を設計することである．

部分回路の属性値の交換とは，（再）設計時に各設計エージェントが採用した設計方法の属性値を，他の設計エージェントと交換するものである．各設計エージェントがこの通知された属性値を基にＮＪ

の再評価を行い他の設計エージェントとの協調を行っている．

次に，ＮＪの通知であるが，これは再設計に失敗した設計エージェントが，成立させたＮＪを他の設計エージェントに通知するものである．各設計エージェントが，この通知されたＮＪを基にＮＪの合成を行い，設計エージェントとの協調を強めている．すなわちＮＪの合成により，遅延に関するＮＪと面積に関するＮＪの組み合わせ（論理積）のＮＪや，複数の遅延に関するＮＪの組み合わせ（論理積）のＮＪ等が生成され，設計の開始時（各パスごとの遅延，及び回路全体の面積に関するデフォルトＮＪしか存在しない）よりも，各エージェント間の関連が強まる．

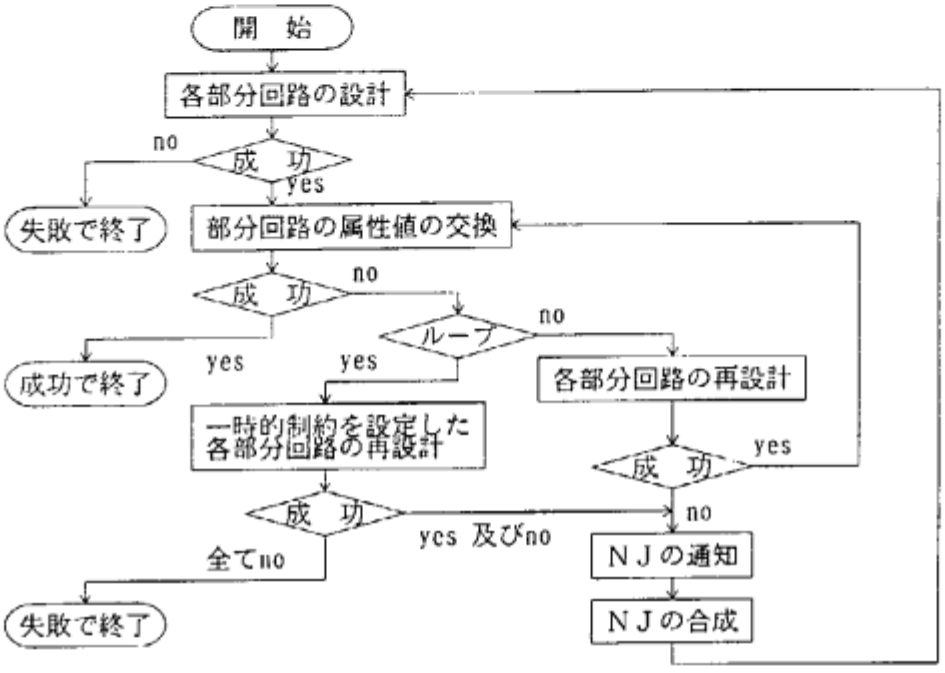


図１　協調動作アルゴリズム

3. シミュレータのシステム構成

本シミュレータは，図２に示すように実行制御部（　　）とテーブル管理部（　　）に分かれる．これらは全てストリームを用いて通信を行っている．

3.1　実行制御部

実行制御部はエージェント管理プロセス，設計エージェントプロセス，及び設計方法プロセスから構成される．

(a)　エージェント管理プロセス

設計エージェントプロセスの同期制御を行うプロセスである．また，各設計エージェントプロセスが成立させたＮＪの他の設計エージェントプロセスへの通知も行っている．並列協調設計システムでは不要となる部分である．

(b)　設計エージェントプロセス

設計方法の選択，及び，評価を行うプロセスである．また，他の設計エージェントプロセスからＮＪの通知を受けた場合，ＮＪの合成を行う．各々個別のプロセッサに割り付けられる部分である．

(c)　設計方法プロセス

各設計方法のデータ（属性値）を保持するプロセスである．各部分回路の設計方法の数だけ存在す

る．

3.2　テーブル管理部

テーブルとして，設計エージェント管理テーブル，ＮＪ管理テーブル，属性値管理テーブル，失敗理由管理テーブル，設計方法管理テーブルがあり，各々を管理するプロセスが存在する．

(a)　設計エージェント管理テーブル

設計エージェントＩＤをキーとして，各設計エージェントプロセスへの要求ストリームを管理するテーブルである．

(b)　ＮＪ管理テーブル

各設計エージェントに設定されたデフォルトＮＪ，及び，ＮＪの合成により生成されたＮＪを管理するスタックである．

(c)　属性値管理テーブル

設計エージェントＩＤ，及び，各部分回路のゲート数に与えられた属性値ＩＤ，または，パスの遅延時間に与えられた属性値ＩＤをキーとして，各部分回路のゲート数，及び，部分回路中の各パスの遅延時間を管理するテーブルである．本テーブルには現在選択中の設計方法の属性値が設定されており，設計エージェントプロセスがＮＪの評価を行う際に参照される．

(d)　失敗理由管理テーブル

各設計方法の評価を行った際に成立したＮＪを格納するテーブルである．全ての設計方法で評価に失敗した場合，このテーブルに格納されている全ＮＪを取り出し，それらの論理積を取って他の設計エージェントプロセスに通知する．

(e)　設計方法管理テーブル

設計方法ＩＤをキーとして，各設計方法プロセスへの要求ストリームを管理するテーブルである．

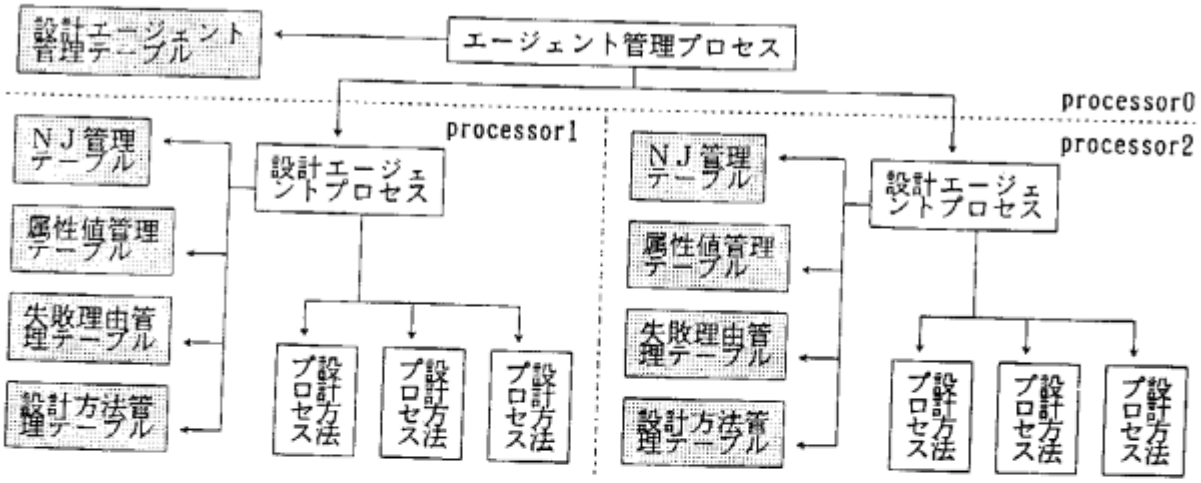
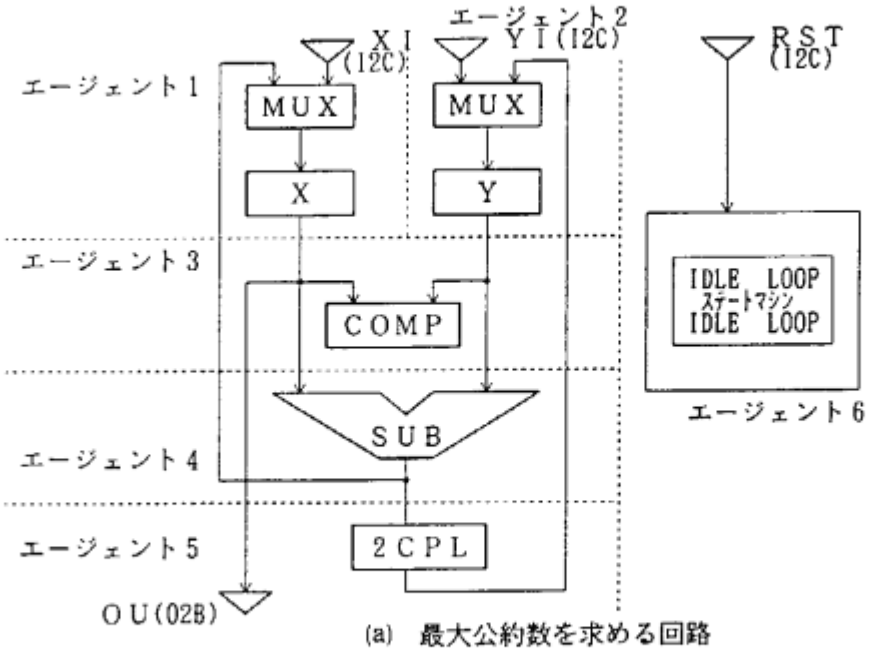


図2　シミュレータのシステム構成
— 設計エージェントプロセスが２つの場合 —

4. シミュレーションの実験結果

今回は，ユークリッドの互除法により最大公約数を求める回路の設計を例に簡単なシミュレーションを行った．図3に示すように，回路全体を６つの部分回路に分割し，面積，遅延合わせて２６種類のデフォルトＮＪを設定した．



(a) 最大公約数を求める回路

S1	P11=XI→MUX(input)→X(input)	P12=MUX(input)→X(input)
P13=X	P14=MUX(select) →X(input)	P15=∨→∧→X(clock)
P16=∧→MUX(select) →X(input)	P17=∧→∨→∧→X(clock)	S2
P21=YI→MUX (input) →Y(input)	P22=MUX(input)→Y(input)	P23=Y
P24=MUX(select) →Y(input)	P25=∨→∧→Y(clock)	P26=∧→MUX(select) →Y(input)
P27=∧→∨→∧→Y(clock)	S3	P31=COMP(>)
P31=COMP(=)	P31=COMP(<)	S4
P41=(X→)SUB	P42=(Y→)SUB	S5
P51=2CPL	P52=OU	S6
P61=idle→∧	P62=loop	P63=RST →¬→∧

(b) 各設計エージェントの属性値（S*はゲート数　P*は遅延時間）

0) S1+S2+S3+S4+S5+S6 > Area	1) P11 > Time	2) P61+P14 > Time
3) P63+P14 > Time	4) P61+P15 > Time	5) P63+P15 > Time
6) P21 > Time	7) P61+P24 > Time	8) P63+P24 > Time
9) P61+P25 > Time	10) P63+P25 > Time	11) P13+P52 > Time
12) P13+P31+P16 > Time	13) P13+P31+P17 > Time	14) P62+P16 > Time
15) P62+P17 > Time	16) P23+P31+P16 > Time	17) P23+P31+P17 > Time
18) P13+P41+P12 > Time	19) P23+P42+P12 > Time	20) P13+P33+P26 > Time
21) P13+P33+P27 > Time	22) P23+P33+P26 > Time	23) P23+P33+P27 > Time
24) P13+P41+P51+P22 > Time	25) P23+P42+P51+P22 > Time	

(c) デフォルトNJ

図3　実験用データ

表１に実験結果を示す．面積制約値(Area)，及び遅延制約値(Time)を与えた際の，（再）設計回数，設計に成功した場合の最大遅延時間，及びゲート数を示す．また，面積制約値１５５０ｃｅｌｌ，遅延制約値１４０ｎｓで設計した場合に成立したＮＪを以下のものである．

１４２.８＞Ｔｉｍｅ∧１５７２＞Ａｒｅａ

表１　実験結果

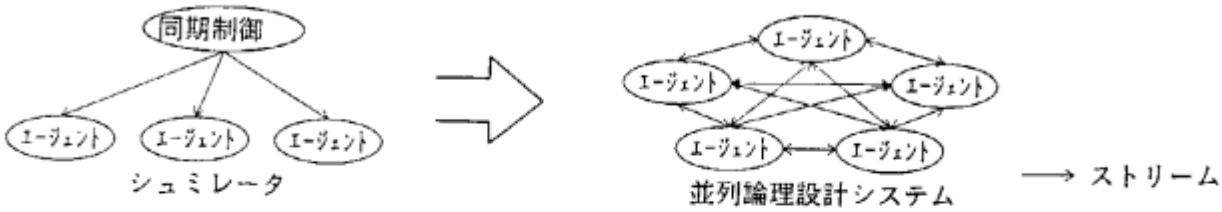
面積制約値 (Area)	遅延制約値 (Time)	設計結果	設計回数	再設計回数	ゲート数	最大遅延時間
1600cell	140ns	成功	1	0	1572cell	132.5ns
1550cell	150ns	成功	1	1	1538cell	143.4ns
1450cell	190ns	成功	1	1	1428cell	180.4ns
1400cell	200ns	成功	2	1	1394cell	191.3ns
1550cell	140ns	失敗	3	2	———	———

5. シミュレータ作成時の問題点と今後の課題

シミュレータ作成時の問題点としては，同期制御に関するものと，テーブルアクセスの最適化の２点があげられる．また，並列論理設計システムの設計に関しては，ＫＬ１とＥＳＰのインタフェースがネックとなると思われる．

5.1　同期制御

はじめに述べたが，本シミュレータでは同期制御を行う管理部分が存在する．並列論理設計システムを設計する際には，直接設計エージェント間にストリームをはり，設計エージェントが相互に同期を取る方式を考案する必要がある．



本シミュレータの設計時に，同期制御を管理する部分を無くす事を検討したが，以下のような問題点があるため同期制御を管理する部分を設けた．

(a)　エージェントの数が固定でないため，エージェント間のストリームを動的にはる必要があり，プログラムが複雑になる．

(b)　プログラムの良い終結方法が無い．各エージェントに優先順位が存在しないため，ショートサーキット法等を用いることができない．

5.2　テーブルアクセスの最適化

本シミュレータは，テーブルアクセスを中心としたプログラムであるのにもかかわらず，テーブルをリスト処理で実現しているため，最適化のための検討が必要である．本シュミレータ作成時に，ＫＬ１で提供されているプールを用いることも検討したが，以下の問題点がありテーブルを自作した．

(a)　複数データの格納（データ参照の後処理で多用）用のメッセージ・プロトコルが用意されていな

い．

(b)　2種類のキーでの検索等の処理が必要である．

(c)　複数プロセスから同一テーブルへ並列アクセスする際の同期が取りづらい．

実際にＫＬ１のプールに，上記の(a)，(b)の処理をくわえたプログラムを作成し，自作のテーブルとの処理速度の比較を行ったが，現在のところ自作のテーブルの方が処理が速い．

5.3　ＫＬ１とＥＳＰのインタフェース

並列論理設計システムでは，ユーザインタフェイスをＥＳＰで作成する．この際ＫＬ１とのインタフェースをストリングｉｏデバイスを用いて行う．現在ＫＬ１では，ストリングと他のデータ形式との変換をサポートする組み込み述語が提供されておらず，設計時にＫＬ１とＥＳＰ間の通信量を減らす等の検討が必要である．ＳＩＭＰＯＳのクラスｓｙｍｂｏｌｉｚｅｒのような機能をＰＩＭＯＳでも用意して頂くことを希望する．

6. まとめ

本シミュレータを用いた簡単な実験の結果から，かなり少ない試行回数で回路の設計結果が得られることが期待される．設計エージェントプロセス間の通信による同期制御の方式を採用すれば，よりマルチプロセサのパワーを活かし高速化できると思われる．

判例を用いた法的推論システム

新田 克己　　星田 昌紀 (ICOT)

1 はじめに

判決の予測，裁判における論理展開，法律にしたがった行動計画など，法律の専門家の活動を支援するシステムを開発するには，法律における推論を解析し，その計算モデルを構築する必要がある。

法律のシステムは，条文を論理式で表現し，法的結論をその論理式で証明する定理証明の問題として形式的に取り扱える場合が多い。しかし，現実の裁判などでは，法律で与えられる知識は具体的な問題を解決するには不完全であり，法の解釈がしばしば問題となる。解釈に依存した知識をどのようにとり扱うかが，法律システムを構築する上での長い間の課題であった。

解釈を行うためには，学説や他の法律や判例などが参照される。ここでは判例の表現法と利用法を考察し，法的推論の並列化の方法を提案する [新田 90]。

2 事例を用いた法的推論モデル

2.1 事例の表現

事例の中で最も重要なものは判決文である。判決文は過去の事件において，観測事実から結論に至るまでの推論過程を表すものである。その主な内容は，どのようなルールがどのような事実のもとで適用されたかという説明である。原告，被告の双方の当事者は，自分に都合の良い事実を強調して，自分に有利な理論を展開し，裁判官はそれぞれの論理を比較し，より適したものを採用する。当事者の論理と裁判官の論理が判決文から抽出できる。

これを図1のような図式で考えることができる [Branting 89]。この図でＲ１，Ｒ２は法律から抽出されたプロダクションルールを表し，ｒ１，ｒ２，ｒ３は判断結果から抽出された事例ルールを表し，ＦＡ，ＦＢ，ＦＣは観測された事実を意味ネットで表現したものを表す。

この例では，ＦＡはＡに，ＦＢはＢに該当したなどとそれぞれ判断されて，最終的にＧｏａｌという結論に至ったことを表す。ここで，ｒ３は状況ＦＢのもとでルール（IF C THEN E）が生成されたことを表す。これは，互いに類似する２つの概念ＣとＥを，ある状況では同一視したいときなどに用いることができる。

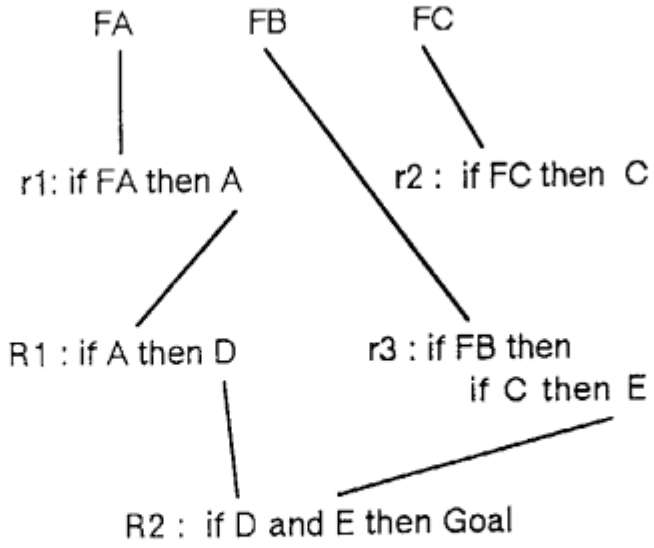


図 1: 判決文の説明

このように，ルールには条文から抽出されるルールの他に，判例として与えられるルール，一般常識を表すルール，新たに裁判官によって創造されたルールなどを含む。一般的なルールと判断のルール（事例ルール）との違いは，前者は条件部がすべて満足されないと起動されないのに対し，後者は条件部がかなり満足されれば起動される可能性があること，前者の条件部は固定されているのに対して，後者の条件部の概念が拡張（一般化）されうることである。また，前者の推論結果は常に正しいが，後者の推論結果は正しいとは限らない [Branting 89]。

ここでは意味ネットを，

{ 対象, 関係, 値 }

の３つ組の集合で表現する。事例ルールの中では

IF { {対象１, 関係１, 値１}, 重要度１} ∧
{ {対象２, 関係２, 値２}, 重要度２} ∧
. .
{ {対象ｎ, 関係ｎ, 値ｎ}, 重要度ｎ}
THEN
述語 / リンク / ノード / ルールの生成

のように，意味ネットをリンクの重要度とともに記述する。

2.2 事例の適合

新しい事件が与えられたときの，過去の事例の利用方法について述べる。新しい事件で成り立つ事実は，意味ネットとその事件を特徴付ける述語の集合で表されているとする。

このような入力データに対し，法律のルール（そのほとんどがプロダクションルールとして記述されている）を直接適用しようとしても，通常はルールの条件部が満足されない。観測される事実と法的概念の抽象度が違うからである。そこでまず事例ルールを起動すると，観測事実の中からいくつかが着目されて抽象的な概念が生成され，その結果，法律のルールが起動されることになる。

事例ルールの場合，条件がすべて満足されなくともある程度の類似性があれば起動される [Branting 89]。例えば，

IF ｛｛船♯1，船長，山田｝，20｝
THEN ～

というルールがあったとき，

｛飛行機♯1，機長，田中｝

というデータがあったとする。2つの3つ組の対象，関係，値のそれぞれが，与えられた階層構造のもとで同一の上位概念を持つ場合に，その3つ組は対応がとれるものとする。この例では，［船♯1］と［飛行機♯1］はともに［乗物］という概念の下位概念（またはインスタンス）であるという知識と［船長］と［機長］はともに［責任者］という概念の下位概念であるという知識と［山田］と［田中］はともに［自然人］の下位概念（またはインスタンス）であるという知識があれば，｛船♯1，船長，山田｝は｛飛行機♯1，機長，田中｝と対応がとれる。対応をとるにあたって，上位概念をたどる段数が少ないほど，両者の類似度は近いと判断される。

判決予測を行うときには，入力された事件の意味ネットが，事例ルールの適用によって次第に詳細化と抽象化が行われ，あらかじめ用意された特定の事件のパターンに合致するとき最終結論に至る。

2.3 判決の例

判決の例として，労働基準法79条

[労働者が業務上死亡した場合においては，使用者は，遺族に対して，平均賃金の1000日分の遺族補償を行わなければならない]

に関する判決を考えてみる。［業務上の死亡か否か］の判断基準は，業務遂行中の死亡であること，業務と死亡との間に因果関係があること，の2つであることが判例で確立している。

ここで因果関係の判定は容易ではない。他の原因が重なっていることもあるし，業務が間接的な原因になっているかもしれないからである。従って，多くの事例が類型化され，それぞれに判断基準がルールとして決められている。

以下の事例は，農協職員山田氏が勤務時間後に，料理屋での会議に業務命令で参加し，その後，帰宅の途中で組合員鈴木氏の家に保険加入の勧誘の目的で行こうとして事故にあった事件である［長野地判 39.10.6 の改変］。

裁判では，業務上の死亡かどうかが争われた。その論点の1つとして，被告（労働基準監督署）は，(1) 山田氏は庶務係なので 保険勧誘は本来の業務（職務）ではない，(2) 鈴木氏の家へ勧誘に行けとの具体的命令は出していない，の理由で業務でないとしたのに対し，原告（山田氏の遺族）は，(1) 保険勧誘の成績は評価の対象となっていたので実質的業務である，(2) 保険勧誘を奨励する局長の訓示やキャンペーンがあったので包括的業務命令があったとみなせる，と主張した。

まず，この事件の観測された事実の意味ネット表現の一部を図 2に示す。

被告が，保険勧誘は山田氏の直接の業務（職務）でないことを理由に，業務でないとしたことは，以下の事例ルールで表現できる（ルールの要部のみ示す）。

```
IF  { {雇用関係 #1, 従業者, 山田氏} , 20} ∧
    { {雇用関係 #1, 雇用者, 郵便局} , 20} ∧
    {not {雇用関係 #1, 職務, 保険勧誘} , 100}
THEN [make(職務外)]
```

それに対し，原告が，保険の勧誘が勤務評価の対象であったので，実質的に業務であったとしたことは以下のように表すことができる。

```
IF  { {雇用関係 #1, 従業者, 山田氏} , 100} ∧
    { {雇用関係 #1, 雇用主, 郵便局 #1} , 100} ∧
    { {郵便局 #1, 業務, 保険勧誘} , 100} ∧
    { {郵便局 #1, 責任者, 局長 #1} , 100} ∧
    { {局長 #1, 職務, 勤務評価} , 100} ∧
    { {勤務評価 #1, 対象, 山田氏} , 100} ∧
    { {勤務評価 #1, 内容, 保険勧誘} , 100}
THEN [make (職務範囲)]
```

ここで新しい事件について考える。これは，建築会社の材木運搬トラックの運転手である田中氏が材木の積み降ろし中に事故にあったものである。田中氏の材木の積み降ろしは雇用契約に含まれていないとすると，労災認定においてどのような判断がなされるかを予測する。

この事件を意味ネットで記述したものの一部を図 3に示す。

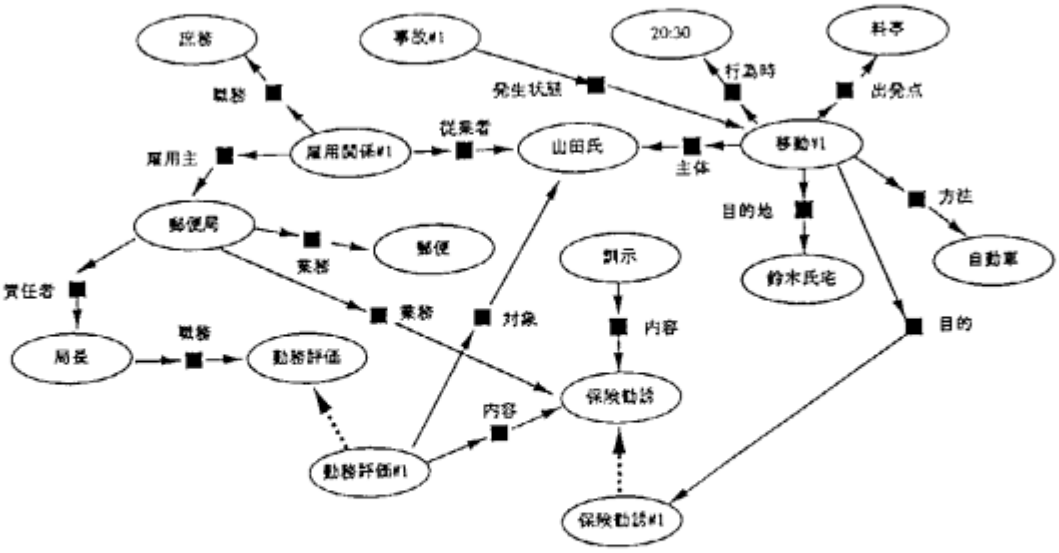


図 2: 山田事件の意味ネット表現

このネットを 図2と比較し，［雇用関係 #1］と［雇用関係 #2］，［山田氏］と［田中氏］，［郵便局］と［建築会社］，．．のようにマッピングすれば対応がとれる。すなわち，山田事件によってなされた双方の主張（職務でない vs 成績評価の対象）の事例ルールを田中事件に適用すれば，田中事件でも同じ議論が成立することが推定される。

2.4 推論結果

同じ観測事実を与えても，どの事例ルールを適用するかによって異なる結果が得られる。事例ルールの適用による新たなデータの生成は，１つの仮説の生成とも考えられる。異なる事例ルールの適用によって異なる仮説が生成されるが，作業記憶の中でそれらは同一に取り扱われるのではなく，異なった世界で成り立つ事実として管理されなくてはならない。

3 法的推論システム

3.1 データフローグラフ

一般に事例ベース推論は，膨大な事例を検索し，照合しなくてはならないから，実行時間が非常にかかることが予想される。そこで，ここでは並列処理による推論時間の短縮

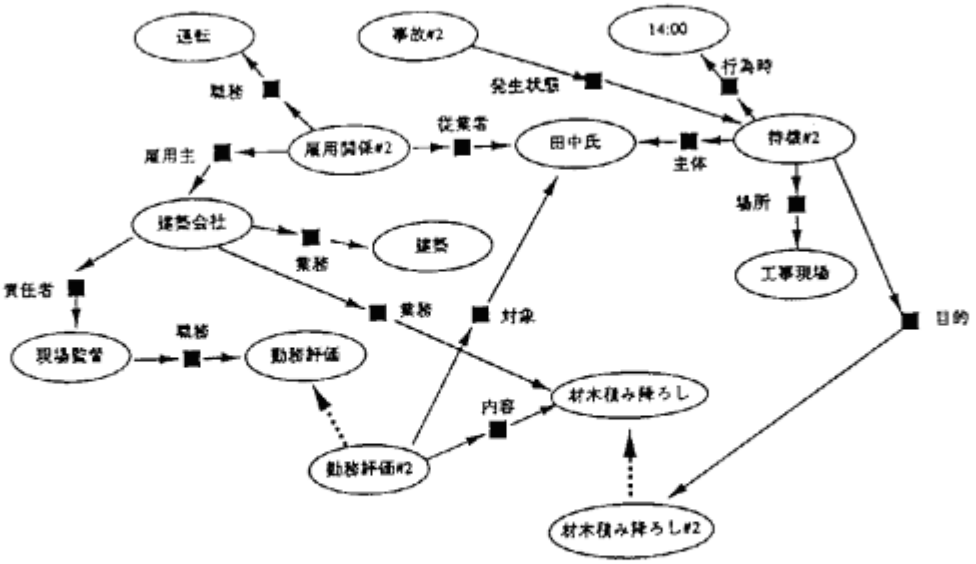


図 3: 田中事件の意味ネット表現

の可能性について検討する。

前に述べたように，法律における説明は一般のプロダクションルールと事例ルールで表される。プロダクションルールの条件部はＲＥＴＥネットなどのデータフローグラフに展開することで高速化が図れることが知られている。事例ルールの場合もＲＥＴＥと類似のデータフローグラフに展開することができる（図 4）。このデータフローグラフが通常のＲＥＴＥネットと異なるのは以下の点である。

- ３つ組の各引き数が同じ上位概念ノードを持つかどうかのチェックが必要である。
- ＲＥＴＥネットの場合には全ての条件が満足されたトークンの組合せが出力されるのに対し，このグラフでは一部の条件が満たされないトークンの組み合わせでも，ある基準以上の類似度があれば出力される。

このデータフローグラフを並列論理型言語ＫＬ１で実現する。これは，事例ルールの条件部をコンパイルして，ＫＬ１プロセスをノードに対応させ，ストリームをリンクに対応させたデータフローグラフを作成することによる。ルートノードのプロセスから，新しい事件の事実データをトークンとして流すと，条件照合の並列処理が実現できる。

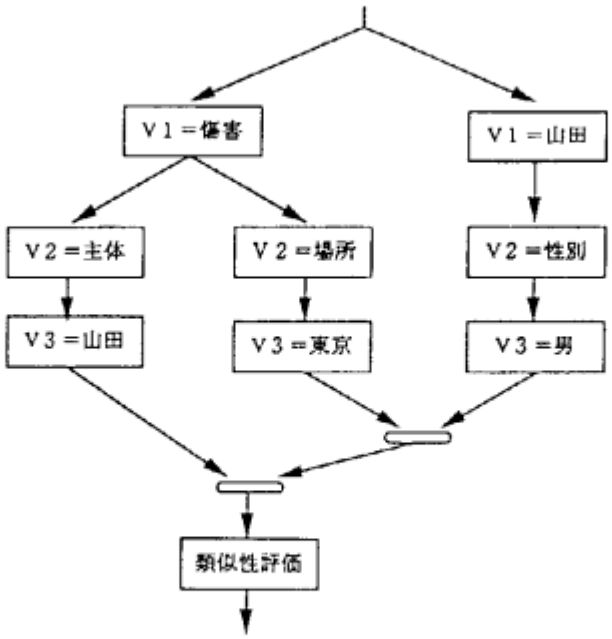


図 4: データフローグラフ

3.2 同期

ルールの実行サイクルに入っている間は，図5に示すようにストリームのループができていて，トークンが常に流れているように見えるが，現在のインプリメントでは条件照合のフェーズと競合解消のフェーズが区別されている。

条件照合のフェーズでは，図の入力ストリームからネットワークの中へトークンが流れている。トークンが全部流れ終わると，その結果は出力ストリームを通って PE0 に競合集合として集められ，競合解消のフェーズとなる。競合解消のフェーズでは，競合集合からある基準で1つのルールが選ばれ，そのルールの実行部の記述にしたがって，新たなトークンを入力ストリームに流して再び，条件照合のフェーズとなる。

このように同期をとるため，ストリームの中にデータの終了を知らせる end 記号を含ませている。入力ストリームの中で end を発見すると各プロセスは1つのサイクルの終了を知り，自分も end を出力ストリームに出力する。競合解消を行うプロセスは end がストリーム中に現れるまで idle 状態にあり，最後の end を発見すると，条件照合のフェーズが終了したことを知って，active な状態になる。

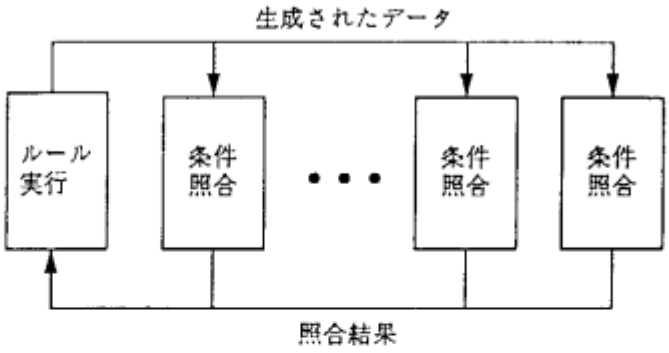


図 5: ＰＥへの割り付け

4　おわりに

法的推論における事例検索の並列化の可能性について述べた。一般にプロダクションシステムは競合解消を行うため，ＫＬ１でインプリメントすると 情報が１つのＰＥに集中する。従って，並列性がどこまで引き出せるかの限界を見極める事が必要である。 ここでは，事例ルールは条件部が複雑になるので，条件照合に時間がかかること，データに環境情報を持たせることにより競合解消が必要ではないことなどから，情報が集中することによるネックが比較的小さいと予想している。

参考文献

[新田 90] 新田, 星田 : 事例を用いた法的推論とその並列化, 情報処理学会研究会資料 (1990).

[Ashley 88] K.D.Ashley, et.al. : "Compare and Contrast, A Test of Expertise", Workshop on Case-Based Reasoning (1988)

[Branting 89] L.K.Branting : "Representing and Reusing Explanations of Legal Precedents", International Conference on Artificial Intelligence and Law (1989).

[Rissland 89] E.L.Rissland, et.al : "Interpreting Statutory Predicates", International Conference on Artificial Intelligence and Law (1989).