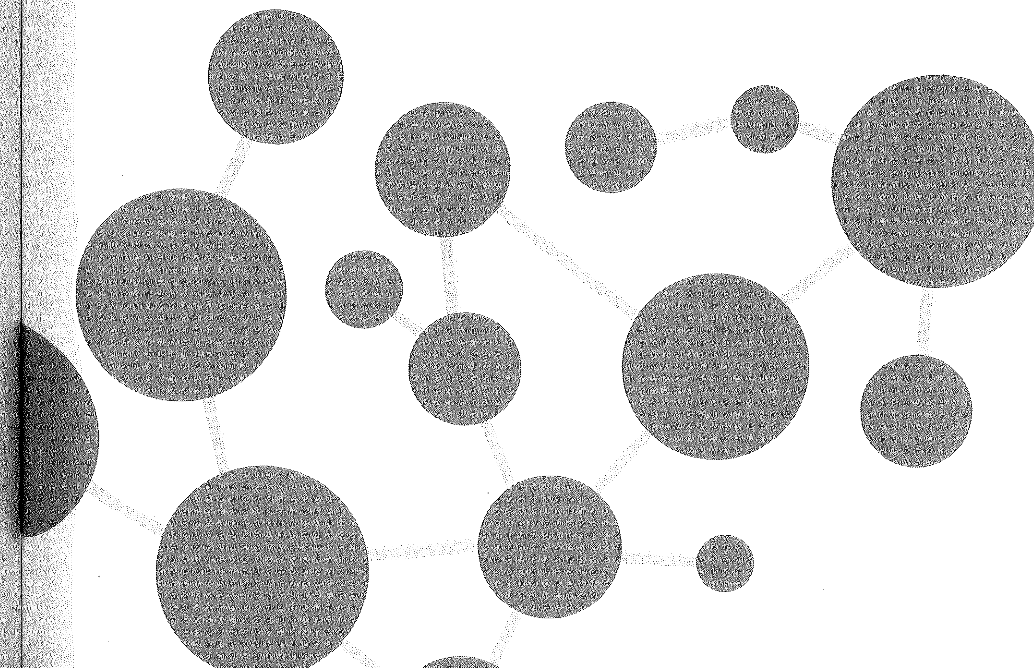
第 III 編

プログラミング応用編

——強力な並行・並列プログラミング言語KL1を得て——



第 III 編　目次および執筆者

章・節	ページ	執筆者
第 1 章　はじめに	153	瀧 和男
第 2 章　並列プログラムを設計する	155	瀧 和男
2.1　並列プログラムを設計するとは	155	
2.2　KL1 プログラミングはプロセス指向プログラミング	158	
2.3　ペントミノと動的負荷分散	166	
2.4　最短経路問題と高並列オブジェクト指向プログラミング	174	
第 3 章　応用プログラム	184	
3.1　並列応用プログラムについて	184	瀧 和男
3.2　LSI 配線	188	伊達 博
3.3　論理シミュレーション	196	松本 幸則
3.4　遺伝子情報処理	210	星田 昌紀 戸谷 智之
3.5　法的推論	219	新田 克己
3.6　定理証明系 MGTP	228	長谷川 隆三 越村 三幸
3.7　自然言語解析	241	山崎 重一郎
3.8　データベース	252	横田 一正 河村 元夫
第 4 章　むすび	262	瀧 和男

第 1 章

はじめに

第 III 編は，本書の最も大きな主題を扱うところである.「プログラミング/応用編 —強力な並行・並列プログラミング言語 KL1 を得て—」という編のタイトルは，この主題をよく表現している. KL1 という強力な並列言語を得て，これまでになかった並列プログラミングが可能となり，これまで成功しなかった新しい並列応用が試されるようになったという意味あいである. 本書のタイトルにある「汎用並列処理への道」とは，このような新しい技術潮流によって，従来はほとんど数値計算分野でしか実用化されていなかった並列処理が，より汎用的に数々の応用分野で利用できるようになるとの予想と期待を込めたものである.

これから第 III 編でお伝えすることは，第一に並列プログラミングのおもしろさと新鮮さ，第二に KL1 言語のすばらしさと実際のプログラミング，第三に並列応用の数々の実例と評価結果，そして大規模並列処理への期待感である. これは誇張でも何でもなく，筆者の素直な実感である.

並行・並列プログラミングのなかの「並行」の部分は，計算機の長い歴史でみると，OS プログラムの設計やバグ取りの難しさとして忌み嫌われていた種類の技術と，本質的に変わらないものである. 一体それがなぜおもしろいのかと聞かれるならば，KL1 言語というすばらしい道具が出現したことによって，並行・並列プログラミングの苦労のほうが激減して，その分おもしろさが目立つようになったといえるのではないだろうか.

もちろん手放しでおもしろいと喜ぶべきものではなく，第五世代コンピュータの並列プログラミングと並列実行は，計算機の世界ではほとんどニューカルチャーと呼べるくらいに，従来の逐次計算機上でやってきたこととは異質である. その分，これから解明しなければならないことが山積しているし，楽しみも大きいということができる.

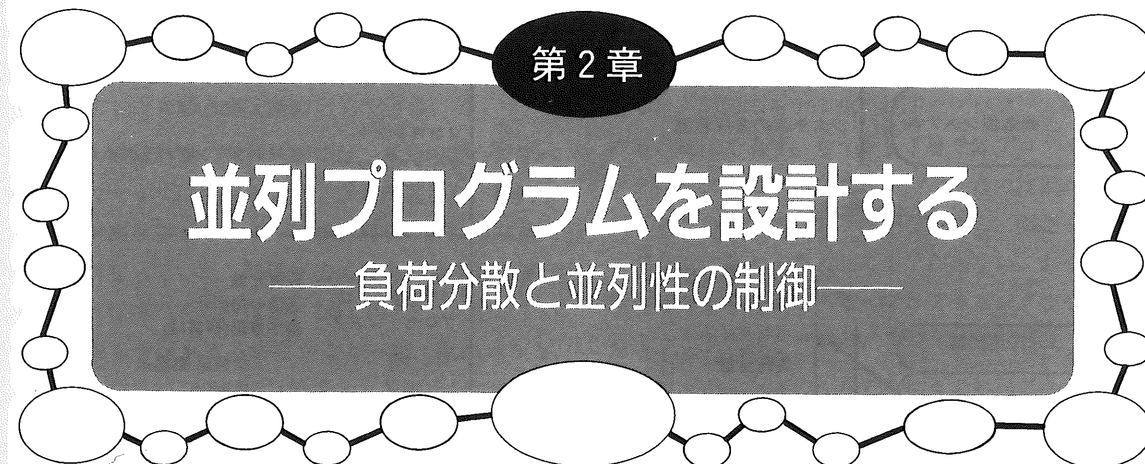
KL1 プログラミングの特徴は，一つにはプロセス指向プログラミングである. 本文中の言葉を借りれば，まるで「配管設計」のようなプログラミングのおもしろさであり，またオブジェクト数万個以上の高並列オブジェクト指向プログラミングの新鮮さである. また一つには，プラグマを使って負荷分散とスケジューリングを思いのままに操れることである. これも本文中の言葉を借りれば，使用するプロセッサ数を増やしていって，負荷分散がうまくいったときの「加速感」はたまらないのである. このような喜びを与えてくれる並列言語と計算機システムは，これまでにはなかったものであり，その上では間違いなく，新世代の計算機文化が醸成されているという実感がある.

第 III 編は二つの主要な章からなる. 2 章では，第五世代コンピュータの並列プログラミングに関する基本的な考え方を示すとともに，プログラムの設計方法，負荷分散手法，評価方法などついて解説する. 説明にあたっては，第五世代コンピュータ国際会議 FGCS'88 のデモに使われたパフォーマンスメータ，ペントミノ，最短経路問題などの具体的なプログラムを取り上げ，そこで工夫された各種の技法を題材としてイメージの掴みやすさに重点をおいた解説を試みる. ここでは，並列プログラミングの新しさ，異質さに対する新鮮な

驚きに接していただきたい．

3章では，FGCS'92 に出展された主要な並列応用プログラム7種を紹介する．LSI の自動設計，遺伝子情報処理，法的推論システム，定理の自動証明，自然言語処理，データベースなど多方面にわたっている．その大部分は256プロセッサの並列推論マシン PIM で実行され，きわめて高い性能を示したものである．それらの実現技術や研究開発方法，評価結果などについて解説する．ここでは，2章で紹介した基礎技術が本格的な応用プログラムに適用され，良好な結果を示した具体例について楽しんでいただくことにする．また，評価結果などをご覧になって，紹介してきた並列処理技術の将来性を感じとっていただくことができればありがたい．

それではこれから，みなさんをニューカルチャーと遭遇する旅にご案内しよう．



2.1 並列プログラムを設計するとは

2.1.1 はじめに

並列プログラムを書くという作業は，従来の逐次計算機上のプログラムに慣れ親しんできた多くの人々にとっては，ずいぶん異質なものと映るかもしれない．その理由は，効率のよい並列処理を実現するためには，これまで考えることもなかった広範囲な事項に注意を払うことが必要なためである．

けれどもそれは，決して難しくてつまらないことを意味するわけではない．特に KL1 という，これまで世界中の誰もがもち得なかった強力な並列プログラム言語を手に入れることができた結果，筆者らの並列処理は，世の中にこんなにおもしろいものはないという領域に突入することとなった．

本節では，並列プログラミングに関する諸々のお話の導入部として，そもそもなぜ並列処理をしたかったのか，ユーザが期待していたことと現実の並列プログラミングとはどのくらい離れているのか，そのギャップを埋めるための技術要素，あるいは技術課題とはどんなものなのか，並列プログラムの研究と開発を進めるには一体どんな配慮が必要なのか，といった基本的な事項について順を追って解説してゆきたい．

またこれまでに並列計算の経験をおもちの読者は，次の点に注意を寄せられたい．ここで扱う並列プログラミングの話は第五世代コンピュータの並列プログラミングであり，従来の数値計算の並列処理に比べると，性質が大きく異なるとともに，概して一段と高度な並列処理技術を取り扱おうとしているものである．したがって注目している技術要素が，従来とは異なるかもしれないことを心にとめながら，以下を読み進まれたい．

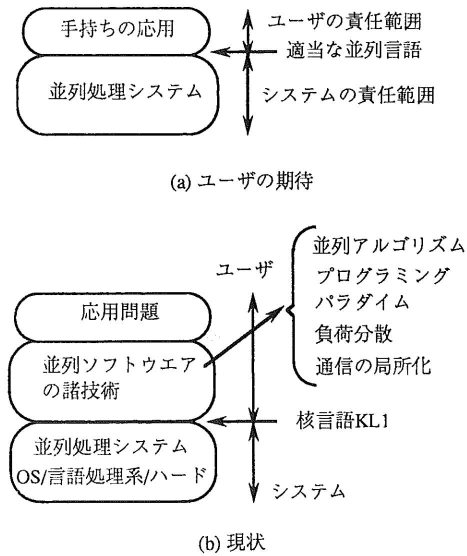
2.1.2 なぜ並列なのか

並列プログラムとは，計算機に並列処理をさせるためのプログラムであることはいうまでもないが，それではなぜ並列処理したいのだろうか．普通は，残念ながらおもしろいから並列にしたいわけではなく，次のいずれかの目的をはたそうとするためである．

1) 計算時間を短縮するため，または計算時間を増加させずにより高度な処理を行なうため
2) 1台の計算機では扱えない大量のデータを扱うため
3) 並列言語を用いてより自然な問題記述を可能とするため

ほかに信頼性を高めることを目的とする場合もあるが，技術的にはやや性質が異なるため本誌では扱わない．

これらの項目の一つ，または複数個を目的として並列処理するわけだが，項目によって重要な技術課題は異なってくる．それらのなかで，第五世代コンピュータプロジェクトの並列処理では，項目1) を主な目的として研究を進めた．あとから出てくる具体例のほとんどはそうである．2) または3) については，副次的な



(a) ユーザの期待

(b) 現状

図 2.1.1 並列処理システムに対する期待と現状

目的となる場合もあったが，それらを中心に据えた研究はまだこれからであり，今後の興味深い課題である．

一方，対象とする計算の性質によっても技術要素は異なる．第 II 編でも述べたように，第五世代コンピュータでは動的に変化する計算 (計算負荷を事前に予測しにくい計算) や均質さの低い大量データを扱う計算 (データに応じて処理内容も大きく変化する計算) を対象としている．筆者らのプロジェクトでは，並列処理技術のなかでも，知識情報処理に必要なこの種の計算を効率よくこなすための技術に，特に重点を置きながら研究を進めてきた．

2.1.3　期待と現状のギャップ

まず意識のずれの問題について説明する．ハードウェアから OS まで含めた層を並列処理のシステムの層と考えよう．ユーザの立場からすると，図 2.1.1(a) に示すような意識となる．並列言語をインタフェースとして，手持ちの応用をその言語で書き下すと，システムが気持ちよくそれを並列実行してくれる．もちろん性能も上がる．そのような期待が，ICOT 設立当初には システム開発者にもあったし，応用プログラム開発者のなかにはいまでも存在している．この場合，手持ちの応用を書き下すところまではユーザの責任範囲で，その言語で書かれたプログラムを効率よく動かすのはシステム側の仕事といった役割分担を想定して

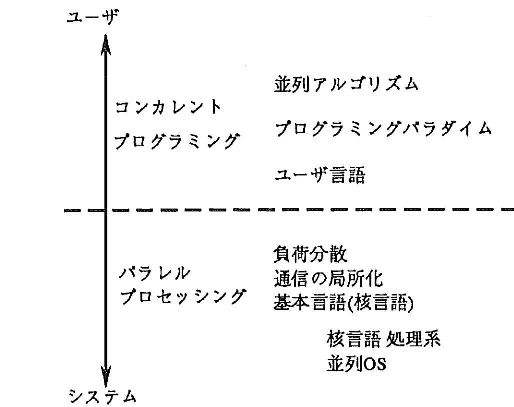


図 2.1.2 並列ソフトウェアの研究課題

いた．ところが，やってみるとどうも違う．そんな単純な話ではなかったのである．並列処理の研究を続けてきて，図 2.1.1(b) に示すように応用問題と並列処理システムの間には，並列ソフトウェアの諸技術とでも呼ぶべき厚い中間層があって，これがほとんど未解決になっていることがわかった．プロジェクト中期の終わり頃から，そのことに対処するための研究活動を本格化した．

2.1.4　並列プログラミングの技術要素

ここで並列ソフトウェアの諸技術と書いたものは，そのまま並列プログラミングの技術要素であり，並列ソフトウェアの研究課題ともいえる (図 2.1.2)．

A.　論理的な並行性と実行時の並列性

研究課題には大きく分けて，ユーザ寄りの話とシステム寄りの話がある．図中，コンカレントプログラミングとは，主に問題解法のアルゴリズムと論理的な並行性を設計・記述することである．あくまで論理的な並行性であり，実行時の並列性は問わない．この部分はユーザ寄りの話である．一方パラレルプロセッシングとは，記述された論理的な並行性を実際に複数のプロセッサに割りつけ，並列実行させることである．実行時の並列性の制御と実行効率のよさを問題にする．この部分は，どちらかというとシステムが面倒を見るべき話である．

このように論理的な並行性の設計・記述と，実行時の並列性の設計・制御に関する事項をはっきりと分離して扱おうとする動きは，第五世代コンピュータプロジェクト以前の並列処理には見られなかったものである．その理由は，知識情報処理の計算の性質がそれを強く要求したためと考えることもできる．

従来の数値計算の並列処理では，多くの場合にすべてのプロセッサには同じプログラムが置かれ，静的なデータ分割とプロセッサへの静的割りつけにより並列処理を行なっていた．問題解法に関する論理的な並行性と，実行時の並列性はおおむね 1 対 1 に対応していて，それらを区別する必要がなかった．逆にいうと，そのような単純な扱いでは効率よく並列実行できないような問題は，プログラム化の難しさのために並列処理の対象にされなかった．一方，第五世代コンピュータが対象とする知識情報処理では，動的に変化する計算 (計算負荷を事前に予測しにくい計算) や均質さの低い大量データを扱う計算 (データに応じて処理内容も大幅に変化する計算) を多く発生する．本質的に取扱いの難しい計算である．このような場合には，問題解法のアルゴリズムとプロセッサへの仕事割りつけの記述が強い依存関係をもっていると，とても手に負えなくなることがわかってきたのである．そこで必要となった技術課題を整理したのが図 2.1.2である．

B.　研究課題

図 2.1.2の一番上は，並列アルゴリズムである．逐次マシンで使われているよいアルゴリズムは，逐次処理であることを利用して高い効率を実現しているものが多い．アルゴリズムが強い逐次性をもつから，これを並列言語で書き下して無理矢理並列マシンにマッピングしても，たくさんあるプロセッサが順に動いてしまうだけである．並列マシンを効率よく動かすためには，並列処理向きのアルゴリズムあるいは逐次性の低いアルゴリズムに作り直さなければならないと考えるべきである．手持ちの応用を，言語だけを並列言語にかえて書き直せば済むという話ではない．ここで注意が必要なのは，並列性を多く含んだアルゴリズムに書き換えるとき，計算量のオーダが増えてしまうようなミスを犯しやすいことである．問題サイズ Pに対し，$P\log P$で計算できていたものが，Pの 2 乗かかるアルゴリズムに化けたりする．これではいくら並列マシンをもってきても，それ以上に計算時間が増えることになりよくない．コンスタント程度は目をつぶるとして，計算量のオーダを増加させないことが重要である．

次はプログラミングパラダイムである．プログラミングの型，手本とでもいおうか．プログラミングの上層では，問題のモデル化の技法やプログラム構造の決め方の技法であり，もう少し下層ではプログラミングテクニックなどが含まれるが，これらの技術が蓄積されていない．つまりお手本がないから，問題を並列プログラムに落とそうとしても手が出ない．お手本の蓄積がぜひとも必要である．プログラミングの型を与えるような並列ユーザ言語の開発も重要である．

プログラミングのなかでこれまで述べてきた辺りの仕事は，最低限ユーザが行なわなければならない仕事である．これをコンカレントプログラミングの研究課題と呼んでいる．ここに含まれる並列性は論理的並列性で，それが実行時にマシンにどうマッピングされるかはこのレベルでは一切扱われていない．アルゴリズムが 10 万の並列性をもっていても，それを 1,000 プロセッサのマシンにマッピングしてもよいし，100 プロセッサのマシンにマッピングしてもよいのである．

マッピングの話はパラレルプロセッシングの問題である．これは本来，システムが丸抱えで面倒をみてくれればありがたいが，まだその段階にははるかに至らない．パラレルプロセッシングにかかわる技術課題は次のとおりである．問題を多くの部分問題に分けプロセッサの負荷が均等になるように，また暇なプロセッサが出ないように割りつけを行なうのが負荷分散である．これは仕事をばらまく話である．一方で，部分問題の間で多くの通信を行なうもの同士は，ネットワークの負荷を軽減するためになるべく近く配置すべきである．これを通信の局所化という．これは仕事をばらまかずに固める話であり，さきに述べた負荷分散とは反対の要求である．これを同時に満たそうとするところに難しさがある．さらにそれらを記述するためにどんな言語を用意するか，OS でどこまでサポートするか等々，確立されてないものばかりである．

C.　目指す並列処理はニューカルチャー

したがって，ユーザがやらなければならないコンカレントプログラミングの話も，システム側にやってもらいたいパラレルプロセッシングの話も，現時点ではこれらをひっくるめてプログラマが書いている．現在の並列マシンでは，ユーザとシステムを作る人の距離は離れておらず，システム作りに手を染めている人が

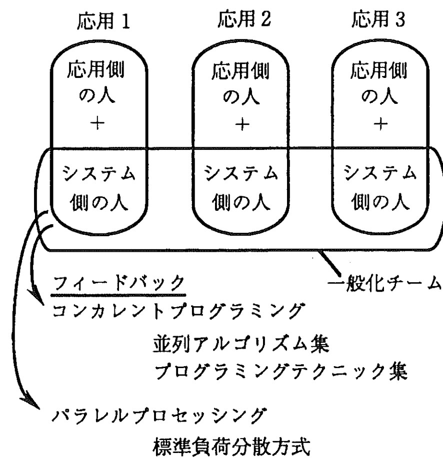


図 2.1.3 並列ソフトウェア研究の進め方

応用プログラムも書き，両方を含めた研究をしている段階といえる．

図2.1.2をもう一度眺めなおしてみると，アルゴリズムも，プログラミングの型も，負荷分散の概念も，さらにICOT のアプローチでは言語も OS も，すべて新しいことに気づく．これはもう，第五世代コンピュータの並列処理は，Computing の世界の New Culture と考えるべきではないだろうか．新しい文化をつくるというエキサイティングな仕事に，いま取り組んでいるのである．

2.1.5 研究と開発の進め方

問題を並列プログラム化しようというとき，応用問題の領域で日常の仕事をしている人たちは,「えっ，負荷分散？ 聞いたことがない」「書きたくないなあ」という．一方システムを作っている人たちは,「システム評価用にプログラムは書きたいけれど，アプリケーションには暗くて」と，ついいってしまう．これでは実用レベルの応用問題の並列化はおぼつかないし，技術の蓄積もままならない．小さくても応用プログラムを書き，走らせては結果をシステムの改良に反映して，ちょうど雪だるまをころがすように技術を太らせてゆかなければならない．そのために図 2.1.3のようなアプローチをとってきた．並列化したい応用問題があるとき，応用側の人とシステム側の人が共同研究チームを作る．応用 1，応用 2，応用 3 というように，問題ごとに共同チームを配置する．さらにそのなかに含まれるシステム側の人は，個々の応用から生まれた技術の一般化を行なうための，横断的なチームを形成する．ここでは，生まれてきたコンカレントプログラミングの技術，パラレルプロセッシングの技術を並列アルゴリズム集やプログラミングテクニック集にまとめる．また標準の負荷分散方式を OS のレベルでサポートするなど，システムに対していろいろなフィードバックがかかるように努力してきた．

一方で並列処理の研究の立場からは，どの問題を題材に研究をしようかと迷うことがある．問題の選定にあたっては,「何をやりたいか」「どうすれば解けるか」がよくわかっている問題を選ぶことが重要である．負荷分散や通信の局所化など，並列処理自体の難しい課題があるので，知識処理などで解き方のよくわからない問題をもってくると，わからないものだらけで手がつけられなくなることが多い．並列処理の研究には「早く解けるとありがたい問題」「やりたいことがよくわかっている問題」を選ぶことが重要である．並列処理で「賢く解きたい」という場合は，賢く解くための方法が定まってから並列化するのが順序であろう．解き方のよくわからないものをいきなり並列プログラム化するのは，同一人物が問題領域のプロと並列処理のプロを兼ねるときにだけチャレンジ可能な，ぜいたくなアプローチといえるだろう．

2.2 KL1 プログラミングはプロセス指向プログラミング

2.2.1 はじめに

ここでは，KL1 プログラムの設計方法とプロセス構造について基本的な事項を解説し，例をあげる．

前半では，ICOT で実際に使われていたプロセス指向のプログラム[†1]を取り上げ，はじめに並列プログラム設計方法を一般論として紹介したあと，プロセス構造の設計例を細部まで見ていくことにする．複数プロセッサを用いて動くプロセス指向プログラムの具体的イメージをつかんでいただくことを目的としている．

†1 プロセス指向プログラム：一度作られるとすぐには消滅しないプロセスがいくつも集まって，互いにメッセージ交換しながら問題を解くタイプのプログラムのこと．

ここではまだ，並列処理で性能を引き出す話は主題としない．

後半では，KL1 プログラムに出てくるプロセス構造の代表的な例を紹介する．KL1 プログラミングへの理解を深めるとともに，あとに示される応用プログラムの記事を読むときの助けともなることであろう．また，おのおののプロセス構造と関連のある負荷分散方式についても簡単に触れる．

2.2.2 プログラムの例

KL1 プログラムが並行に動作するプロセスの集まりとして記述されることは第 II 編で述べたとおりである．ここでは，複数のプロセッサを使って並列に動作するプロセス指向プログラムのイメージをパフォーマンスメータの例題をとおして示すことにしよう．

A. パフォーマンスメータ

パフォーマンスメータは，並列計算機の動作状態を実時間でモニタするために作られたプログラムで，すべて KL1 言語で記述されている．並列推論マシン実験機マルチ PSI の完成と同時に作られ，計測表示ツールが充実するまでの 3 年余りの間，愛用され続けたプログラムである．

口絵 3 に，パフォーマンスメータの表示画面を示す．小さな正方形がプロセッサの 1 台 1 台に対応しており，それぞれの色がプロセッサの忙しさに対応して実時間で変化する．口絵では，64 プロセッサの Multi-PSI 用の表示画面である．通常の利用では，画面は 2 秒に一度リフレッシュされ，直前の 2 秒間の各プロセッサの平均稼働率が，それぞれの正方形の色として表示される．

パフォーマンスメータは，プロセス指向プログラムらしいプログラムであるとともに，負荷分散プラグマ，実行優先度指定プラグマをともに使用した初期の例であり，また KL1 で非決定的な処理を記述した例にもなっている．計測表示ツールではあるが，普通のユーザプログラムと同じように起動すればよい点も特徴である．

B. プロセス構造の設計

KL1 で記述する並列プログラムの設計手順は，およそ次のようである．

1) 問題のモデル化
2) 問題解法のための並列アルゴリズム設計
3) 並列計算機への仕事割りつけ (マッピング) 方式の基本検討
4) プロセス構造の設計
5) 問題解法のアルゴリズムに関するプログラム記述 (マッピングの記述は含まない)
6) 単一プロセッサ (擬似並列実行) による実行とデバッグ (問題解法アルゴリズムに関するバグの除去)
7) マッピング設計とマッピングに関するプログラム記述
8) 並列計算機シミュレータによるテスト
9) 実機テストと実行効率のチューニング (性能に関するバグの除去)

KL1 で記述する場合の一つの特徴は，5) の問題解法のアルゴリズムに関するプログラム記述，すなわち論理的な並行性に関する記述と，7) のマッピングに関するプログラム記述，すなわち物理プロセッサへの仕事割りつけに関する記述が分離されていることである．こうなっていることのありがたさについては，あとの最短経路問題のほうがより適切な例題となっているため，そちらで詳しく述べる．ここではもう少し基本的な，プロセス構造の設計と記述について示そう．

プログラム設計手順のなかで，前述の 1)～3) は並行して進められることが多い．相互に関連が大きいからであるが，パフォーマンスメータの例では，この段階でどんな方法でプロセッサの忙しさを測定し，どのように結果を集めて表示するかを検討することになる．

通常のユーザプログラムと同様に動作することを必要条件と考えたので，プロセッサの忙しさを測定する方法としては，KL1 の実行優先度指定の機能を活用する次のような方式を取った．基本的なアイディアは，計測対象のすべてのプロセッサに，優先度が最低のプロセスを配置する．これをアイドルプロセスと呼ぶことにする．ほかのユーザプロセスは，かならずアイドルプロセスよりも高い優先度で実行されるものとする．アイドルプロセスは，ほかの意味のある仕事 (ユーザプロセスの仕事) がまったくなくなったときにだけ実行される．そこで一定期間のプロセッサのアイドル率を知りたければ，アイドルプロセスがこの期間にどれ

だけ走ったかを知れば計算できることになる．具体的には，アイドルプロセスはひたすらループを回り続けるプロセスとし，ループを回った回数を実行時間と考えることにした．一定期間ごとにループ回数を集計し報告するプロセスが必要であるが，こちらはユーザプログラムに邪魔されずに動く必要があるので，高い実行優先度を与えておく．

このような基本的アイディアに基づいてパフォーマンスメータのプロセス構造を設計したのが図2.2.1である．KL1プログラムでは，プロセス構造の設計を十分に時間をかけてやっておくことがきわめて大切である．設計したプロセス構造をKL1プログラムとして実現することはやさしいが，あとでそれを変更するのは結構な手間がかかり，バグが入る元になる．どのような役割のプロセスを配置し，その間にメッセージやデータの流れる通信路をどのように張りめぐらせるか．KL1のプロセス構造の設計は，あたかもプロセスの配置と，その間のデータが流れるパイプの配管設計をしているかのようである．

C. パフォーマンスメータのプロセス構造

パフォーマンスメータのプロセス構造は，ほとんど静的である．最初にプロセスの生成が完了すると，その後の構造は変化せず，プロセス間でデータやメッセージがやりとりされることで処理が進む．図2.2.1に示したプロセスの，それぞれの役割と動作の概要について，プログラムと対応させながら説明しよう．図2.2.2，図2.2.3にはソースプログラムの全体を示す．表示プログラムやタイマとのインタフェースは簡略化してある．以下の説明中の番号は，図中のコメントの番号に対応している．また特に断らない限り，パフォーマンスメータ以外のユーザプログラムが動作していない状態を仮定した説明となっている．

a. アイドルプロセス―(9)

最低優先度のプロセスで各プロセッサに一つずつ配置され，一度生成されると存続し続ける．より優先度の高い動けるプロセスが存在しない限り，自らを再帰呼出しして動き続ける．

```
Trigger_stream =
    [trigger, trigger, trigger, ... ]
```

という無限長のリストを生成する．一度再帰呼出しするごとにリストのcdrにtriggerという要素を1個追加するように動作する．triggerは，カウンタプロセスに対するカウントアップ要求メッセージとして機能する．普通の場合は，プロセスはもう一つ引数をよけいにもって終了条件を判定するのであるが，ここでは簡単のため省略している．

b. カウンタプロセス―(7)

高優先度のプロセスで各プロセッサに一つずつ配置され，一度生成されると存続し続ける．タイマ制御プロセスとアイドルプロセスの両方からのメッセージを待っており，通常はサスペンドしている．

アイドルプロセスからのtriggerメッセージが到着すると，自ら保持しているCountを+1したあと，再びサスペンドする．

カウンタプロセスのほうがアイドルプロセスより優先度が高いために，triggerメッセージが到着するとすぐに動く．カウンタプロセスがサスペンドすると，再びアイドルプロセスが動いてtriggerメッセージを送ってくる．このように，ほかのユーザプログラムが動いていない場合は，アイドルプロセスとカウンタプロセスは交互に動作する．この周期は，マルチPSIの場合，10.2μsである．

タイマ制御プロセスからは，指定された周期(通常は2秒)ごとにclockメッセージが送られてくる．メッセージを受け取ると，カウンタの値からプロセッサの稼働率を計算する．

まず，カウンタの値を98で割ると，アイドルプロセスが動いた時間(正確にはアイドルプロセスとカウンタプロセスが交互に動いたその合計時間)がmsで求まる．clock周期(ms)から上記の値を引いた残りが，ユーザプロセスの実行時間(有効稼働時間)となる．これから稼働率を求める．

$$\text{稼働率}(\%) = \frac{\text{clock 周期 (ms)} - \text{カウンタ値}/98}{\text{clock 周期 (ms)}} \times 100$$

この計算には(8)のプロセッサ稼働率計算ルーチンを呼んでいる．これが求まると，プロセッサ番号と稼働率の対{Pe, Work_rate}を作って，(10)のmergeプロセス経由で，(4)の出力制御プロセスへ報告する．それが済むとカウンタをクリアして再びサスペンドする．

c. merge プロセス―(10)

2本の入力ストリームから別々に送られてくる{PE, Work_rate}の対を1本の出力ストリームに送り出す

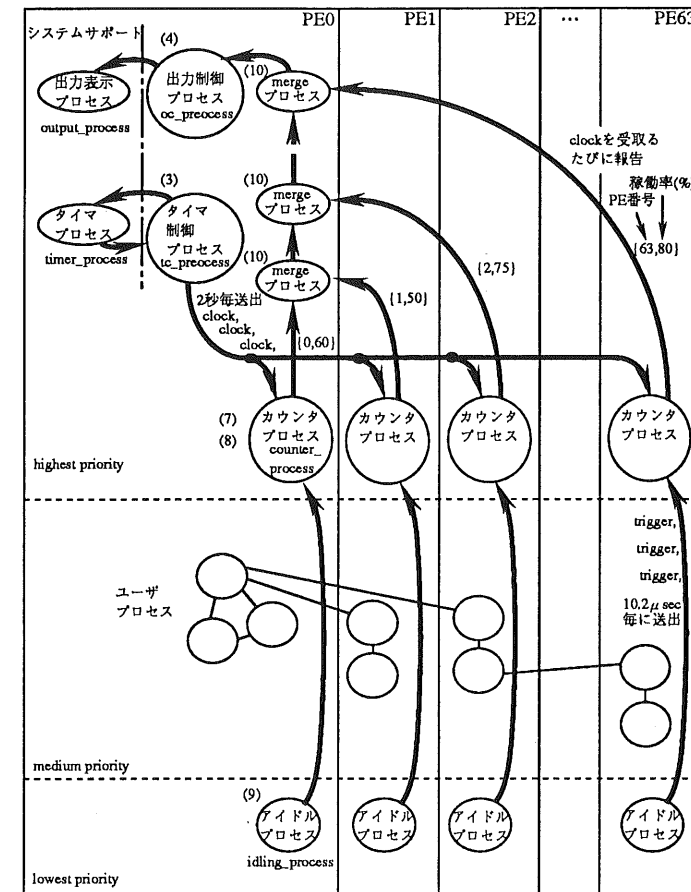


図 2.2.1 パフォーマンスメータのプロセス構成

働きをする．パイプの統合役のようなものである．ストリームからデータが来ないときはサスペンドしている．パフォーマンスメータの例では，プロセッサ数分のmergeプロセスがPE0に置かれている．実際のプログラムでは効率向上のため，ユーザ定義のmergeではなくて，任意本のストリームを統合できる組込みmerge述語を用いる．

d. 出力制御プロセス―(4)

プロセッサ数の要素をもつベクタを作り，各プロセッサからの{Pe, Work_rate}の対を受け取っては，ベクタのしかるべき位置にWork_rateを挿入する．

```
{Work_rate_0, Work_rate_1, ... ,
                    Work_rate_63}
```

ベクタが完成したら出力表示プロセスへ送る．データの来ないときはサスペンドしている．

出力表示プロセスは受け取ったデータに基づいてディスプレイに描画する．この部分は本当はOSインタフェースであり，表示プロセスはOSのプロセスとなるが，この例では簡単のために組込み述語を呼ぶと出力表示プロセスが起動されるように表現している．

e. タイマ制御プロセス―(3)

タイマプロセスとの間に双方向のストリームをもつ．タイマプロセスに{on_after, Interval, Timing}というメッセージを送ってタイマを起動する．その後，タイマ制御プロセスはTimingに値が決まるまでサスペンドする．ここでon_afterは，Intervalで指定した時間が経過すると要求元に知らせろという意味で，Intervalは通常2000msである．2秒後にタイマプロセスによってTimingの値がjust_nowにされると，タイマ制御プロセスは実行を再開し，すべてのカウン

```
%%%%%%%%%%%%%%%  Source Program of the Performance Meter  %%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%  Written by K.Taki, 1988.10.19  %%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%(1) トップレベルの呼出し．この部分は簡略化してある．%%%%%%%%%%%
:- performance_meter(64,2000,Timer_stream,Output_stream),
   timer_process(Timer_stream),          % タイマプロセスおよび出力表示プロセスの
   output_process(Output_stream).        % 呼出しは組込み述語として表現している

%%%%%%%%(2) 初期プロセス生成のトップレベル%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
performance_meter(PE,Interval,Timer_stream,Output_stream) :- true |
        timer_control(Interval,Timer_stream,Timing_stream)@priority(*,4094),
        output_control(Report_stream,Output_stream,PE)@priority(*,4094),
        process_distribution(Timing_stream,Report_stream,
                                 PE,Interval)@priority(*,4095).
%%%%%%%%(3) タイマ制御プロセス tc_process の生成%%%%%%%%%%%%%%%%%%%%%%
timer_control(Interval,Timer_stream,Timing_stream) :- true |
        Timer_stream = [{on_after,Interval,Timing}|Cdr],
        tc_process(Timing,Timing_stream,Cdr,Interval).
tc_process(just_now,Timing_stream,Timer_stream,Interval) :- true |
        Timing_stream = [clock|Tmg_cdr],
        Timer_stream = [{on_after,Interval,Timing}|Ts_cdr],
        tc_process(Timing,Tmg_cdr,Ts_cdr,Interval).

%%%%%%%%(4) 出力制御プロセス oc_process の生成%%%%%%%%%%%%%%%%%%%%%%%%%
output_control(Report_stream,Output_stream,PE) :- true |
        new_vector(Vector,PE),
        oc_process(Report_stream,Output_stream,PE,PE,Vector).
oc_process(Report_stream,Output_stream,0,PE,Vector) :- true |
        Output_stream = [{display,Vector}|Os_cdr],
        new_vector(New_vector,PE),
        oc_process(Report_stream,Os_cdr,PE,PE,New_vector).
oc_process([{Pe_no,Work_rate}|Rs_cdr],Output_stream,Count,PE,Vector) :-
        Count =\= 0 |
        set_vector_element(Vector,Pe_no,_,Work_rate,New_vector),
        New_count := Count - 1,
        oc_process(Rs_cdr,Output_stream,New_count,PE,New_vector).
```

図 **2.2.2** パフォーマンスメータのソースプログラム (その 1)

タプロセスに clock メッセージを送出する．同時にタイマを再起動し，自らはサスペンドする．

タイマプロセスは，この例では組込み述語で起動されるようになっているが，本来は OS のなかのプロセスである．その場合でも，タイマプロセスとの通信方法はここで示した例と変わらない．

以上のプロセスは，すべて初期化段階で作られ永続的に存在する．それ以外のプロセスは初期化のためなどに一時的に存在するものであるが，重要な部分についてだけ触れておこう．

f.　トップレベルの呼出し—(1)

performance_meter の第 1, 第 2 引数は，プロセッサ数とタイマ周期 (ms) である．timer_process と output_process の呼出しは，組込み述語によりシステムの組込み機能を使用しているつもりである．トップレベルは PE0 で起動されると仮定している．

g.　初期プロセス生成のトップレベル—(2)

ここでは 3 種類の初期化プロセスを優先度つきで起動している．優先度指定は@priority(*, 4095) のように指定しているが，これは優先度を絶対値で指定す

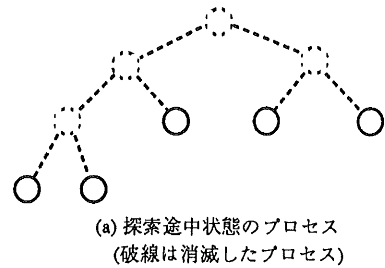

```
%%%%%%%%(5) 各プロセッサへの初期化プロセスの配置と merge プロセスの生成%%
process_distribution(_,Report_stream,0,_) :- true |
        Report_stream = [].
process_distribution(Timing_stream,Report_stream,Count,Interval) :- true |
        New_count := Count - 1,
        merge(S1,S2,Report_stream),
        measure(Timing_stream,S1,New_count,Interval)@node(New_count),
        process_distribution(Timing_stream,S2,New_count,Interval).

%%%%%%%%(6) 各プロセッサにおけるプロセスの初期化%%%%%%%%%%%%%%%%%%%%%%%%%%
measure(Timing_stream,Report_stream,Pe,Interval) :- true |
        counter_process(Timing_stream,Trigger_stream,0,
                                Report_stream,Pe,Interval)@priority(*,4093),
        idling_process(Trigger_stream)@priority(*,0).

%%%%%%%%(7) カウンタプロセス%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
counter_process([clock|Ts_cdr],Trigger_stream,Count,
                                Report_stream,Pe,Interval) :- true |
        Exec_time_by_ms := Count / 98,
        work_rate(Exec_time_by_ms,Interval,Work_rate),
        Report_stream = [{Pe,Work_rate}|Rs_cdr],
        counter_process(Ts_cdr,Trigger_stream,0,Rs_cdr,Pe,Interval).
counter_process(Timing_stream,[trigger|Trgs_cdr],Count,
                                Report_stream,Pe,Interval) :- true |
        New_count := Count + 1,
        counter_process(Timing_stream,Trgs_cdr,New_count,
                                    Report_stream,Pe,Interval).

%%%%%%%%(8) プロセッサ稼働率計算ルーチン%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
work_rate(Exec_time,Interval,Work_rate) :- Exec_time >= Interval |
        Work_rate = 0.
work_rate(Exec_time,Interval,Work_rate) :- Exec_time < Interval |
        Work_rate := ((Interval - Exec_time)*100)/Interval.

%%%%%%%%(9) アイドルプロセス%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
idling_process(Trigger_stream) :- true |
        Trigger_stream = [trigger|Ts_cdr], idling_process(Ts_cdr).

%%%%%%%%(10)merge プロセス%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
merge([X|XT],Y,Z) :- true | Z=[X|ZT],merge(XT,Y,ZT).
merge(X,[Y|YT],Z) :- true | Z=[Y|ZT],merge(X,YT,ZT).
merge([],Y,Z) :- true | Z=Y.
merge(X,[],Z) :- true | Z=X.
```

図 **2.2.3** パフォーマンスメータのソースプログラム (その 2)

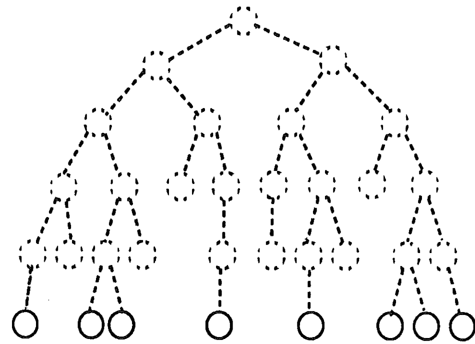
る書式である．実行優先度は，0 から 4095 の 4096 段階で指定できる (言語処理系のパラメータ設定により変更可能)．一度与えられた優先度は，再指定されるまでは，その子孫のゴール (プロセス) に対して継承される．

h.　プロセッサへの初期化プロセスの配置—(5)

各プロセッサにおける初期化プロセス measure(6) をプロセッサに配置するとともに，merge プロセスを PE0 に，プロセッサの個数分だけ作る．プロセッサにプロセスを割りつけるのは@node(New_count) で指定



(a) 探索途中状態のプロセス
(破線は消滅したプロセス)



(b) 解に到達したプロセス

図 2.2.4 全解探索のプロセス構造

している．New_count で PE 番号を指定する．指定されたプロセッサに割りつけられたプロセスの子孫 (そこから派生したプロセス) は，新たな指定がない限り同じプロセッサに留まる．

i. 各プロセッサでのプロセスの初期化—(6)

ここでは，カウンタプロセスを優先度 4093 で，アイドルプロセスを優先度 0 で起動している．ユーザプログラムのプロセスは，優先度 1 から 4092 の範囲で動くことを仮定している．

これらの一連の初期化プログラムのなかで，プロセスの初期配置とともにそれらの間のメッセージストリームの接続が行なわれている．

プロセス構成図 (図 2.2.1) とプログラムリスト (図 2.2.2,2.2.3) を見ながら，パフォーマンスメータプログラムの構造と動きのイメージをつかんでいただけただろうか．先に，あたかも配管設計をしているかのようだといった意味が，おわかりいただけたと思う．逐次プログラムを書いてきた者にとって，この「配管設計」は実に新鮮であり，うまく動いたときの喜びは格別である．この設計段階にしっかり時間をかけておくならば，そのあとの KL1 コーディングは難しいものではない．

2.2.3 プロセス構造のいろいろと負荷分散

KL1 を用いて並列の応用プログラムをいろいろ書いてみた経験から，そのなかで出会ったプロセス構造の主要なものをまとめてみた．プロセスの生成および接続の構造に対して，それと相性のよい負荷分散方式があるようで，そのことについても触れることにする．

A. 動的木構造

与えられたデータに基づいて探索を行ない，処理が進むにつれて動的に探索木を展開してゆくタイプである．プロセスは，探索木の各節点に対応する．実行してみるまで，木のどの枝が大きく成長するかわからないことが特徴である．このタイプについては 2 種類紹介する．

a. OR-並列型全解探索

パズルの全解法を求める問題などがこれに当たる．次節のペントミノはその具体例である．このタイプは，木の根から最終的な葉に向かってひたすら枝を展開すればよく，兄弟の節点間では特別な情報交換を必要としない．最終的な葉に到着するとそれが解となる．

したがってこのタイプでは，節点にあたるプロセスが子孫のプロセス (子の節点) を生成すると，自らは消滅してかまわない．子の節点同士での情報交換が不要だからである．つまりは，生きているプロセスは常に探索の最前線 (そのときどきの葉) の部分にしか存在せず，木探索を行なっているにもかかわらず，生きているプロセスは木の形状には配置されていないことが特徴である (図 2.2.4)．ただし，最終的な葉のプロセスから解を回収する必要があるため，生きているすべてのプロセスから解回収プロセスに向かってのストリームを動的に張ることが多い．

このタイプの問題では，何らかの動的負荷分散を必要とする．実行しないことには負荷の大きさがわからないためである．動的負荷分散とは実行時に仕事の割りつけを決めることであるが，これには二通りがある．プロセッサの忙しさを何らかの方法で知って，忙しくないプロセッサに仕事を与える適応的 (adaptive) 動的負荷分散と，プロセッサの忙しさを知ることはしないが負荷の均等化を図るような何らかの方法により仕

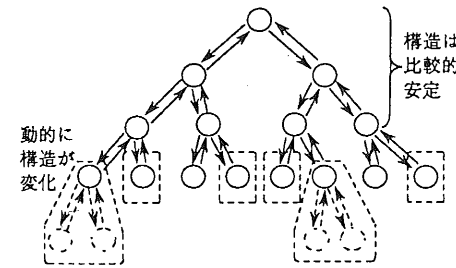


図 2.2.5 ヒューリスティクサーチのプロセス木構造とストリーム接続

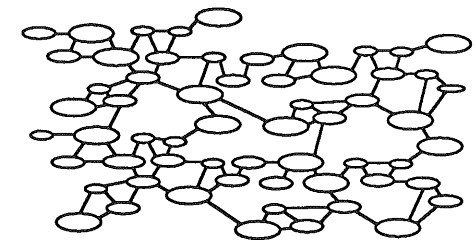


図 2.2.6 プロセスネットワーク構造

事を割りつける (たとえば部分問題の数をプロセッサ数より十分多くしておいて，乱数を用いて割りつけを決めるような) 非適応的 (non-adaptive) 動的負荷分散がある．ペントミノでは，初期のプログラムにおいて 2 種類とも試みたが，あとの版では多階層の適応的動的負荷分散を用いて良好な結果を得ている．

b. ヒューリスティックサーチ

前例とは異なり，探索木の兄弟の節点間で情報交換が必要なタイプである．ある部分木での探索の中間的な評価値を別の部分木の探索に反映し，探索を進めるか枝苅りをするかなどの判断に使用する．このため，子孫の節点を生成したプロセスも消滅することなく存続し，プロセス間で木構造をそのまま反映したメッセージ通信路をもつ場合が多い (図 2.2.5)．

このタイプでは，探索木のそのときどきの葉に当たる部分では，プロセスの動的な生成消滅が繰り返されるが，それより根に近い部分のプロセス構造は安定している．その部分のプロセスは，通常はサスペンドしていて，評価値の交換のときだけ起きだして動くという特徴をもつ．このタイプの例としては，ICOT で開発した詰め碁がある [1]．詰め碁ではゲーム木の探索を行ない，アルファ・ベータ枝刈り法の並列化を実現した．この場合も，先に述べた 2 種類の動的負荷分散を適用した．

B. ネットワーク構造

プロセスネットワークを構成する応用例は多いと考えられる．並列オブジェクトモデルに基づくプログラムは基本的にこの構造を取るし，それ以外にもいろいろ扱ってきた．ここでは 3 種類を紹介する．

a. 静的ネットワーク

実行前からネットワーク構造が静的に決められる場合である (図 2.2.6)．パフォーマンスメータはこの例であった．また並列オブジェクトモデルに基づくプログラムのうち，オブジェクトの数と接続構造が実行前から決まっている場合がこれに当たる．あとの例では，最短経路問題がそうであり，論理シミュレーションのもともとのモデルもこれに該当する．オブジェクト指向にはなっていないが，遺伝子配列解析の 3 次元 DP マッチングも，静的なプロセスネットワーク構造を取る．

これら 3 種のプログラムは，いずれもプロセスの個数がたいへん多く，数万以上に達するという特徴をもつ．これらに対しては，基本的に静的負荷分散を適用してきた．ネットワークを，近傍接続関係をもつ部分ネットワークに分割しておき，それらをプロセッサに割りつける．部分ネットワーク間で計算負荷がばらつく場合には，プロセッサ数に比べて部分ネットワークの個数が十分大きくなるように分割し，それらを乱数を用いてプロセッサ割りつけする．詳細はプログラムの説明のなかで述べる．

b. 部分動的ネットワーク

静的なプロセスネットワークの一部において，動的なプロセスの生成消滅を発生する場合である．LSI 配線がこの例に当たる．この場合，静的負荷分散に加えて，生成消滅するプロセスに対しては，動的負荷分散を適用する必要がある．LSI 配線では非適応的方法を使用している．

c. レイヤードストリーム

動的なネットワークの一種であるが，メッセージストリームのなかにストリームが多重化して流れてくる特徴をもつ．その多重状態自体がデータ表現の一部であり，多重化されたストリームを取り出してはそれに対応した処理プロセスを動的に生成する．あとの例では，自然言語解析がこの手法を取っている．データになるべく近い場所にプロセスを動的に生成するような

負荷分散方式を試みたが，通信の局所性を高めることはやさしくない．

C. データ分割または SPMD

このタイプは，データを分割してうまくプロセッサに配置することが，仕事を分割・配置することと1対1に対応する場合である．同じ働きのプロセスが各プロセッサに1個ずつ存在し，配置されたデータに対して各プロセッサで同じプログラムが動く．仕事の途中結果を交換する必要があるときは，これらのプロセス間で通信が起こる．SPMD とは Single Program, Multiple Data の略である．数値計算を分散メモリ構造の並列計算機で実行する場合，多くはSPMD型のプログラム構造を取る．

あとに出てくる例では，データベース，法的推論中の事例ベースが，基本的にこの考えに基づいてプログラムされている．すなわち，データベースなどのデータを分割し複数のプロセッサに配置することによって，検索の並列処理などを実現している．論理シミュレーションは，並列オブジェクトモデルに基づいて設計され，初期の版ではプロセス構造が静的ネットワークとなるように実装されたが，後の版ではデータ分割型に変更された．すなわち，各プロセッサにはシミュレーションエンジンを司るプロセスが1個ずつ存在し，そこへゲートのデータを分割配置している．この変更は，シミュレーションの基本性能を少し向上させるのに役立った．

プログラム経験からいうと，このタイプの構造をもつプログラムでも，実はプロセッサ間で非対称な処理をする必要性はときどき発生し，そのような部分の記述には，実は並列オブジェクトモデルのほうが適するように思われる．対称な処理しかない場合には，C言語に通信と同期を入れたような言語で書いても大した不便はなさそうであるが，並列オブジェクト的に書きたい部分があちこちに散りばめられているようだと，KL1 でなければ書ききれなくなると考えられる．たとえば論理シミュレーションでは，シミュレーションエンジンの基本的な部分は SPMD 型プログラムといえるが，大域的な時刻合せを行なう部分の記述は，並列オブジェクトモデルのほうが適しているのである．

これまでに手掛けた応用プログラムに現われる主なプロセス構造と，そのプロセス構造と相性のよい負荷分散方式について見てきた．プロセス構造を整理するという意味では，すべてを尽くしたことにはなっていないが，多くの応用プログラムのプロセス構造が，これらのいずれかのタイプまたは，複数のタイプの組合せとして分類できると考えている．

2.3 ペントミノと動的負荷分散

2.3.1 はじめに

前節では，並列プログラムを設計するための基本事項と，KL1 による並列プログラムのプロセス構造について基本的な説明を行なった．本節以降ではいよいよ，並列実行で高速処理を実現するためのプログラミングについて，具体例を用いて紹介していく．

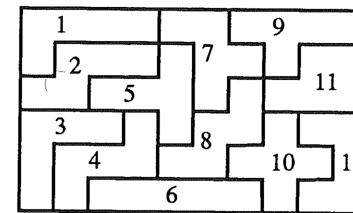
ここでは，詰込みパズルまたはペントミノと呼ばれるパズルの例を用いて，まず全解探索を行なう並列プログラムの設計について紹介する．これは探索問題としては最も単純で並列化しやすく，並列処理の入門用例題として適当なものである．次に，並列処理で高い効率を実現するための仕事の割りつけ方法として，動的負荷分散方式とそのプログラム設計について説明する．これも負荷分散の考え方としては基本的なものの一つであるが，KL1 のプロセス指向プログラムとして実現する方法には興味深いところがある．負荷分散に関してはさらに，処理のオーバヘッドや，総合的な計算の効率についても考える．最後に並列処理性能の計測結果を示し [2]，性能の表現方法について実例とともに解説する．

2.3.2 ペントミノ または 詰込みパズル

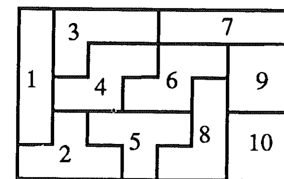
A. 問題とその性質について

詰込みパズルは，玩具屋でプラパズルという商品名で売られている．図 2.3.1に示すように，いろいろな形をしたプラスチックのピースを長方形の箱に隙間なく詰め込むパズルである．ピースの形状や大きさにはいろいろあるが，図 2.3.1(a) はペントミノと呼ばれるもので，ピースの形が12種類あって，いずれも1辺の大きさが1の正方形を5個集めてできる図形になっている．これを大きさ 6×10 の長方形の箱に詰め込む．

正方形4個でできる形状のピースを使うのはテトロ



(a) ペントミノ



(b) テトロミノ

図 2.3.1 詰込みパズル

ミノである．5種類のピース各2個，合計10個を大きさ 5×8 の箱に詰めるパズルとして売られている (図 2.3.1(b))．あとで計測評価に使う例題はこれである．

これらを計算機に解かせる場合，可能なすべての解 (詰め込み方) を求める全解探索問題として与えることができる．解き方の一例をあげると，ペントミノの場合では，12個のピースから1個を選んで，まず箱の左上隅に置くことを考えるが，これだけで12通りある．選んだピースの回転および裏返しによって，さらに場合の数が増える．1個置くと，次のピースを選ぶ．このように，ピースを一つ箱に詰めることが，探索を1段深めることに相当する．異なるピースの選び方および置き方があるとき，それぞれは，別の解に至る探索の出発点となる．すなわち，独立な部分探索空間を構成する．こうしてできる探索木は，OR 木で表わされる．部分木の探索をまったく独立に行なってよいことが特徴である (図 2.3.2)．

新たなピースを選び，回転と表裏を決めると，たとえばそれをすでに置いてあるピースに接して，なるべく左寄りに，そして上寄りの位置に置く．これを次々に繰り返して，箱がピースで埋まったらそれが解である．また，途中でピースが置けないようになったなら，その探索枝は解に到達しないまま終了である．探索の深さが深まるとともに OR 木の幅は次第に広がる．ただし，ピースが置けなかった枝はそれ以上伸びないので，ある深さを境にして OR 木の枝の数は次第にまばらになる．

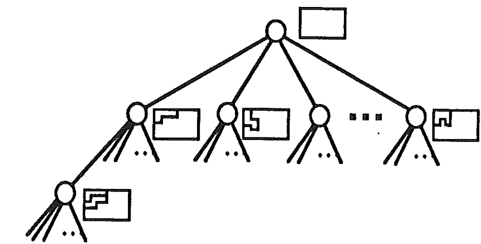


図 2.3.2 ペントミノの解探索と OR 木

B. 基本的なプログラムの設計

問題のモデル化と基本的な問題の解き方については，厳密ではないが前項ですでに示した．次は，問題を並列に解くための並列アルゴリズムの設計に移ろう．

a. 並列化の方針

OR 木の探索は，逐次プログラムなら，メモリ消費が少ないので深さ優先で行なうのが普通である．けれども OR 木の場合，兄弟の節点の下の部分木の探索は，互いにまったく独立に行なっても何ら問題がない．そこで並列化に当たっては「部分木の探索を並列化する」という方針が簡単に思い浮かぶ．しかしながらペントミノの場合，1段探索を深めると，節点の数は10倍以上に増えるので，部分木のどの部分を並列実行の対象とするか悩むことになる．つまり，探索木の浅いところで部分木に分けるならば大きな部分木が少ない数できるし，深いところの部分木を並列実行の対象と考えるならば，小さな部分木を多数取り扱うことになる．

b. 並列アルゴリズムとプロセス構造

KL1 のありがたさは，ここで悩まなくてよいことである．問題を解くための並列アルゴリズムの設計段階では，最終的にどのような単位で並列実行するかはさて置いて，並列に実行させたいかもしれない最も細かい単位を意識して，アルゴリズムの設計をしておけばよい．ここでは探索木の各節点での処理，すなわち選ばれたピースが置けるかどうかを確かめて，置けたならば次のピースの候補を生成する，という処理を最小の単位と考えてみる．そうすると，探索木の末端まで，意味上は生成された節点ごとにすべて並列に動作可能なアルゴリズムになる．ここでいう並列処理の最小単位は，KL1 のプロセスとして自然に表現することができ，その場合，探索を1段深めるごとにピースの置き方の場合の数だけプロセスがフォークする実行モデルとなる．

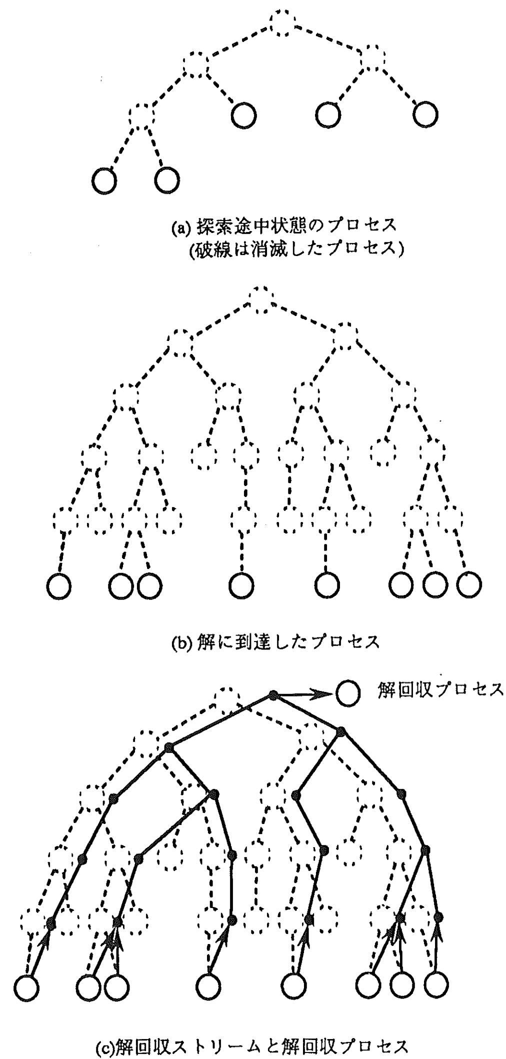


図 2.3.3　ペントミノのプロセス構造

c.　負荷分散の基本方針の検討

並列実行のことはさて置いてと書いたが，そのことをまったく考えていないとあとで対処のしようがなくなる場合もあるので，上記の方法でうまくプロセッサにマッピングできるかの検討はしておく必要がある．たとえば，探索木の適当な深さを選んで，その深さに作られるプロセスを別々のプロセッサに配置し，それらのプロセスの子孫たちは配置された同じプロセッサに留まる，といった方針をとることにより，部分探索木単位の並列実行が実現できる．ここではその程度の見通しでよいことにする．

ここまで来れば，あとはプロセス構造の詳細を設計し，KL1プログラムを記述することができる．

d.　解の回収について

プロセス構造の基本は，探索木に従ってプロセスがひたすらフォークを繰り返すものであることは先に述べた．ピースが置けなかった節点に対応するプロセスはフォークせずに自ら消滅する．最後のピースを置くことができた節点は解を得たことになるが，問題はこれを表示するなりファイルに書き出すなりするために一箇所に集める方法である．よく用いる方法は，解の回収をするプロセスを一つ用意して，探索木の葉のプロセスすべてから，この回収プロセスへストリームを張る方法である (図 2.3.3)．

一つの節点プロセスは，子のプロセスを一つ生成するたびに，解回収用ストリームを分岐しては生成した子プロセスにわたす必要がある．こうしないと，最終的にすべての葉プロセスに回収用ストリームが行きわたらない．ストリームの分岐点にすべてユーザ定義 merge 述語を置いたのでは，探索木と同じ形の merge プロセスの木構造ができてしまい，解が回収プロセスに到達するまでに多くの merge プロセスが動くことになって効率が悪い．そこで，組込みの merge 述語を用意し，1個の組込み merge プロセスだけで任意本数のストリームを1本に束ねられるようにしている．

これでプロセス構造は決まり，負荷分散部分を除いてプログラム化することができる．プログラムができると，単一プロセッサ用の擬似並列実行を行なう処理系で実行して，テストおよびデバッグする．問題解法のアルゴリズムにかかわるバグは，この段階で取り除くことができる．

2.3.3　要求駆動的負荷供給（負荷分散）方式

次はいよいよ，プロセスをプロセッサにマッピングしていかに並列実行させるかを決める負荷分散の設計である．まずはじめに，適応的な動的負荷分散の一方法を紹介する．

A.　基本的な考え方

この方式は，適当な探索深さから下にある部分探索木おのおのの探索をサブタスクとして，仕事をもたないプロセッサに動的に割り当てるものである．負荷分散の制御は中央集権的である．すなわち，探索問題全体（タスク）をサブタスクに分割するとともに，負荷分散の管理者としても働く特別のタスクを一つ置く (KL1で書くとプロセスの集合となるのでタスクと表現した)．このタスクが部分探索木に当たるサブタスクを生成して，仕事のないプロセッサに動的に割りつけるのである．

B.　サブタスクの生成

サブタスクの生成を行なう特別のタスクをここではサブタスクジェネレータと呼ぶことにしよう．サブタスクジェネレータは，特定の PE（マスタ PE と呼ぶ．PE は Processing Element の略でプロセッサのこと）で実行され，図 2.3.4のようにサブタスクを生成する．図中，大きな三角形は OR 木で表わされる探索空間全体を示す．縦棒で塗りつぶされた小さな三角形はサブタスクジェネレータを示し，探索の深さがある値（負荷分散（開始）レベルと呼ぶ）に達するまでマスタ PE 上で実行される．円は生成されたサブタスクを示し，その大きさがサブタスクの計算量，すなわち粒度を表わす．サブタスクの粒度は，負荷分散レベルを深くするほど小さくなる．サブタスクは部分探索木の探索に対応するため，その粒度は一般的にばらつくのが普通である．各サブタスクは，各プロセッサにかかる負荷を均等化するためのしかるべき戦略によって，各プロセッサに割りつけられる．

C.　サブタスクの粒度について

サブタスクの計算量の大きさ，すなわち粒度に関しては，並列処理性能を高める上で，相反する二つの要求が存在する．

a.　負荷バランスに関する要求

まず，多くの PE をすべて忙しく働かせるとともに，各 PE の忙しさをなるべく均等化することが要求される．これを**負荷バランス**という．負荷バランスを取りやすくするには，サブタスクの数を多くしなければならない．サブタスクの数が PE 数程度しかないと，タスクの粒度のばらつきが PE の忙しさのばらつきとなって，平均性能は低下してしまうからである．サブタスク数を多くすることは，一般に粒度を減少させることである．

b.　サブタスク供給に関する要求

一方，サブタスクの生成時間と比較して，粒度（サブタスクの実行時間）をある程度以上大きくすること

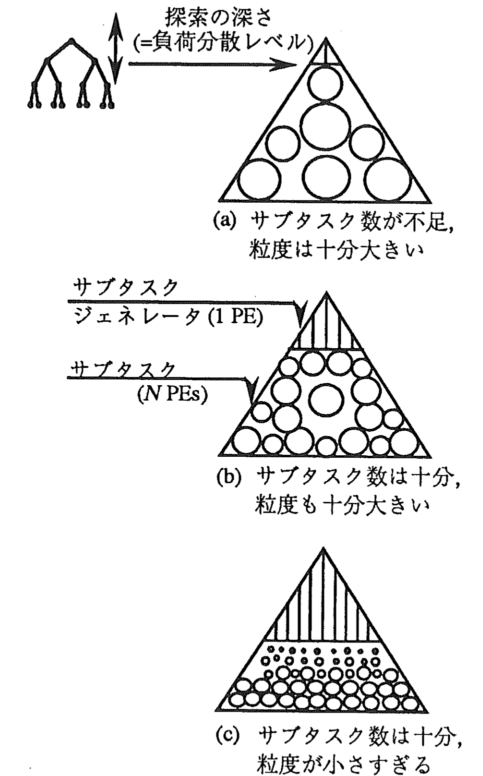


図 2.3.4　サブタスクの生成

が重要である．さもないと，多くの PE にサブタスクを分配するのにサブタスクの供給が追いつかない．すなわち，**負荷供給ボトルネック**となって，仕事をもらえずに待つ PE が生じ，平均性能は低下してしまう．

c.　負荷分散の手間に関する要求

また，負荷分散にかかる手間（時間）はシステムに依存するが，この手間に比べて粒度をある程度以上大きくしないと，**負荷分散のオーバヘッド**により性能が押えられる．たとえば1の時間で終了するサブタスクを他の PE に割りつける処理に，1以上の時間が掛かるようでは，他の PE にやらせずに自ら実行したほうが得になってしまう．

このように，二通りの理由により，粒度を大きくする必要がある．

図 2.3.4(a) の場合は，サブタスクの粒度が十分大きいので負荷供給ボトルネックはないが，サブタスク数が少なく，粒度のばらつきのために負荷バランスが取れず十分な性能が得られない．また図 2.3.4(c) は，サ

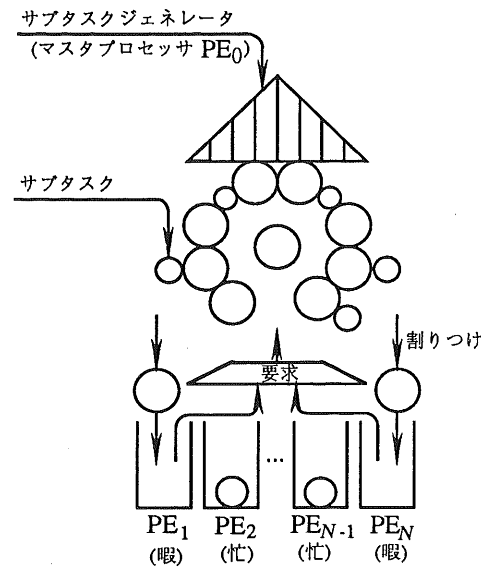


図 2.3.5 要求駆動による動的負荷分散

ブタスクの数は十分で負荷バランスは取りやすいが，粒度が小さすぎてサブタスクの供給がボトルネックとなる場合である（ここでは負荷分散の手間は十分小さい場合の説明とした[†1]）．したがって，よい並列処理効率を得るには，図 2.3.4(b) のように，チューニングによって適切な粒度，すなわち適切な負荷分散レベルを決める必要がある．

D. サブタスクの割りつけ

前節のように生成したサブタスクは，各 PE の負荷が均等化するように，仕事のなくなった PE に対して動的に割りつけられなければならない．これには，仕事のなくなった PE から，マスタ PE のサブタスクジェネレータに対して，新たなサブタスクを要求するメッセージを送る方式をとっている．これを要求駆動による負荷供給（負荷分散）方式と呼んでいる (図 2.3.5)．処理手順は次のとおりである．

1) 初期状態では全 PE は暇であり，マスタ PE 上のサブタスクジェネレータにサブタスク要求メッセージを送る．
2) 要求を受けたサブタスクジェネレータは，要求元の PE にサブタスクを割りつける．
3) 割りつけを受けた PE ではサブタスクを実行し，終了すると新たなサブタスク要求メッセージを送出する．

†1 システムの造りにより負荷分散の手間が相対的に大きい場合には，負荷供給ボトルネックではなく，負荷分散のオーバヘッドにより性能が頭打ちとなる．

PE が暇になってからサブタスクが割りつ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実行時間に比べて無視できない場合は，サブタスクのダブルバッファリングを行なえばよい．

2.3.4 多階層動的負荷分散

A. サブタスク生成ボトルネック

前節で示した要求駆動方式による動的負荷分散は，プロセッサ台数の拡張性に乏しいという欠点がある．すなわち，プロセッサ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個数が増加する．その数がサブタスクジェネレータで単位時間に生成できるサブタスクの個数を越えると，供給ボトルネックの状態となり，プロセッサ台数をこれ以上増やしても，並列処理性能は頭打ちとなる．

B. 多階層動的負荷分散方式

このボトルネックをなくすには，一つのサブタスクジェネレータに対してサブタスク要求を出すプロセッサの数を制限すればよい．それを実現する方法として，サブタスクジェネレータを分割し，複数のプロセッサで実行する次の方式を試みた．図 2.3.6は，サブタスク生成を 2 段階で行なう方法の概念図である．スーパーサブタスクジェネレータは，複数のスーパーサブタスクを生成して，あらかじめ定められたプロセッサグループ (後述) に対して動的に割りつける．そこで実行されるスーパーサブタスクとは，実はサブタスクジェネレータのことであり，サブタスクを生成してはグループ内のプロセッサに動的に割りつける．

a. 第 1 段階の負荷分散

この様子をもう少し詳細に見よう (図 2.3.7)．まずスーパーサブタスクジェネレータは，マスタ PE に割りつけられ，探索が第 1 負荷分散レベルに達するまで処理を進め，探索問題全体をスーパーサブタスクに分割する．これを第 1 段階の負荷分散と呼ぶ．ここで，全体で N台の PE は M個のプロセッサグループ (PG) に均等に分けられる．各グループ中の特定の PE はグループマスタと呼ばれる．第 1 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によって，暇なグループマスタ PE に割りつけられ，グループ間での負荷の均等化が行なわれる．

このとき，スーパーサブタスクジェネレータでの供給ボトルネックが起こらないように，グループの個数を決める必要がある．

b. 第 2 段階の負荷分散

次にグループマスタ PE に割りつけられたスーパーサブタスクは，第 2 負荷分散レベルに達するまで処理を進め，スーパーサブタスクをサブタスクに分割する．これを要求駆動方式を用いて，グループ内の PE に動的に割りつける．これを第 2 段階の負荷分散と呼ぶ．

このときも先と同様に，供給ボトルネックが発生しないようにグループ内の PE 数を決める．

ここでは簡単のため 2 階層負荷分散の例を説明した．プロセッサ数がさらに多くなって 2 階層でも供給ボトルネックが発生する場合には，同様の方法で多階層の負荷分散へ拡張することができる．

C. グループマージ

多階層動的負荷分散は，プロセッサ台数の拡張性に富んでいるが，スーパーサブタスクの個数が不十分なときは，グループ間で負荷の不均衡が生じることがある．すなわち，あるプロセッサグループではスーパーサブタスクもサブタスクもすべて終了したのに，別のプロセッサグループではすべての PE がまだ忙しく仕事をしているという状況が生じ得る．

この問題を軽減するために，まったく暇になったグループの PE をばらばらにして，忙しいグループに併合する方法をとっている．これをグループマージと呼んでいる [2]．

2.3.5 負荷分散のプログラミング

A. 負荷分散コードは追加するだけで

前節で述べた多階層動的負荷分散方式を，先に示した詰込みパズルに適用した．詰込みパズルの並列プログラムは，探索木の根から葉に向かって木の形状に従い，プロセスがひたすらフォークする構成を取ることは 2.3.2節で述べたとおりである．そこでは，部分探索木の探索は複数のプロセスから構成されるが，これを一つのサブタスクとして扱い，負荷分散の対象としなければならない．

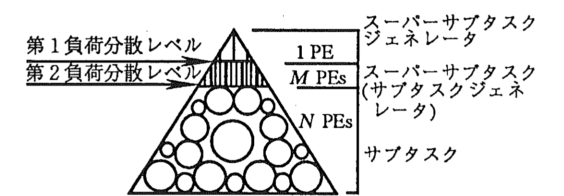


図 2.3.6 2 段階のサブタスク生成

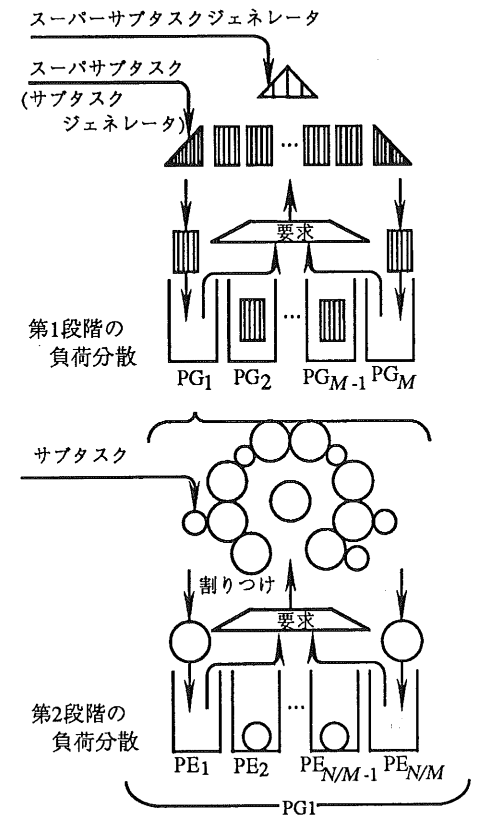


図 2.3.7 2 段階の動的負荷分散

幸いなことに木探索の場合は，部分木の根に当たる節点プロセスをサブタスクそのものだと思って対象プロセッサに割りつけることにより，その子孫のプロセスは，特に指定しない限りすべて同じプロセッサ上に生成され実行される．

本来望ましいのは，先に設計した詰込みパズルの並列プログラム（負荷分散は含まないもの）には変更を加えずに，負荷分散コードの追加と負荷分散プラグマの付加だけで，負荷分散ありの完全なプログラムができ上がることである．しかしながらこの例では，多階層動的負荷分散を実現するために，負荷分散レベルを

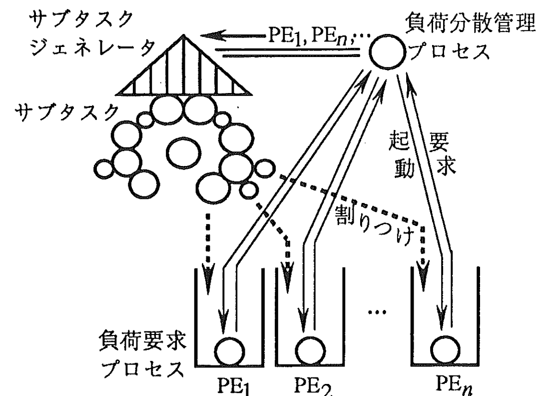


図 2.3.8 負荷分散管理プロセスの接続構造

プログラム中で知る必要があった．該当レベルに当たるときは，プロセスを指定プロセッサに配置するコードを呼ぶような変更を行なう必要があった．このための主な変更は次の 3 点であった．

1) 節点プロセスを表わすゴールに探索深さを示す引数を追加すること
2) 1) が指定レベルであれば別コードを呼ぶこと
3) 負荷分散管理プロセスと通信するためのストリームも引数にもたせること

しかしながら基本的なプロセス構造を変える必要はなかったため，修正は難しくなかった．

負荷分散にかかわるそのほかの部分は，すでに書いたプログラムの変更ではなく，コードの追加として実現することができた．

B. 暇なプロセッサの検出方法

プロセッサでサブタスクの処理が終了し，ほかにする仕事がなくなったならば，すなわちプロセッサが暇になったならば，スーパーサブタスクに対して新たな仕事を要求しなければならない．これを実現するのに，木探索のプログラムには極力変更を加えたくなかったため，パフォーマンスメータのところで述べたアイドルプロセスと同様の考え方を用いることにした．

サブタスクが割りつけられる側のプロセッサには，あらかじめサブタスクより低い実行優先度をもつ負荷要求プロセスを配置しておく．このプロセスは，動くよう指令を受けると，自分より優先度の高いプロセスの実行が終了したあとに動きだし，サブタスクジェネレータに対してサブタスク要求メッセージを 1 個だけ送出する．サブタスクジェネレータは要求への返答として，サブタスクの割りつけを行なうとともに，負荷要求プロセスに再度動作指令を出す．割りつけられたサブタスクは優先度が高いので先に実行される．実行が終わると，また負荷要求プロセスが動き出し，新たな要求メッセージを 1 個送る．

スーパーサブタスクの負荷分散についても同じ方法が適用できる．負荷分散を階層化する際に実行優先度の利用も階層化しておくと，同じプロセッサにスーパーサブタスクの実行とサブタスクの実行の役割を重ねて与えることができる．この場合，優先度の高い順に，スーパーサブタスク，スーパーサブタスクに関する負荷要求プロセス，サブタスク，サブタスクに関する負荷要求プロセスとなる．当然ながら，スーパーサブタスクの割りつけと実行が優先される．

C. 負荷分散管理プロセス

サブタスクジェネレータのなかにあって負荷分散の制御を司るのが負荷分散管理プロセスである．負荷分散管理プロセスは，各プロセッサに置かれた負荷要求プロセスからの要求メッセージを受け取り，メッセージ中のプロセッサ番号を取り出す．負荷要求プロセスは，この管理プロセスとしか通信しないところがミソである (図 2.3.8)．

一方, 木探索プロセスも負荷分散管理プロセスとの通信路をもっている．木探索が負荷分散レベルに達すると，普通は探索木の子節点に当たる子プロセスをただ生成する代わりに，負荷分散プラグマを付加して生成する．付加されたプラグマ中で，プロセスの移動先を示す変数に値が定まっていないと，プロセスの割りつけは待たされる．負荷分散制御はこの性質を巧みに利用している．

負荷分散レベルに達した節点プロセスでは，プラグマを付して子プロセスを生成するが，プラグマ中のプロセス移動先を示す変数を負荷分散管理プロセスにわたし，移動先を決めてくれるよう依頼する．管理プロセスでは，負荷要求メッセージが届くと，メッセージ中のプロセッサ番号を取り出して，生成された子プロセスの移動先とする．移動先が決まると，子プロセスは直ちにそのプロセッサへ移動する．すなわち割りつけが行なわれる．負荷要求メッセージがなかなか来ないときは，移動先の決まらないままの子プロセス (サブタスク) がたくさん溜って，割りつけられるのを待つことになる．

表 2.3.1 1 段階負荷分散の並列処理性能

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
効率 (%)	L3	100	88.8	72.5	48.4	30.8

このように，もともとの木探索を行なうプロセスは，負荷分散レベルに達したときにだけ，ここで述べた負荷分散管理プロセスだけとの間で通信することによって，負荷分散を実現している．木探索に関するプロセスの構造をほとんど変えないまま，負荷分散の機能を追加することに成功したといえる．

多階層動的負荷分散の場合は，管理プロセスと負荷要求プロセスの組を階層の数だけ用意して異なる優先度を与えておき，木探索プロセスは負荷分散レベルに達したときだけ，対応する管理プロセスと通信して負荷分散を依頼すればよい．

この方式による多階層動的負荷分散プログラムは，木探索タスク本体との独立性が高いため，後に負荷分散ライブラリとして分離され，PIMOS オペレーティングシステムの一部として供給されるようになった．

2.3.6 処理性能の計測

処理性能の計測は，64 台構成の Multi-PSI 上で行なった．パズルの問題は，図 2.3.1のテトロミノを用いた．プロセッサ台数を変化させたときの実行時間を測定し，台数効果と効率を求めた．

a. 性能の表現方法について

実行時間は，計算の開始からすべてのプロセッサでの計算が終了するまでの時間である．仕事が均等に分散されず，遊ぶプロセッサが出てくると，当然ながら実行時間は長くなる．また本来するべき計算以外に，通信のための処理に時間がかかる場合も，実行時間は伸びる．

台数効果はスピードアップとも呼ばれ，プロセッサ 1 台による実行と比べて，プロセッサ N 台を用いた並列実行では何倍速くなったかを表わす．次式により計算される．

$$\text{台数効果} = \frac{\text{プロセッサ 1 台による実行時間}}{\text{プロセッサ } N \text{台による実行時間}}$$

次にここでいう効率とは，並列処理でプロセッサがどの程度有効に利用されたかを示す値で，次式により求められる．

$$\text{効率 (\%)} = \frac{N\text{プロセッサの台数効果}}{N} \times 100$$

問題が大きいと 1 プロセッサでは実行できず，台数効果を求めることができないが，その場合でも使える効率の定義を 2.4.6節に示す．

b. 計測結果

1 段階の負荷分散と 2 段階の負荷分散で，それぞれ負荷分散レベル (負荷分散を行なう探索の深さ) を変えて計測を繰り返した．表 2.3.1に，1 段階負荷分散で最も高い性能を示した負荷分散レベル L3(探索深さ 3 での負荷分散) の結果を示す．表 2.3.2には，2 段階負荷分散の結果を示す．L2-L4 とは，第 1 負荷分散レベルが L2，第 2 負荷分散レベルが L4 のことである．台数効果のグラフを図 2.4.1に示す．

なお，2 段階の負荷分散はプロセッサグループ (PG) の大きさをプロセッサ 4 台 (4PE) として測定したものである．したがって,64PE のときは 16PG から構成され，32PE では 8PG，16PE では 4PG，8PE では 2PG の構成となった．また 1PE の場合は，すべてのタスクを同一 PG の同一 PE に割りつけた場合として計測した．

図 2.4.1の L3 のグラフを見よう．1 段階の負荷分散では，最も性能のよかった L3 の場合でも，プロセッサ数が増えるに従って，台数効果のグラフの傾きはしだいに緩くなる．これは，プロセッサ数が増加すると，サブタスクの供給ボトルネックと負荷の不均衡の両方の理由によって，並列処理性能が押えられているためである [2]．一方，2 段階の負荷分散では，いずれも

表 2.3.2 2 段階負荷分散の並列処理性能

台数	1 台	8 台	16 台	32 台	64 台
実行時間 (秒)					
L1-L4	261.2	34.2	17.5	9.2	5.3
L2-L4	264.8	34.4	17.6	9.4	5.3
L2-L5	265.1	37.3	18.8	9.7	5.3
台数効果					
L1-L4	1	7.6	14.9	28.4	49.3
L2-L4	1	7.7	15.0	28.2	50.0
L2-L5	1	7.0	14.1	27.3	50.0
効率 (%)					
L1-L4	100	95.0	93.1	88.8	77.0
L2-L4	100	96.3	93.8	88.1	78.1
L2-L5	100	87.5	88.1	85.3	78.1

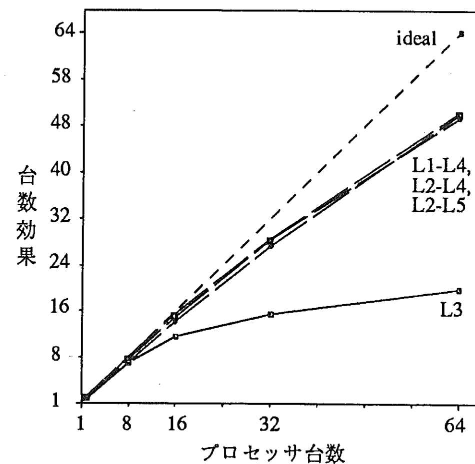


図 2.4.1 詰込みパズルの台数効果

良好な台数効果を示しており，グラフは理想値 (プロセッサ数に等しい台数効果) に近づいている．64 プロセッサ使用時の台数効果は約 50 であり，このときの効率は 80% 近くを達成している．この値は，プロセッサ数の多い分散メモリ型の並列計算機としてはきわめてよい値であり，2 段階の動的負荷分散が良好に動作したことを示している．なお 2 段階負荷分散でも，負荷分散レベルを表 2.3.2に示した選び方から上下にずらしてゆくと，性能は次第に低下する．このことに関する解析結果が報告されている [2]．

2.4　最短経路問題と高並列オブジェクト指向プログラミング

2.4.1　はじめに

ここでは，最短経路問題と呼ばれる最良解探索問題の例を用いて，並列オブジェクトモデルに基づくプログラムのなかでも特に多数のオブジェクトを取り扱う場合のプログラム方法論について解説する．

このプログラム方法は，KL1 によるプログラミングのなかで生まれてきたもので，従来の方法とは異なった特徴的なものである．大規模な並列計算機を効率よく動かすためのプログラム方法として期待されるものの一つであり，次章の応用プログラムのなかでも使われている．

まずはじめに，最短経路問題を例として問題のモデル化と並列アルゴリズムの設計，プログラム方法について解説する．次にこのモデルに適する静的負荷分散方式について説明し，処理効率を高めるための仕事割りつけの方法と計測結果を紹介する．そのなかでは特に，並列処理の効率を議論するのにとりわけ重要な，負荷バランスと通信オーバヘッドの削減，それらのトレードオフについて説明する．また現実的な並列応用プログラムでよくみかける，見込み計算 (speculative computation) について触れる．さらに，並列処理の粒度，問題の大きさと並列処理の効率についても基本的な考え方を示し，測定結果を紹介する．

2.4.2　最短経路問題

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．基本的には，出発点から目的点に至るあらゆる可能な経路のうちから最短の経路を探し出す問題で，最良解探索問題に分類される．最もよい解を一つ見つけ出すという点で，前節の全解探索問題に比べると並列処理の難しい問題といえる．

最短経路問題を簡単に定義しておこう (図 2.4.2)．

n 個の点と e 個の辺からなる有向グラフを考える．一つの辺は，2 個の点と向きで与えられている．すべての辺には，非負の実数値が与えられている．これを辺のコスト，または長さと呼ぶ．$l+1$ 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 v_{i+1} への辺 e_i が存在するとき，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．ある経路に含まれる辺のコストの和をその経路のコストと呼ぶ．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呼ぶ．図 2.4.2 に，グラフと最短経路 (shortest path) の例を示す．このグラフの例では，2 点間の辺のコストは，両方向ともに等しい．

与えられたグラフ上の出発点と呼ぶ特定の 1 点から，ほかのすべての点のおのおの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．これを解くのに，読者の方々はどのようなアルゴリズムを思いつかれるだろうか．

最短経路問題を高速に解くアルゴリズムとしてさまざまな研究がなされてきた．逐次アルゴリズムでは，n 個の点と e 個の辺をも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 [3] がよく知られている．これを改良した逐次アルゴリズムがいくつか報告されている [4]．並列/分散アルゴリズムも研究されている [5]．Chandy と Misra[6] は，分散最短経路アルゴリズムを与えている．

しかし，10^2程度以上のプロセッサ数の大規模汎用 MIMD マシン上で 10^3以上の点をも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，遠隔データへのアクセスはコスト高である．したがって，問題を解くための並列アルゴリズムとして，データをなるべく分散管理することが重要であるとともに，グラフをどのようにプロセッサに割り当てるかが実行効率を左右する．

そこで 2.4.3節ではまず，多数のプロセッサを用いて並列処理可能な並列度の高いアルゴリズムの設計について説明する．その方法は，最短経路問題を並列オブジェクトモデルに基づいて定式化し，その上でデータの集中管理をできる限りなくした分散アルゴリズムを設計するというものである．このモデル化は，逐次計算に慣れ親しんだ読者の方々にはきわめて新鮮に映るに違いない．さらに 2.4.4節で，オブジェクトをどのようにプロセッサに割り当てて実行効率を稼ぐかのマッピング設計について解説する．このなかで，負荷の均等分散と，通信の局所化による通信オーバヘッドの削減について説明する．

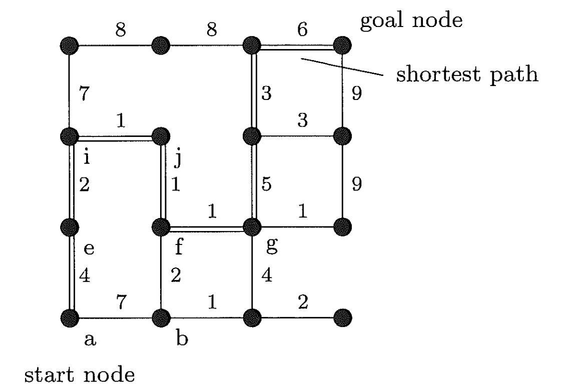


図 2.4.2 グラフと最短経路の例

2.4.3　高並列オブジェクトモデルに基づくプログラム設計

A.　高並列オブジェクトモデル

はじめにモデルを定義しておこう．

多数のオブジェクトが互いに通信し合いながら並行動作しつつ問題の解を求めるモデルである．オブジェクトの個数は多いが種類は少なく，各オブジェクトの動きは非同期的である．計算は，オブジェクト間でのメッセージの流れに応じて不均質に発生する．問題解法のアルゴリズムは，センターを置かない分散アルゴリズムを基本とし，各オブジェクトは受け取ったメッセージに応じて自律的に動作する．オブジェクト間には，メッセージの交換に関して何らかの局所性があるものとする．

上記説明中，粒度とは，通信頻度の逆数と定義する．すなわち通信頻度が高いと粒度は小さい．同一プロセッサ内でのオブジェクト間（プロセス間）通信頻度の逆数，または通信発生の平均周期を PE 内粒度と呼ぶことにする．また PE 間通信の平均周期を PE 間粒度と呼ぶ．

このモデルの特徴のうち，性質のばらつきが小さい多数のオブジェクトの集まりとして問題をモデル化する点は，データ並列処理 (均質なデータを扱う意味で) に近い．一方，計算の発生は非同期的で不均質である．この点はデータ並列処理 (同期的で均質な処理の意味で) と異なる．オブジェクトの種類が少ないものに限定したのは，その数が多いと，プログラムとして記述

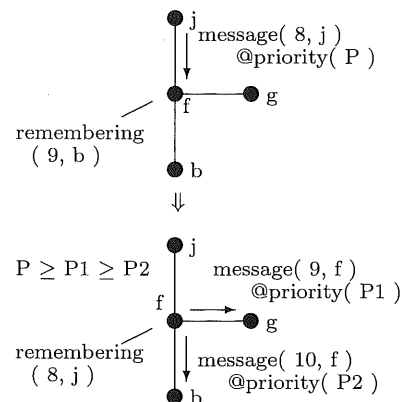


図 2.4.3 f 点におけるメッセージ処理

しきれなくなるからである．

このような形に定式化できる問題が，たくさん存在するか否かはまだ明らかでないが，少なくともデータ並列処理以外の新しい問題領域を対象としている．データ並列よりは自由度の高いモデルであることから，適用範囲は広いと期待している．

B. プログラム作成手順

プログラム作成手順は，次の 4 段階からなる．詳細は文献を参照されたい [7]．

a. 第 1 段階：問題の定式化

前節の並列オブジェクトモデルに基づいて問題を定式化する (し直す)．重要なのはオブジェクトの個数が多いことであり，粒度の小ささは必要条件ではない．オブジェクトの個数を確保することにより，プロセッサ数の多い並列計算機上での負荷バランスを容易にする．

最短経路問題では，最大限の並列性を実現するため，グラフの点 1 個 1 個をオブジェクトとしてモデル化した．辺で結ばれたオブジェクト同士は通信路をもち，最短経路探索のためのメッセージを交換し合う．この問題の場合，モデル化は第 2 段階の分散アルゴリズムの設計と不可分である．

b. 第 2 段階：分散アルゴリズムの設計

上記定式化に基づき，問題解法の分散アルゴリズムを設計する．分散アルゴリズムとは，計算順序の制御をどこか一か所で行なうのではなくて，オブジェクトがメッセージ交換により自律的に動き，オブジェクト間のメッセージの連鎖が解を導く方式のことである．オブジェクトが自律的に並行して動くぶん，並列度が高い．

1 点から全点への最短経路を求める分散アルゴリズムを次のように設計した．

点オブジェクト v_j が受け取るメッセージはコスト経路情報 $cp(\mathrm{cost}, v_i)$ である．cost は，出発点からメッセージが通過してきた経路のコスト，v_i は直前の点の情報である．点オブジェクトはその状態として，これまでに到着したコスト経路情報のうち最小コストのもの $cp(\mathrm{min_cost}, v)$ を保持する．点オブジェクトはメッセージが到着すると，次のように振る舞う．cost ≥ min_cost の場合，何もしない．cost < min_cost の場合，新着メッセージのほうが短い経路を通ってきたことになるので，(1) 自らの状態を新着のコスト経路情報に書き換え，(2) 隣接のオブジェクトに対してコスト経路情報 $cp(\mathrm{cost}+\mathrm{ck}, v_j)$ を生成してメッセージ送出する．ck は隣接オブジェクトへ至る辺のコストである．図 2.4.2のグラフの f 点におけるメッセージ処理の様子を図 2.4.3に示す．

任意のオブジェクト v_j の初期状態は，$cp(+\infty, v_j)$ である．実行は，出発点 v_s にメッセージ $cp(0, v_s)$ を流し込むことにより始まる．出発点の状態は $cp(0, v_s)$ に更新され，隣接オブジェクトへのコスト経路情報が生成されてメッセージ送出される．メッセージはおおむね，出発点から近隣のオブジェクトへと波面状に広がってゆく．オブジェクトが構成するネットワーク上からすべてのメッセージが消滅すると，アルゴリズムは終了する．このとき，各オブジェクトの状態として保持されるコスト経路情報が，出発点からそのオブジェクトへの最短経路情報である [4]．各点オブジェクトは，直前の点の情報だけを保持するため，最短経路を求めるには，その点から逆向きに出発点までたどることが必要である．

このアルゴリズムにおいて，小さいコストのメッセージから先に処理されるようメッセージ処理の優先度を制御することができれば，時間計算量は Dijkstra のアルゴリズム [3] と同じ $O(n^2)$ になる．これを行なわなければ，コストの大きい経路を通ってきたメッセージ (最終的には捨てられてしまうメッセージ) のほうが先に到着して処理されることがしばしば起こり，計算量は増加する．

c. 第 3 段階：プログラム化

オブジェクトを，通信し合う軽いプロセスとしてプログラム化する．この段階で，プログラムの論理的並列性の設計が完了する．KL1 では，この状態での擬似並列実行が可能であり，問題解法の分散アルゴリズムにかかわるバグの除去は，この段階で行なってしまう．

最短経路問題のプログラム化に当たっては，オブジェクトを KL1 のプロセスとしてプログラム化した．オブジェクト間には通信用のストリームを張り，与えられたグラフと同一の接続トポロジを実現する．オブジェクトが扱うメッセージとして，コスト経路情報のほかに，最短経路の問合せ，終結処理などを用意した．すべてのメッセージが消滅したことを検出する方法として，(1) ショートサーキット法 [8]，(2) メッセージの取扱いを独立の荘園内に閉じ込め，荘園の終了検出機能を利用する方法 [9] の 2 種類で試作し，最終的に性能のやや高い後者を使用した．

計算の複雑さを低く保つために，小さいコストのメッセージから先に実行することを厳密に実現しようとすると，すべてのメッセージを優先度つき待ち行列で集中管理する必要が生じる．これは高い並列性を実現する目的と矛盾する．そこで，KL1 ではゴールごとに実行優先度をきめ細かく (現状では 4096 レベル) 指定できることを利用し，メッセージ送出を行なうゴールの実行優先度をメッセージ中のコスト情報に基づき変化させる方式をとった．すなわち，コストの小さいメッセージの送出を担当するゴールほど，高い優先度が与えられる．実行優先度に基づくゴールの実行順制御は，プロセッサ単位で管理実現されているため，複数プロセッサによる実行では，小さいコストのメッセージから実行することは必ずしも厳密には実現されない．これにより，厳密な実行順制御の場合に比べて，本来必要のないメッセージに関する計算が発生し，計算量は幾分増加する．その一方で並列性は確保される．

d. 第 4 段階：仕事の割りつけ

前の段階まででオブジェクトを論理的な並列処理の単位としたプログラム設計は完了したが，次はそれを物理的なプロセッサ上での並列実行に結びつけるマッピングの設計を行なう．これは計算の仕事をプロセッサへ割りつけることである．負荷バランスと通信オーバヘッドの削減を同時に一定水準以上のよさで実現することが目標である．KL1 プログラムでは，プラグマの記述を追加することによって，第 3 段階まででで設計したアルゴリズムやプログラム構造を変更することなく，仕事の割りつけが実現できる．負荷バランスや通信オーバヘッドは設計段階の予想が外れることも多く，そのときは仕事割りつけのやり直しが必要になる．そのようなとき，KL1 プログラムで仕事割りつけの記述が第 3 段階までのプログラム設計と分離して扱えることが，たいそう有利となっている．

仕事割りつけの方法について考えよう．

静的な接続関係をもつオブジェクトがプロセッサの数より十分多く存在する場合を考える．これを複数のプロセッサに割りつけるには，オブジェクトを適当な方法でグループに分けて，グループを単位として割りつければよい．このとき，プロセッサ間通信をなるべく少なくするために，何らかの近傍接続関係をもつオブジェクト同士をグループ化するのが普通である．

まず負荷バランスについて考えよう．オブジェクトはメッセージのやりとりに従って，おのおのが独立に動いており，瞬時瞬時の計算負荷は一様ではない．異なるグループ間でも同様のことがいえる．このため各プロセッサでの負荷をなるべく均等化するには，グループの数をできるだけ多くして，それらをランダムにプロセッサに割りつける．最も極端には，オブジェクト 1 個 1 個を別グループとすればよい．各プロセッサは複数のグループを担当するため，グループ間での計算負荷のばらつきはプロセッサ上で平均化される．これは言葉を換えると，負荷バランスをよくするには，問題をなるべく細かく分割して，複数のプロセッサ上にばらまきなさいといっているに等しい．

ところがこのような割りつけをすると，オブジェクト間での通信がすべてプロセッサ間通信となり，通信オーバヘッドの問題を引き起こしやすい．通信オーバヘッドは，通信の前処理と後処理でプロセッサが働かなければならないことと，通信ネットワーク上での転送時間がかかることから生じる．メッセージ交換 1 回当たりのオブジェクトの計算時間が，上記の通信の前処理後処理時間や，ネットワーク転送時間に比べて十分大きいならば，オーバヘッドは無視し得るので問題ない．そのときは負荷バランスにだけ注意を払えばよい．けれども，オブジェクトの計算時間が短く，通信オーバヘッドが目立つ場合には，プロセッサ間通信を減らすことが必要である．このときは，近傍接続関係をも

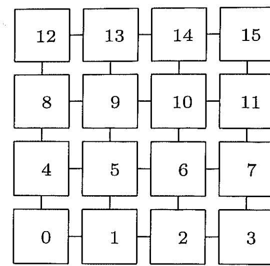
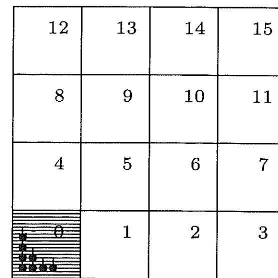


図 2.4.4 16 台構成のマルチ PSI の論理プロセッサ配置



斜線部分はプロセッサ P_0に割り当てられる

図 2.4.5 2 次元単純マッピング方式におけるグラフ分割

つオブジェクトをなるべく多く含むようにグループ化し，プロセッサに割りつける．これは，オブジェクト間の通信をなるべくプロセッサのなかに閉じ込めることにあたる．別の言葉でいうと，問題をなるべく分割しないで，オブジェクト間の通信の局所性を最大限に保って割りつけることである．ただしこうすると，グループの数が減るので，負荷バランスには不利となる．

それでは負荷バランスと通信オーバヘッド削減の両立は難しいのだろうか．負荷バランスのために重要なのは，グループの個数がプロセッサ数に比べて十分多いことであり，通信オーバヘッド削減に重要なのは，グループの外に出ていく通信が少ないことである．扱う問題が大きいほど，いい換えるとオブジェクトの粒度が一定ならオブジェクトの数が多いほど，両者を同時に満たすことが容易となる．両者を満足する割りつけを設計するための方法論は [7] に，また具体例を用いた説明は 2.4.6節に示す．

最短経路問題での具体的な割りつけと，それを行なったときの並列処理性能については，節を改めて解説する．

2.4.4 最短経路問題のマッピングと処理性能

以下では，仕事割りつけをマッピングと呼ぶ．最短経路問題のマッピングとして試みたいくつかの方式と，並列処理性能に関する実験結果を示す．

A. 2 次元単純マッピング方式

まず，筆者らは**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，各ブロックを一つずつプロセッサ配置に従ってプロセッサに割り当てる．たとえば，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 2.4.4のようになっているとき，グラフは 4×4 ブロックに分割される (図 2.4.5)．プロセッサ P_0には斜線部分のブロックが割り当てられる．この方式は，同数の点を各プロセッサに割り当てながら，グラフ内の局所性を最もよく保つものである．

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．隅の点の一つを出発点とした．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．

この場合の実験結果を図 2.4.9，2.4.10 に示す．図 2.4.9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わす．縦軸は実行時間 (秒)，横軸は台数である．図 2.4.10は，図 2.4.9における台数効果を表わしたものである．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，結果はあまりよ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 (図 2.4.6)．各プロセッサは波が来るまで暇な状態であり，波が去ったあと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．

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 (プロセッサ数に対する性能向上率) を計算した [4] ところ，p 台のプロセッサで実行した場合の台

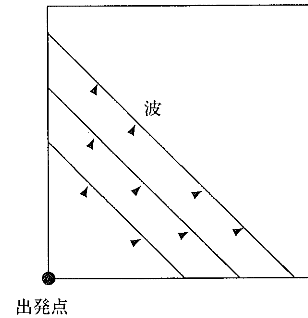
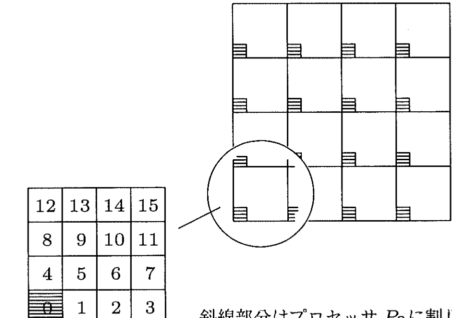


図 2.4.6 コスト経路情報生成の波



斜線部分はプロセッサ P_0に割り当てられる

図 2.4.7 2 次元多重マッピング方式におけるグラフ分割

数効果は$\sqrt{p}$となった．すなわち，プロセッサ稼働率は $1/\sqrt{p}$ということになる．

筆者らは，各マッピング方式について 16 台のプロセッサを用いて前述のグラフ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 2.4.11)．棒グラフは，各プロセッサの稼働率をパーセントで表わしたもの ((稼働時間/総実行時間) ×100) で，斜線部分がそのうちの通信に費やされた時間を表わしている．左から右へプロセッサ 0 から 15 の順である．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．

B. 2 次元多重マッピング方式

2 次元単純マッピング方式ではよ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一部分のみを割り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いくつかの部分を受けも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一部分ずつ，合計 k個のブロックを割り当てられることになる．図 2.4.7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か所の点集合を通りすぎる際に稼働する．したがって，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 $k = 4 \times 4$ のときの台数効果は，プロセッサ台数を 4，16，64 台と変化させた場合のそれぞれについて 2.56，7.25，18.26 であった．同様に $k = 8 \times 8$ のときの台数効果は，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よいプロセッサ稼働率が得られたことがわかる (図 2.4.10)．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 $k = 4 \times 4$ のとき 80%，$k = 8 \times 8$ のとき 94.4% となっている (図 2.4.11)．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，あと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．

C. 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り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，それらを一つずつプロセッサに割り当てる．図 2.4.8は $p = 16$ の場合である．

実験結果 (図 2.4.10) 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

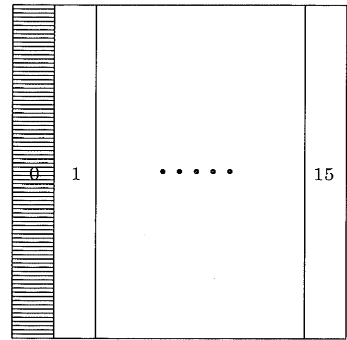


図 2.4.8 1次元単純マッピング方式におけるグラフ分割

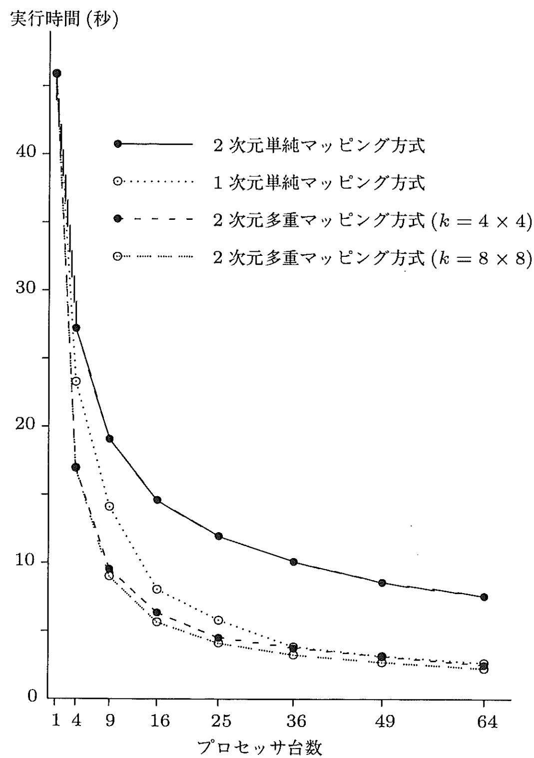


図 2.4.9 マッピング方式，プロセッサ台数と実行時間

程度の性能が得られたことがわかる．結果の解析については省略する [4].

2.4.5 無駄な見込み計算について

無駄な見込み計算 (speculative computation) について触れておこう．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，筆者ら

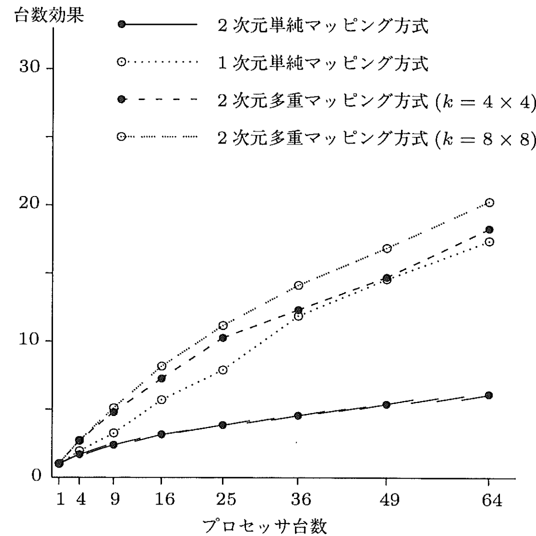


図 2.4.10 マッピング方式，プロセッサ台数と台数効果

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なわれる．

筆者らのアルゴリズムにおいて，小さいコストのメッセージから先に処理されるようメッセージ処理の順序を完全に制御できれば，時間計算量は Dijkstra のアルゴリズム [3] と同じ $O(n^2)$ になる．けれども，これを厳密に実現しようとすると，すべてのメッセージを優先度つき待ち行列で集中管理する必要が生じる．これは高い並列性を実現する目的と矛盾する．そこで，メッセージの処理順序の管理を少しルーズにすることにより，並列実行の可能性を高める方法をとっている．具体的には，KL1 のゴールに実行優先度を指定できる機能を利用し，コストの小さいメッセージの送出に高い優先度が与えられるようにした．実行優先度制御は，プロセッサごとに独立に管理実現されているため，複数プロセッサによる実行では全体を合わせてみると，小さいコストのメッセージから実行することは必ずしも厳密には実現されない．これにより，厳密な実行順制御の場合に比べて，あとで不必要とわかるメッセージに関する計算が発生し，計算量が増加するのである．このように，計算の時点では最終的に役に立つか立たないかわからない計算であってもやってしまうことを見込み計算と呼んでいる．

図 2.4.11から読み取れるプロセッサ稼働率のうち，通信以外の部分が有効な仕事をした部分に当たり，本来ならこの値が図 2.4.10の台数効果に反映されてほしいところである．けれども注意して見ると，台数効果のほうが低い値となっていることがわかる．台数効果を押し下げている原因が，無駄な見込み計算と考えられる．

無駄な見込み計算はなくしてしまえばよいのかというと，この例題の場合，高い並列度を実現するための代償ということができる．ただし，見込み計算による計算量の増加で，時間計算量のオーダが悪化するようでは，並列処理の効果は相殺されかねない．その点に注意した上で，見込み計算を許したアルゴリズムを工夫して，並列処理の効率を稼ぐことが必要である．

2.4.6 問題の大きさと並列処理効率

トレードオフの関係にある負荷バランスと通信オーバヘッドの削減を両立させて，高い並列処理効率を実現しようとすると，大きな問題を対象とするほうが容易に行なえることは先に述べたとおりである．このことを，図を用いてもう少し詳しく見た上で，最短経路問題の実験結果と比べてみよう．

A. 問題サイズと粒度と処理効率

まず，問題の大きさを変えた場合の粒度 (ここでは PE 間粒度のこと) と効率の関係を定性的に説明する．

図 2.4.12は，問題の大きさを大，中，小と 3 種類に固定したそれぞれの場合の，並列処理効率と PE 間粒度の関係を示す．縦軸は並列処理効率を示し，横軸は PE 間粒度を表わす．効率の定義は次式のとおりであり，真の計算時間とは通信処理を除いた処理に要した時間を意味する．

$$\text{並列処理効率} = \frac{\sum_{\text{全 PE}} (\text{PE における真の計算時間})}{\text{実行時間} \times \text{PE 数}}$$

a. 問題が十分大きい場合

図 2.4.12(a) は，問題が十分大きい場合である．問題の大きさとは総計算量である．実線が，粒度を変化させたときの効率のグラフである．1 の領域は，PE 間粒度を小さくしすぎたために CPU が通信処理を行なう時間が相対的に増加し (通信オーバヘッドが顕著になり)，効率を低下させる部分である．2 の領域は，粒度を大きくしすぎたために粒の個数が不十分となり，負荷の不均衡で CPU の遊ぶ時間が発生して効率を低下させる部分である．問題が十分に大きいと，1 と 2 の領域は離れており，中間部分には粒度を変化させても高い効率を維持できる領域が広く存在する．すなわち問題が大きいと，仕事の割りつけに苦労しなくても性能を得やすい．

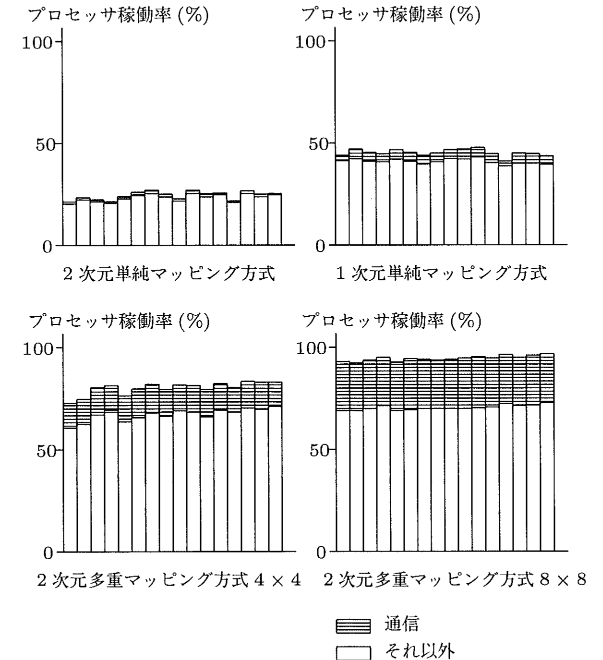


図 2.4.11 マッピング方式とプロセッサ稼働率 (プロセッサ 16 台の場合)

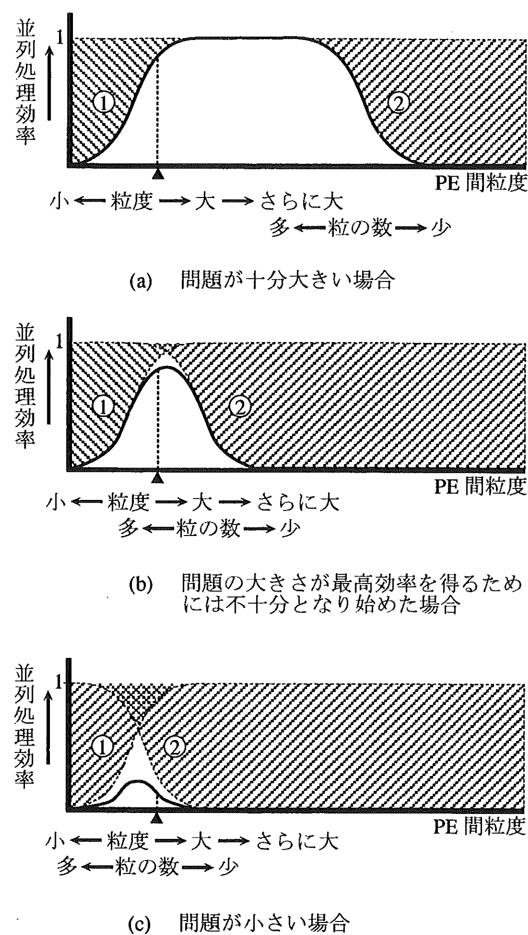
b. 問題の大きさがやや不十分な場合

問題の大きさがやや不十分な場合を 図 2.4.12(b) に示す．問題が小さくなると，粒度を固定した場合の粒の個数は少なくなり，負荷バランスは取りにくくなる．すなわち粒度を増加させた場合の負荷の不均衡は，同図 (a) の場合より早く起こり，グラフの 2 の領域は左に移動する．同図 (b) では，1 と 2 の領域がちょうど重なり始めるような問題の大きさを示す．これより問題が小さくなると，もはや 1 に近い効率を得ることはできない．

文献 [7] の仕事割りつけ方式は，図中の三角印，すなわち通信オーバヘッドが出始める付近に，粒度をもってこようとするものであり，粒度設計の一方法論を与えている．

c. 問題が小さい場合

図 2.4.12(c) は，さらに問題が小さい場合で，1 と 2 の領域が大きく重なっている．重なった部分では，通信オーバヘッドと負荷の不均衡の両方が原因で効率



(a)　問題が十分大きい場合

(b)　問題の大きさが最高効率を得るためには不十分となり始めた場合

(c)　問題が小さい場合

図 2.4.12 問題の大きさを変えたときの粒度と効率の関係

が低下し，総合的な効率は実線のように低いものとなる．この場合に限っては，[7] の仕事割りつけ方式に従って三角印のところに粒度を設定しても，高効率を得ることにはならない．

しかしながら，このように問題の小さい場合は，効率最高の点を見つけるには，粒度と仕事割りつけを慎重に調整する必要があり，それが必要なときは三角印の点を調整の出発点として用いるのがよかろう．

このように効率最良の点を試行錯誤ででも求めたい場合というのは，大きさがほぼ等しく性質も同様の問題を繰り返し計算する必要のあるときで，かつ，プロセッサ台数の多い並列計算機を独占することができ，効率が低くても1個1個の解を少しでも早く求めたいようなときに限られる．

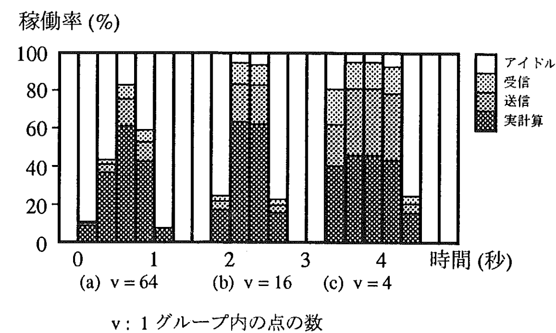


図 2.4.13 最短経路問題 (16,384 点) におけるプロセッサ稼働率

1 台の並列計算機を複数人で使う場合，このように小さい問題では使用プロセッサ数を減らすべきで，そうすると負荷バランスは改善でき並列処理効率も上がり，システム全体の有効利用を図ることができる．

B. 最短経路問題における測定

点の数 16,384 からなる 2 次元格子グラフ (1 辺 128 点からなる正方形) を測定に用いた．各辺のコストは前と同様，乱数を用いて 0 から 99 の範囲の整数を生成して与えた．プログラムは 64 プロセッサのマルチ PSI 上で実行した．

a. プロセッサ稼働率と通信オーバヘッド

仕事割りつけ単位のグループ 1 個に含まれる点の数を 3 種類選び，64 プロセッサで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 2.4.13 に測定結果のグラフを示す．縦軸は稼働率を，横軸は経過時間を示す．棒グラフ 1 本 1 本は，0.25 秒ごとの全プロセッサの平均稼働率および平均の通信オーバヘッドを示す．以前のグラフと横軸の意味が異なるので注意されたい．通信は，送信と受信に分けて表示されている．(a) はグループ内の点の数が 64 個，(b) は 16 個，(c) は 4 個の場合である．計測と表示は，マルチ PSI の OS である PIMOS に組み込まれたツールを用いた [10]．

各グラフとも，中央で稼働率が高く両端で低いのは，実行の開始，終了のタイミングが 0.25 秒刻みの途中に来ていること，実行の開始直後および終了前はメッセージ数が少なく計算量が少ないことによる．

グループ内の点の数を 16 個，4 個と少なくするに従い，問題サイズが一定であるためにグループの個数は次第に増加し，負荷バランスは次第によくなる．これは平均のアイドル時間の減少という形でグラフに現われている．一方，通信オーバヘッドは次第に増加することがグラフから読み取れる．プロセッサが問題を解くためにどれだけ働いたかを示す有効稼働率は，実計算の部分の%値を平均して得られることから，グループ内の点の数 16 個のときが最も効率よく働いたことがわかる．

このような特性を示すのは，前節の説明に従えば，問題が小さい場合に相当する．すなわち PE 間粒度を注意深く調整しないと効率最高の点を見いだすことは難しく，その点においても得られる効率は十分ではない．

b. 問題を大きくした場合

前節で十分な効率が得られないのは，問題が小さいためであった．ここではグラフの点の数を 16 倍の 262,144 にして，プロセッサ稼働率と通信オーバヘッドを計測した．グループ内の点の数は 64，プロセッサ数も 64 である．

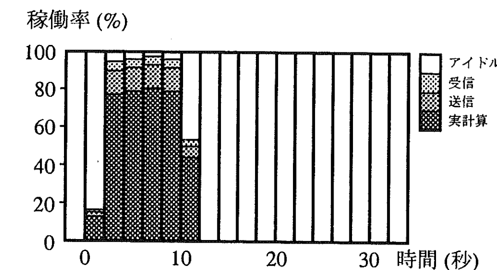
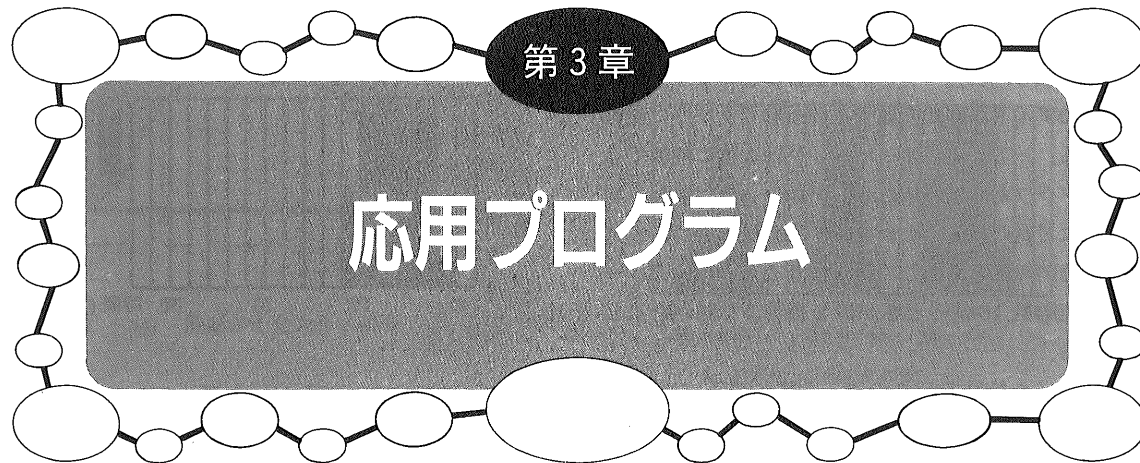


図 2.4.14 規模の大きい最短経路問題 (262,144 点) におけるプロセッサ稼働率

結果を図 2.4.14に示す．通信オーバヘッドは，図 2.4.13(a) の場合とほぼ同じで，idle の割合が十分減少し，グラフの中央部分で 75%を越える有効稼働率 (真の計算時間の割合) を実現することができた．これ以上の大きさの問題では，PE 間粒度を同一にしておくといつも同程度の高い有効稼働率が実現できる．



3.1 並列応用プログラムについて

3.1.1 はじめに

3章では，KL1で記述し並列推論マシンPIMの上で走らせそして評価した，多様な並列応用プログラムについて紹介する．これらのプログラムは，1989年から1992年の間に，2年から4年の期間を費やして研究・開発されたものである．一つの計算機システムの上に，これだけ多様でしかも本格的な並列応用プログラムが短期間のうちに作られた例は，ほかには存在しないものと考えられる．さらにそれらの多くが，従来の数値計算の並列処理のような均質な同期的な計算とは異なって，より取扱いの難しい動的で不均質な計算を含む問題だということも特筆すべきことであろう．

3章の読み方としては，本格的な並列応用プログラム開発とはいかなるものかという具体例として見ていただくのもよいし，並列応用プログラムとはこのように多様なものだという点をとらえていただいてもよいし，256台もプロセッサを使ってこんなに台数効果が出るのかという点に感心していただくのもよいし，あるいは，KL1を使うとこんな並列プログラムが短期間に書けるのかと驚いていただくのもよいと思う．そのなかでも筆者らが最も喜ぶのは，「どうも並列プログラミングはおもしろいらしい．自分も挑戦してみよう」と思っていただくことである．

3.1節では まずはじめに，おのおのの並列応用プログラムの概要とプログラム構造，および実行時の特性を簡単にまとめ，以下の各節の見出し代わりに使えるようにしておく．次に，約4年間の並列応用プログラム開発をとおしての編者の開発経験と感想をまとめる．さらに，より大規模な並列計算機システムへ移行してゆくための展望を述べたい．

3.1.2 応用プログラムおのおのについて

A. LSI 配線

LSIの設計を支援または自動化するCAD(Computer Aided Design)システムは，人の知的作業を助けるコンピュータシステムの代表例の一つであって，第五世代コンピュータ技術を適用するための格好の標的であった．このためプロジェクトの初期から研究開発が試みられたが，プロジェクト中期までは主に，知識処理でCADシステムをインテリジェントにする方向で取り組まれていた．

並列処理を用いるLSI CADへの本格的取組みは，並列実行環境としてマルチPSI/V2が使えるようになったプロジェクト後期からスタートした．大規模なLSIの設計を短時間で完了させるために，並列処理でLSI CADの高速処理を実現する試みとして，ここに述べる配線処理の並列化と，次に述べる論理シミュレーションの並列化を行なった．これらは，第五世代コンピュータの並列処理技術の特徴を生かしたユニークな成果を生み出した．

LSI配線では，大規模な並列計算機での実行可能性を高めるために，配線問題のモデル化をまったく新しい方法で行なった．すなわち前章で述べた高並列オブジェクトモデルに基づくモデル化とプログラミングをはじめて配線問題に適用した．その方法は，配線セグメントの断片一つひとつをすべてオブジェクト(KL1

のプロセス)とし，それらがネットワーク状につながり，互いに通信し合いながらよい配線経路を決めていくものである．前章の最短経路問題と同様のプログラム方法を現実の設計問題に適用したわけである．最短経路問題との性質の違いは，配線経路探索が進むにつれてオブジェクトの個数や接続関係が実行時に動的に変化することで，プロセス構造の分類でいうと部分動的ネットワークに当たる．KL1言語と並列推論マシンPIM上の言語処理系のおかげで，数万以上の粒度の小さいオブジェクトからなる配線プログラムが効率よく並列処理できた．

B. 論理シミュレーション

論理シミュレーションも，LSI CADのなかでは処理を高速化したい問題の一つである．ここでは，これまで並列論理シミュレーションには適用されたことのないタイムワープという分散時刻管理メカニズムを実現し，きわめて高い並列処理効率を得た．実現方式が複雑になりがちなタイムワープを短期間で実装できたのは，KL1言語の記述力の高さに負うところが大きいと考えられる．

プログラム構造とその性質について説明すると，実行モデル上はシミュレーション対象のゲートをオブジェクトに対応させているため，比較的均質な多数のオブジェクトがネットワークを構成し，それらがメッセージ交換して計算が進むのと等価な働きをする．すなわち並列オブジェクトモデルに基づいている．オブジェクトの個数と接続トポロジは静的であり，実行前に決定されてその後は変化しない．一方，オブジェクト間でのメッセージ交換は非同期的で，メッセージは動的に生成消滅する．

データが多数かつ比較的均質で，個数などが静的に定まるところはデータ並列モデルと共通するが，メッセージによって起動される計算はきわめて動的，非同期的であるところは異なっている．

実際のプログラムでは処理効率を改善するため，ゲートを直接オブジェクトとして実現するのではなく，オブジェクトの状態をデータとして管理し，オブジェクトの動きを模擬するプロセスを用意して，プロセッサに一つずつ配置した．別プロセッサ上にあるゲートオブジェクトとのメッセージ交換は，これらのプロセス間のメッセージ交換となる．この実現方法は，いわゆるSPMD(Single Program Multiple Data) [†1]に近いものであるが，プロセッサに1個ずつ割りつけられた上述のプロセス間で，大域的な時刻合わせと記憶領域の解放処理を，ときどき非同期的に行なう．これらの処理があるため，数値計算のSPMDプログラムに比べて，通信や同期の記述はより複雑になっていると思われる．

C. 遺伝子情報処理

遺伝子情報処理は，対象とするデータが膨大となるために大規模並列処理および大規模データベース/知識ベースのよい適用対象として注目されている．FGCSプロジェクトでは後期に入って本格的研究が始められた．並列処理の応用としては，遺伝子のタンパク質配列解析や構造予測プログラムの研究開発を行なったが，本書ではそのなかから2種類のタンパク質配列解析プログラムについて紹介する．

一つは3次元DP(ダイナミックプログラミング法による)マッチングによって，タンパク質中のアミノ酸配列の類似性を見つけ出すものである．3次元DPマッチングでは3個のタンパク質について，それぞれのアミノ酸配列から似ている部分構造を取り出す処理を一括して行なう．3個のタンパク質を一括して扱えることで，従来の2次元DPよりメリットが大きいが計算量も大きいため，並列処理で高速化することの恩恵が大きい応用である．

この方法は，3次元格子上の最短経路探索問題と等価である．プログラム構造と性質は，格子点のおのおのにKL1プロセスを配置し，格子の辺に対応したプロセス間通信路を配置している．メッセージ駆動で経路探索する点はLSI配線と共通するが，メッセージの流れる向きが一定していることを利用して，複数のDPマッチングについてパイプライン処理できるよう工夫している．各プロセスでは，すぐ上流に当たる複数のプロセスからのメッセージ到着をすべて待ったうえで計算を行なっている[†2]．複数メッセージを待つことからオブジェクト指向とはやや異なるが，粒度の小

†1 SPMDは，データ並列モデルに基づくプログラムをMIMDマシン上に実現する場合のプログラム構造であるとともに，さらに広く，一様なデータを分割して複数プロセッサに割りつけ，並列処理する場合のプログラム構造でもある．

†2 メッセージの流れは本来は動的，非同期的でよいが，このプログラムではメッセージ処理のパイプライン化のために同期処理を行なっている．

さい高並列のプロセス指向プログラムといえる．プロセスの個数，接続関係は静的である[†1]．

もう一つは，やはりアミノ酸配列の類似性解析をシミュレーテッドアニーリング(SA)と呼ぶ最適解を確率的に解く方法によって求めるプログラムである．こちらは，並列処理で高速化を目指すのではなく，従来よりも最適化能力の高い新しいSAアルゴリズムを並列処理の性質を生かして実現した点がきわめてユニークな仕事となっている．プログラムの性質はSPMD的であり，KL1が使えない並列処理システムでも実現できそうである．

D.　法的推論

法的推論とは法律分野の専門家が，裁判で論争などをするときに行なう推論のことで，人工知能の応用として古い歴史をもつ．やっていることは，法令文と判令という2種類の知識ベースを用いて，新しい事件に対する法的解釈を与えるものである．知識処理技法の有効性を試す格好の題材であるとともに，大量のデータベース検索や利用に処理時間がかかることから，並列処理による高速化も必要とされる分野である．

知識処理と並列処理というともに難しい二つのテーマを抱き合わせにしてシステムを構築することは，一般にたいそう困難で開発リスクの大きいものであるが，これは並列処理，知識処理，双方の難しさを解決して高い処理効率を実現した数少ない例となっている．

紹介する試作システムは刑法を対象としている．プログラムは2種類のタイプの異なる推論エンジンを組み合わせた構造である．ルールベースの推論エンジンには，定理証明系のMGTPを使用している．プログラムは基本的に生成・検査系であり，生成部分がOR並列で木探索を行なうのと同様に別解を発生し，この部分について動的負荷分散を行なっている．

事例ベースの推論エンジンでは，判例のデータが各プロセッサに分配されており，新しい事件に対するデータの検索と類似度計算を並列に実行する．基本的には，並列にデータベース検索を行なうのにデータを複数プロセッサに分割配置しておくのと同様の考え方である．

†1　データの静的な性質と計算の同期的性質があることから，データ並列モデルに基づいて設計し直すこともできそうなプログラムと考えられる．

E.　定理証明系MGTP

そもそもPrologをはじめとする論理プログラミング言語は，一階述語論理の定理証明過程を起源として生まれてきたものであった．したがって定理証明系を並列プログラム化することは，ここで紹介する数ある応用のなかでも，論理プログラミングとのかかわりが最も深いものといえる．定理証明系は，法的推論でも用いられたように一種の汎用推論エンジンとして利用できるものであり，知識表現と推論という知識処理の中核技術と深く関係している．

ここで紹介するのは，一階述語論理の定理証明系をモデル生成法に基づいて実現したものでMGTPと呼んでいる．1992年時点で世界最高速を誇り，数学の未解決問題をはじめて解いた実績をもつ．

MGTPの動作は基本的には生成・検査(generation and test)系であるが，これをOR並列，AND並列の二つの並列化方式で試作し，それぞれに適した負荷分散方式を検討している．256プロセッサのPIMで200倍を越えるきわめてよい台数効果を記録している．

F.　自然言語解析

自然言語解析はFGCSプロジェクトの初期から取り組まれたテーマであるが，並列化への取組みは，プロジェクト中期の並列構文解析プログラムPAXから始まった．自然言語解析プログラムは，計算機に自然言語を理解させいろいろな仕事をさせる場合の，入力部分を構成するのに汎用的に利用されるものである．ここで紹介するLAPUTAは，PAXの技術を利用しながら，自然言語解析のなかの形態素，構文，意味解析の流れを型理論という枠組みのもとに並列協調プログラムとして一体化したものである．

プログラムの構造としては，重複を許さない全解探索プログラムを効率よく実現するための一つのパラダイムであるレイヤード・ストリーム法[†2]に基づいている．レイヤード・ストリーム法は通信を局所化する効率のよい負荷分散が難しいため，LAPUTAではPAXに比べてそれを改善するために，プロセスとストリームの接続関係を工夫している．

†2　レイヤード・ストリーム法はICOTで松本裕治氏(現在，奈良先端科学技術大学院大学)により考案された．

G.　データベース

データベースは大量の知識を格納するのになくてはならないものであり第五世代コンピュータの基礎技術の一つであるが，知識を格納するデータベースに推論機能をもたせるように拡張したものが知識ベースである．FGCSプロジェクトでは，関係データベースに演繹推論機能の拡張を施した演繹データベースに，さらにオブジェクト指向機能をもたせる方向で知識ベース管理システムとして研究を続けている．ここで紹介するのは，その下位層を支える並列データベース管理システムKappa-Pである．

Kappa-Pは，階層化された知識やデータを効率よく扱うことができる非正規関係データベース管理システムであり，逐次型のKappa-IIの並列版として開発された．大容量のメモリを備える並列推論マシンを活用するために主記憶データベースの機能を備える．

並列化にあたっては，同じ組として管理されているデータを分割して別のプロセッサに配置し，それらに対して同じデータベース演算指令を出す方式をとっている．これを実現するために，プロセッサごとにローカルなデータベース管理システム(DBMS)を配置し，全体的な管理情報はサーバDBMSにもたせている．ローカルDBMSのつくりはSPMD的なプログラムといえる．

3.1.3　全体の傾向について

1988年頃からいくつもの並列プログラムを開発してきたが，初期の実験的プログラムには探索問題が多く[11]，探索木の動的な展開を行なうものが主であった．その後，規模の大きい並列プログラムを作るようになって，しだいに並列オブジェクトモデルないしプロセス指向によるプログラムが増えてきたように思う．探索問題では，大規模並列処理が可能な場合が多いが，その機能は問題解決メカニズムの部品として現われるのが普通で，それ単独で一つの問題解決器となる場合はむしろ少ないであろう．それに対して，問題解決器全体を表現し，複数の部品をつなぎ合わせるのに，並列オブジェクトモデルを用いることは自然と考えられる．

並列オブジェクトモデルに基づくプログラムにも幾通りかのタイプがある．一つは，粒度が大きく粒の数は少なく，粒度およびオブジェクトの属性のばらつきが大きいタイプである．協調問題解決プログラムなどはこのタイプであり，複数の部品を集めてシステムを実現する場合もマクロな構造はこのタイプと見られる．それぞれのオブジェクトは，異なる機能に対応している．ただし，このタイプだけからなるプログラムは，大規模な並列計算には向かない．

もう一つは，粒度が比較的小さく粒の数は非常に多くて，オブジェクトの属性と粒度のばらつきが小さいタイプである．論理シミュレーションやLSI配線がこれであり，大量のオブジェクトは大量のデータに対応している．大規模な並列処理に対応しやすい．このタイプはデータ量の多いことがデータ並列モデルと共通しているが，計算が動的，非同期的であるところが異なる．このタイプはさらに，オブジェクト生成およびその接続関係が静的なものと，動的に生成・消滅するものに分かれる．このタイプは，それ単独で問題解決器を構成することも，より上位のシステムの部品となることもできる．

これらとは別に，データベースあるいは知識ベース的なものの処理には，データないしルールを複数プロセッサに分割して保持し，並列検索するタイプがある．法的推論の事例ベースとICOTのデータベースシステムKappa-Pがそうである．ただし，データ分割と並列検索だけならSPMDとして実現可能なので，むしろ取り出したデータを使うか加工する段階で，動的で不均質な処理が必要なときにKL1の技術が生きてくるように思われる．

これまでの経験に基づく個人的見解としてまとめを行なう．大規模並列処理で効率を上げるには，大量の計算が必要なことは明らかである．対象を動的で均質さの低い大量計算に限定するならば，そのような大量計算を発生する元になるのは，木探索または生成・検査系のような一種のデータ生成系か，多数のオブジェクトの上での動的，非同期的計算(オブジェクトがデータに対応する場合)の二通りくらいが中心ではなかろうか．問題解決器のなかでそれらが有効に使えるためには，前者に関しては，たとえばMGTPのようなかなり汎用的に部品として使えて，計算量の大きいプログラムモジュールないしライブラリの整備が必要であろう．また後者に関しては，並列オブジェクトモデルと対応づけたデータの表現技術と，その上での並列問題解決アルゴリズム(分散アルゴリズムになることが

多かろう) の開発が重要になろう．また両者に対応した，効率のよい負荷分散技術もなくてはならないものである．データベース，知識ベースのようなデータ蓄積，検索系も大量計算を必要とするが，なかでも知識ベースのように複雑な動きを必要とし SPMD として実現しにくいものでは，並列オブジェクトモデルに基づくプログラムが適するように思われる．それを使いやすい汎用モジュールにすることも，もちろん大切である．

3.1.4 超並列処理への展望

超並列処理として，ここではプロセッサ数 千台以上の MIMD 計算機上の処理を想定しよう．大規模な記号処理で，動的で均質さの低い計算を発生する問題の場合に，上記の規模にスケールアップしてゆくには，どのような問題が出てくるだろうか．

Multi-PSI 64 プロセッサから，PIM/m 256 プロセッサにスケールアップしたときのことを考えると，まず Multi-PSI で用いていた問題は PIM/m では小さすぎ，処理時間は短縮されたが台数効果は十分に上がらなかった．そこでより大きな問題をもってきて実行すると，メモリ不足を引き起こすものも出た．一般に，並列処理効率を低下させないことを目指したとき，プロセッサ数を n 倍にしたときに，問題の大きさ (計算量) を何倍にしなければならないかを，等効率解析と呼ぶ [12]．n の定数倍なら非常によく，2 乗やそれ以上になることも少なくない．これが大きすぎると，メモリ不足で実行不可能になったり，現実的な時間で解けなくなったりする．

このような場合，アルゴリズムのオーダを下げることができないならば，プログラムの改良で実行時間やメモリ消費のコンスタントファクタを削減し，実行可能なように改良することが必要となる．

プロセッサ数が 64 から 256 に変わっただけで，実感としてはずいぶん大きな問題が実行可能になったし，また大きな問題を用意しなければ，高い台数効果が得られなくなったようにも思われる．これが 1,000 になると，また一段とその印象は強くなるであろう．いまのプログラム構造やデータの性質から考えて，問題さえ大きくすれば 1,000 プロセッサにでも対応できそうなものは，論理シミュレーション，LSI レイアウト処理，タンパク質配列解析，MGTP である．法的推論は，事例ベース，ルールベースの大量データを用意すること自体がたいへんなようである．

プロセッサ数が 1,000 を越える超並列マシンを対象として，動的で均質さの小さい計算問題の並列処理を多様な応用領域で効率よく実現するためには，先にも述べたように，大量データに関する生成系，蓄積・検索系のプログラムモジュール/ライブラリの充実や，並列オブジェクトモデルに基づくデータの表現手法と並列アルゴリズム，そしてそれらのための効率のよい負荷分散手法の開発・蓄積がなくてはならないものと考える．

3.2 LSI 配線

3.2.1 はじめに

LSI を設計するにはたくさんの工程があり，計算機による設計支援 (CAD) システムが必要不可欠となっている．この LSI CAD システムのなかで最も自動化が進んでいる処理の一つに，「LSI の配線設計」がある．これは，LSI の表面に配置されたトランジスタなどの素子間をどのように結線するかを設計する処理である．

ここでは，この「LSI の配線設計」の並列処理について紹介しよう．

LSI 製造技術の進歩により，一つのチップに搭載できるトランジスタ数は年々増加しており，たとえばマイクロプロセッサの場合，すでに 100 万個を越えている [13]．配線設計の処理時間は，トランジスタ数のほぼ 2 乗に比例するため，全 LSI 設計時間における配線設計の占める割合も高くなっており，高速な LSI 配線プログラムの開発が望まれている．

このような背景のもと，並列処理を用いた LSI 配線設計の高速化の研究として，さまざまなアプローチがなされてきた．それらを分類すると，次の二つに大別される．一つは，ハードウェア エンジンと呼ばれる特定のアルゴリズムを実行するハードウェアを設計・試作したもの，そしてもう一つは，汎用並列計算機上でソフトウェアにより実現したものである．前者は後者に比べ高速であるが，ある機能を付加したり，プログラムの一部を変更しようとしたときに柔軟に対応できないという問題点がある．

筆者らは，そのような仕様変更に対する柔軟性を重視し，MIMD 型汎用大規模並列マシンに適用できる並列配線プログラムの開発を続けてきた．そして，前章で述べた並列オブジェクトモデルに基づいて配線プログラムを設計した [14]．その特徴は，配線格子上のすべての線分をオブジェクトとし，それらの線分オブジェクト間でメッセージを交換することにより配線経路を決定する点にある．

このような配線プログラムを実現するしようとすると，各線分オブジェクトの動的な分裂・統合と，それに伴うメッセージ通信路の張替えなどの複雑な同期処理を記述しなければならない．

KL1 言語処理系では，各オブジェクトの処理は，そのすべての変数が具体化されるまで待ちの状態になる．この自然な同期機構を利用することで，C 言語などの手続き型の言語と比較して，配線プログラムにおける複雑な同期機構の記述が容易にできる．

また，並列配線プログラムの処理速度は，各線分オブジェクトをどのプロセッサで実行させるかによって決まる．そこで，線分オブジェクトのプロセッサへの割りつけ方法を変えて，その処理速度を測定する必要がある．このとき，プロセッサの割りつけ方法を変えるたびに，ソースプログラムを変更していたのでは大変な手間がかかる．

KL1 言語では，各オブジェクトを実行させるプロセッサを示すプラグマを指定するだけで，基本となるソースプログラムを変更しなくてすむのでプロセッサへのオブジェクトの割当て方法と処理速度の関係などを効率よく測定できる．

以下では，並列推論マシン PIM および並列論理型言語 KL1 の応用プログラムの一例として，筆者らの並列配線プログラムの設計方法および実行方法について述べる．そしてプログラムの並列化に伴い生じた問題点とその対処方法を概説し，最後に評価結果と今後の課題についてまとめる．

3.2.2 LSI 設計におけるブロック間 2 層配線問題

LSI の動作仕様を記述した論理回路を，実際にハードウェアとして実現するのがレイアウト処理である．この処理は，配置処理と配線処理の二つからなる．配置処理では，論理回路の機能を実現するための部品

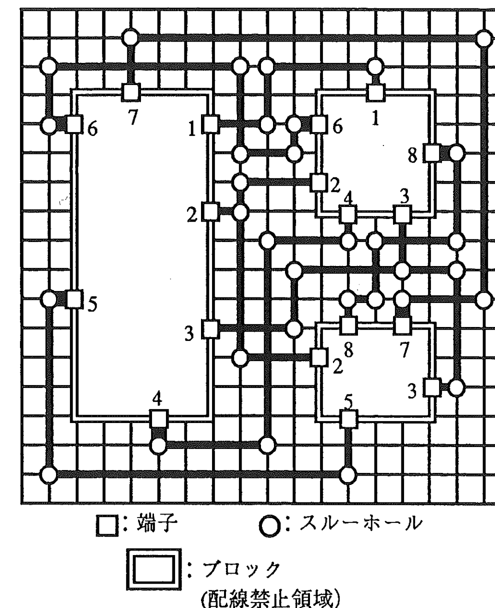


図 3.2.1 2 層配線の例

(以下，ブロックと呼ぶ) が決められた制約を守るように配置される．このとき，配置されたブロックの端子間をなるべく未結線がないように全端子組を結ぶのが配線処理である．ここで，接続すべき端子の組はネットと呼ばれ，全ネットの集合はネットリストと呼ばれる．

一般に，LSI の配線処理は，複数の層を用いて行なわれる．ここでは，そのなかで最も基本となる 2 層配線を取り上げる (図 3.2.1参照)．

二つの配線層を用いて LSI の端子間を配線する場合，配線が層間にわたるときは，via ホール (スルーホールとも呼ばれる) という穴を開けて，層間の配線を接続する．このとき，via ホールを無条件に開けると，電気的影響が出て回路の正常な動作を妨げたり，LSI の構造上の強度が低下して破損が起こりやすくなったりするので，via ホールを打ってはいけない場所が指定される．また，一般にブロックの上は配線できないので配線禁止領域となる．ここでは，第 1 層目を縦方向の配線，第 2 層目を横方向の配線に用いるとする．

ブロック間 2 層配線問題とは，上で述べたネットリストと配線禁止領域および via ホール禁止点が与えられたときに，全ネットに対して，与えられた制約を満たす端子間の接続経路を見つける問題である．

それでは，次にこの問題を並列に解くプログラムをどのようにして設計し，実行させるのかを見ていこう．

3.2.3 並列オブジェクトモデルに基づいた配線プログラム

前項で述べたように，並列オブジェクトモデルに基づく問題の定式化は，問題から並列性を引き出そうとするときに有効な方法の一つである．ここでは，筆者らの並列配線プログラムがどのように設計され，いかにして並列実行されるのかを，並列オブジェクトモデルに基づく並列プログラム設計法に沿って見てみよう．

A. 並列プログラミング手法

並列オブジェクトモデルに基づいた並列プログラムの設計は，並列アルゴリズムの設計と並列実行方法の設計 (プロセッサ マッピング設計) の二つからなる．

最初に並列アルゴリズムの設計では，1) 何をオブジェクトとするか，2) オブジェクトをプロセスとして実現したときにプロセス間でどのようなメッセージ交換をさせるか，そして各プロセスにどのような応答をさせるか，の二つを決めなければならない．

また，並列実行方法の設計では，どのような規則でグルーピングするか，そしてそれらをいかにしてプロセッサに割り当てるのかを決めなければならない．これをプロセッサマッピングと呼ぶ．KL1 言語を用いるとこれらの二つの設計を独立にできるため，効率よく並列プログラムを設計できる．

それでは，筆者らの配線プログラムにおける各事項が，どのように設計されたのかを見ていこう．

B. 並列配線アルゴリズム

従来提案されている配線アルゴリズムの代表的なものとしてチャネル配線法，線分探索法，迷路法などがある．

このなかからブロック間 2 層配線問題を解くための基本アルゴリズムを選ぶ場合，並列オブジェクトモデルと親和性がよいのは線分探索法である．

なぜなら，ブロック間 2 層配線問題における配線領域は不規則な形状をしているため，チャネルと呼ばれる矩形を配線領域として仮定するチャネル配線法では対処できない．また，迷路法は不規則な配線領域を扱えるが，処理単位が格子点なので，計算の効率が悪い．

一方，線分探索法は線分を処理単位とするので，各線分をオブジェクトに対応させると，自然に並列オブジェクトモデルとして定式化でき，高い並列性を内在させ得る．よって筆者らは，1) 各配線線分をオブジェクトとし，2) 各線分オブジェクトを**ラインプロセス**として実現し，処理途中の配線状況を各ラインプロセスの内部状態として表現した．そして，線分探索法に配線率向上のための改良を加えた予測線分探索法 [15] を基本アルゴリズムとして分散アルゴリズムを設計した．

それでは，予測線分探索法とはどのようなアルゴリズムなのか，またこのアルゴリズムを並列化したときにラインプロセス間でどのようなメッセージ通信を行なうのか，そしてこのようなアルゴリズムを実現するためには，ラインプロセス間のメッセージ通信路はどのように決めればよいのか，について順に見ていこう．

(1) 予測線分探索法

基本アルゴリズムとして採用した予測線分探索法 [15] とは，線分探索法に**先読み**を加えた逐次配線手法であり，二重探索を避けるフラグとバックトラックなどを導入し，経路が存在すれば必ず発見できることを保証している．

それでは，ここでその流れを説明しよう．

いま，始点から目標点に向かって経路探索を行なう場合を考える．

最初に，始点から折れ曲がり 1 回で到達可能な点 (期待位置) を探索し，その点から目標点までの距離をそれぞれ計算する．ここで，via ホール禁止点では折れ曲がれないので，via ホール禁止点に対する探索は行なわないこととする．

そして，それらの期待位置のうちで目標点に一番近い点を選ぶ．以上の処理を**先読み**と呼ぶ．

次に，選ばれた点に到達できる折れ曲がり点と始点とを結線する．この一連の処理を繰り返して，目標点までの配線経路を決定する．

このように予測線分探索法は，1 回の先読みで得られた評価値 (目標点までの距離) の，最もよい方角へ進路をとってゆく，一種の山登り法ということができる．しかし袋小路に入ると探索が終了してしまう．すなわち局所最適点で停止してしまうので，そこから脱出するために，探索領域を制限する方法として IEP フラグを導入している．これにより，最短経路に比べると遠回りになることが多いが，局所最適点から脱出

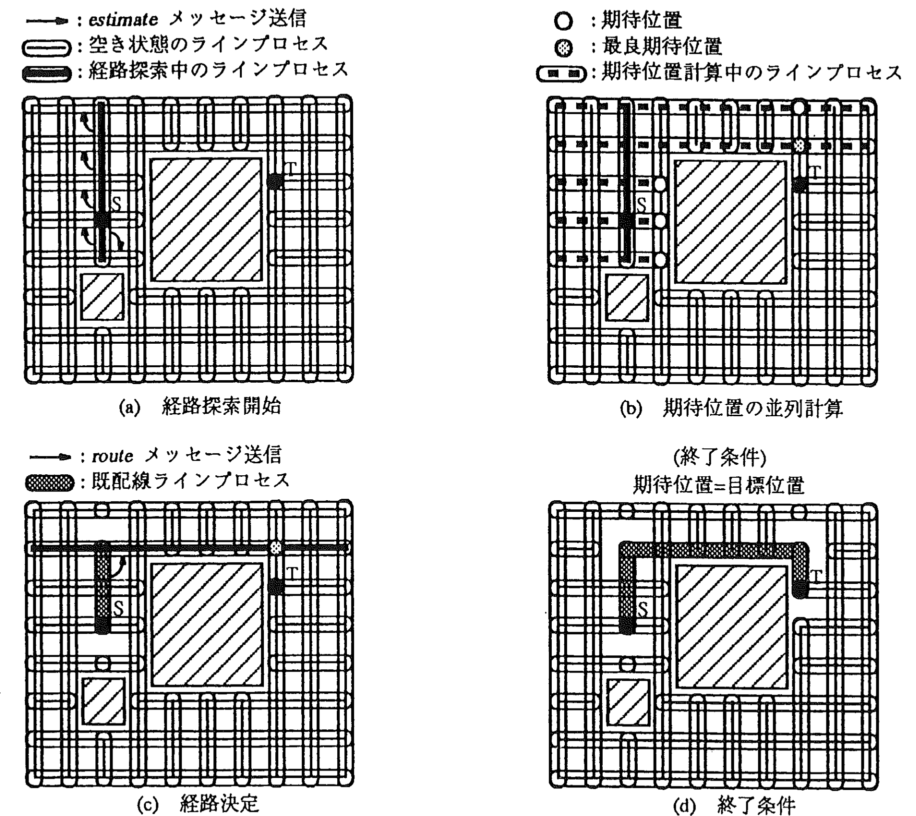


図 3.2.2 先読みの並列実行例

できる．さらに，IEP フラグを用いても期待位置が求まらないことがあるが，その場合は真の行止りなので，バックトラック処理を行ない，一つ前の探索線に戻って別の経路を探す．それでもよい経路が見つからなかったら，バックトラックを繰り返す．そして最終的にスタート点まで戻ってしまったら，経路が存在しなかったことを意味する．このようにして，経路が存在すれば必ず発見できることを保証している．

(2) 先読みの並列実行例

筆者らのプログラムでは，二通りの並列性を実現している．第一は，一対の端子間を配線するアルゴリズムのなかで，予測線分探索法の先読み部分を並列化している．第二は，複数のネットを同時並行に配線することによる並列性である．

それでは，1 ネット探索での先読み処理を並列に行なう場合に，ラインプロセス間でどのようなメッセージが通信されるのかを見てみよう．

いま，図 3.2.2(a) において，S を始点，T を目標点とする．

S から垂直方向に期待位置を求める場合，図 3.2.2(b) に示すように S を含む未配線領域のラインプロセスは，自分と直交する同じく未配線領域のラインプロセスに対して，各領域中で目標点に最も近い位置，すなわち期待位置の計算を依頼するために ‘estimate’ メッセージを送信する．そして依頼元のラインプロセスは，すべての計算結果が返されるまで待ちに入る．

‘estimate’ メッセージを受け取ったラインプロセスは，各自の期待位置の計算結果を依頼元のラインプロセスに返す．依頼元のラインプロセスは回答がすべて集まった時点で，そのなかから最も目標点に近い期待位置を回答してきたラインプロセスを選び，自分との交点と S との間を新たな既配線領域とする．すなわち図 3.2.2(c) に示すように，探索中のラインプロセスが，新たな既配線領域のラインプロセスと残りの未配線領域のラインプロセスとに分裂する．

次に，目標点に最も近い期待位置を回答してきたラ

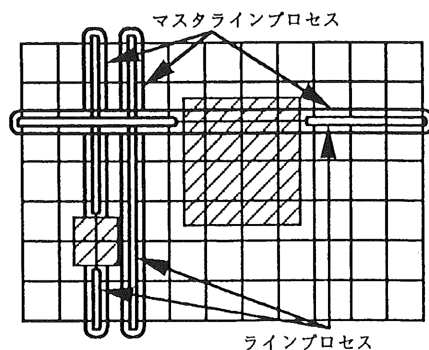


図 3.2.3 マスタラインプロセスとラインプロセス

インプロセスに 'route' メッセージを送り制御を移す．そして目標点＝期待位置となるラインプロセスまで制御が移れば，図 3.2.2(d) に示すように 1 ネットの配線が終了する．

以上の処理において，目標点 T に到達する前に経路探索が終了してしまう場合は，逐次版の予測線分探索法と同様に IEP フラグを用いて迂回路を探索する．それでも期待位置が求まらない場合は，バックトラック処理を行ない，一つ前の探索線に戻って経路探索を続ける．バックトラック処理の結果，始点 S に戻ってしまったら経路が存在しなかったとする．

(3) 基本プロセス構造

最初に，各ラインプロセスは，どのラインプロセスと通信しなければならないかを考えてみよう．まず，並列先読み処理のために，各ラインプロセスは，それに直行する全ラインプロセスとメッセージ通信路を確保しなければならない．また一度，配線経路となったラインプロセスが，バックトラックによって再び元の状態に戻るときに，隣のラインプロセスを統合しなければならない．

よって，各ラインプロセスは，それと直行するラインプロセスおよび隣接するラインプロセスとの間で交信する必要がある．

もし，各ラインプロセスが，お互いにメッセージ通信路を直接確保すると，そのラインプロセスが，分裂，統合する場合にメッセージ通信路の張替えを行なわなければならない．これは，ベクタのコピーなどを伴う重い処理である．

このようなプロセス間のメッセージ通信路の張替え処理を軽減させるため，図 3.2.3に示すように，ライン プロセスのほかに，**マスタラインプロセス**を加えた．各マスタラインプロセスは，配線格子の縦線または横線各 1 本に対応しており，対応する格子線上のラインプロセス群を管理するとともに，直交するラインプロセスから自らの管理下のラインプロセスへ送られるメッセージを仲介する．

このようにマスタラインプロセスを経由してラインプロセス間が通信することにより，ラインプロセス間で直接ストリームを張り替えるのを解消している．

C. 並列実行方法の設計

KL1 プログラムを PIM 上で効率よく実行させるためには，並列アルゴリズムに基づいて設計された各プロセスをプロセッサに分配する際に，1) 負荷の均等化と 2) 通信の局所化を考慮して行なわなければならない．

KL1 言語では，プロセスを実行するプロセッサを指定するためのプラグマが用意されており，基本的なソースプログラムはそのままで，このプラグマを変更することでプロセッサの割りつけ方法を変えることができる．

筆者らのプログラムでは，まず通信の局所化として，マスタラインプロセスとその上のラインプロセスを一つのグループとした．そして負荷の均等化に関しては，各グループをプロセッサにランダムに割り当てることにより対処している．

筆者らの並列配線アルゴリズムは，基本的に配線・探索に必要なメッセージ通信量が多く，上記の並列実行方法だとプロセッサ間通信の量に関する抑制という点では不十分である．今後，プロセッサ間通信をなるべく行なわないようなプロセッサマッピング方式について検討していく必要がある．

3.2.4 並列化に伴う問題点と対処法

筆者らの方式で並列に予測線分探索法を行なおうとすると三つの問題点が生じた．第一に，デッドロックの問題，第二に，プロセッサ間通信路を表現するためのメモリ不足，そして第三の問題点は，複数ネット間の競合の問題である．

A. デッドロック

[問題]

1 ネットについての探索を，複数のネットについて同時に進めることは，まったく独立に実行可能なわけではなく，ある場合にはデッドロックを起こし得る．

たとえば，図 3.2.4のように直交する線分のラインプロセスがほぼ同時期に探索を行なった場合，先読み処理の並列化の説明にもあるように，双方のラインプロセスは，直交する線分のラインプロセスからの期待位置計算結果がすべて返ってくるのを待って，その結果を集計する．

しかし，プロセスは基本的には一つのメッセージの処理が完了するまでは，それ以降に届いたメッセージに対する処理を行なうことができない．したがって，現在処理中の探索処理の実行が終わるまでは，直交するラインプロセスからの期待位置計算要求メッセージに対する処理を行なうことはできない．このようにして直交するラインプロセスが，互いに相手からの期待位置計算結果を待ち合って，デッドロックに陥る場合がある．

この問題点を解決するには，どうすればよいだろう．筆者らのプログラムでは，以下のようにして対処している．

[対処方法]

まず最初にメッセージを二つのグループ A，B に分ける．A は，その処理中に他のオブジェクトに新たなメッセージ送出を行ない，その応答を待たないと終了できない種類のメッセージとする．また B は，メッセージ処理が始まると，無条件に一定時間内にその処理を終了し，応答を返す種類のメッセージとする．そして A の処理に入ると，並行してオブジェクトの入口にあとから到着してくるメッセージをすべて監視し，そのなかに B のタイプのメッセージを発見すると，A の処理中でも B に対する処理を行ない応答を返すことにする．こうすれば，期待位置要求を出したプロセスは必ず返事がもらえ，デッドロックしない．

これを実現するために，探索処理を行なっている間に届くメッセージを監視し，直交するラインプロセスからの期待位置計算要求メッセージが届いたならば，それに対して探索処理の開始直前の状態に基づいた仮の値を計算して返すという処理を行なっている．また，監視中に届いた新たな探索要求メッセージはバッファにためておく．

B. メッセージ通信路のためのメモリの不足

[問題]

筆者らの並列配線プログラムの場合，ラインプロセス間のメッセージの仲介を行なっているマスタラインプロセスは，直交するすべてのマスタラインプロセスと通信をしなければならない．よって配線格子規模の増加とともにプロセッサ間にわたる通信路も増えるので，現状のシステムでは，配線格子数が 750 × 750 程度で通信路を表現するメモリがオーバフローしてしまう．

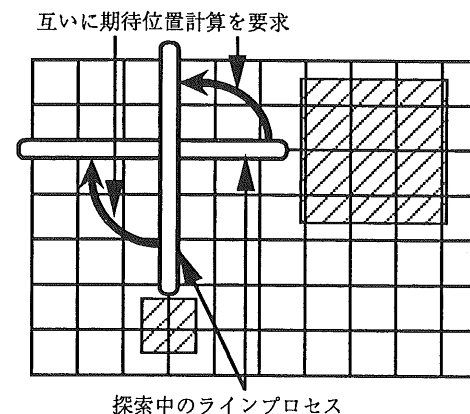


図 3.2.4 デッドロックの発生例

この問題点に対処するには，プロセス構造の改良が必要である．

[対処方法]

前節の問題点を解決するためには，マスタラインプロセスとラインプロセスのプロセッサへの割当て方法はそのままで，各プロセッサに図 3.2.5に示すような**ディストリビュータプロセス**をおき，プロセッサ間にわたるストリームの数を削減する．

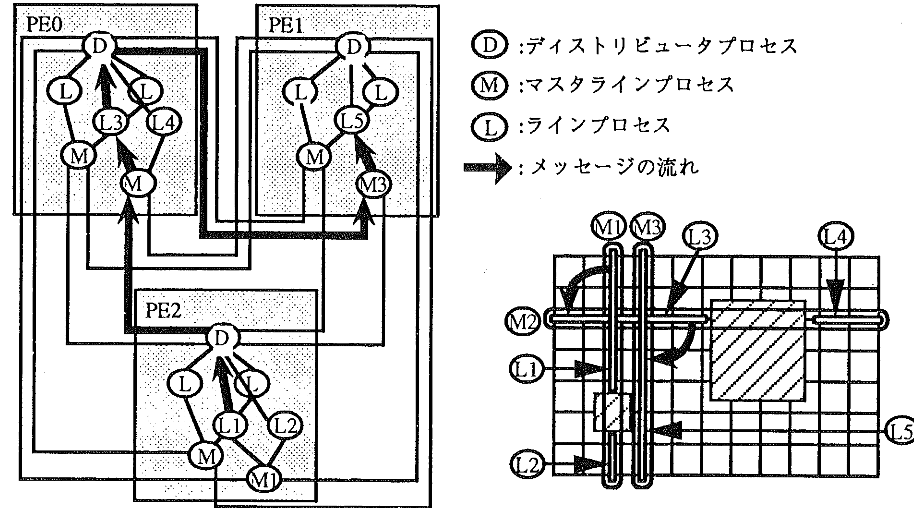
各ディストリビュータプロセスは，プロセッサ内のラインプロセスと通信路をもっており，他プロセッサに割り当てられているマスタラインプロセスとも通信路をもっている．

ただし，プロセス間通信が一段余計にかかるので，処理速度が低下するとともに，プログラム構造が，元の並列オブジェクトモデルを素直に反映したものではなくなり，プログラム開発には不利となる．

ここで，このプロセス構造に変更したことによってどの程度プロセッサ間通信が削減できたかを見積もってみよう．

いま，プロセッサの数を P，一つのプロセッサが担当するマスタラインの数を Mとする．

全マスタラインプロセスが互いに直接メッセージ交換を行なう場合，一つのプロセッサをわたる出力のた



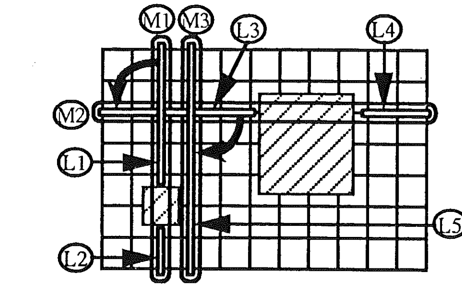


図 3.2.5 ディストリビュータプロセスを付加したプロセス構造

めのメッセージ通信路の数は，一つのマスタラインプロセスに対して $M(P-1)$ 本なので，全部で $M^2(P-1)$ 本となる．また，入力のためのメッセージ通信路の数は，出力と同じだけ必要なので，一つのプロセッサ当たりの全通信路数は

$$2M^2(P-1) \tag{3.1}$$

本となる．

一方，改良したプロセス構造では，一つのプロセッサをわたる出力のためのメッセージ通信路の数は，$M(P-1)$ 本である．また，入力のためのメッセージ通信路の数は，一つのマスタラインプロセスに対して $(P-1)$ 本なので，出力と同じく $M(P-1)$ 本となる．よって，一つのプロセッサ当たりの全通信路数は

$$2M(P-1) \tag{3.2}$$

となる．

いま，旧プロセス構造における一つのプロセッサ当たりの最大マスタラインプロセス数を M_{old}，改良したプロセス構造における一つのプロセッサ当たりの最大マスタラインプロセス数を M_{new} とすると，式 (3.1) と式 (3.2) とから

$$M_{new} = M_{old}^2 \tag{3.3}$$

が得られる．

実験によると旧プロセス構造では，PIM/m($P=256$) 使用時に 750×750 程度の格子規模でプロセッサ間通信路を表現するためのメモリが不足してしまった．

このことから，式 (3.3) により，PIM/m($P=256$) では，改良したプロセス構造にすると 4395×4395 規模まで適用可能であると考えられる．実際，テストデータを用いて実験した結果，4500×4500 規模でオーバフローが生じた．

C. 複数ネット間の競合

[問題]

複数ネットを同時配線すると，異なるネット間で同じ配線領域を取り合うことがある．このような状況では，一つのオブジェクトに対するメッセージの順序化により，早いもの勝ちで配線が行なわれる．あとから到着した配線要求は達成されず，バックトラックを発生する．(普通は，期待位置計算時に既配線を除外するので，はじめから別経路をとるが，競合状態のときにだけバックトラックとなる．) ところが，競合に勝って先に配線領域を確保したネットが，あとになって結局バックトラックせざるを得ないことがある．この場合，別ネットが競合した配線領域は未使用に戻されるが，先に競合に負けたネットは二度と同領域を探索しない．すなわち，同領域は使われずに残ってしまう場合がある．それが迂回の多い配線長の長い配線を生成し，配線品質の低下や配線率の低下を招く場合がある．

表 3.2.1 データの仕様

データ	格子規模	ネット数
$D1$	322×389	71
$D2$	262×106	136
$D3$	2746×3643	556
$D4$	2524×3424	1088

[対処方法]

このような相互干渉を起こりにくくするためには，ネットの投入順や，同時投入の本数を制御するなどの対処方法が必要となる．筆者らの配線プログラムでは，ネットの同時投入本数を自由に変えられる．いかにしてそれを制御するかは，文献 [16] を参照していただくこととし，ここではこのような相互干渉がどの程度配線率を低下させるのかを，LSI の実データを用いた実験によって見てみよう．

3.2.5 実験および評価

ここでは，(1) 問題の規模と台数効果との関係，(2) 並列度と配線率との関係，(3) 汎用計算機との性能比較の 3 点を明らかにするために 4 種類の LSI 実データを用いて実験を行なった．

それらのデータの仕様を表 3.2.1に示す．$D1$ は，接続すべき端子が比較的全体に分散しているデータである．また $D2$ は，局所的に端子が集中しているデータである．そして $D3$ と $D4$ は，大規模なデータである．

A. 問題の規模と台数効果

一つのプロセッサで実行した場合に比べて，複数のプロセッサを使用したときに何倍の速度向上になったかを台数効果と呼ぶ．

一般的に問題の規模が大きくなれば，並列に動作するプロセスの数が増えるので，台数効果も大きくなると考えられる．そこで，筆者らの並列配線プログラムに関して，データの規模と台数効果との関係を測定した．

その測定結果をグラフとしてまとめたのが，図 3.2.6 である．このグラフからデータの規模が大きくなるに従って台数効果が向上することがわかる．$D4$ のデータに対して，256 台のプロセッサを用いると約 92 倍の速度向上が得られることがわかった．今後，どの程度の規模のデータまで台数効果が向上するのかを調べる予定である．

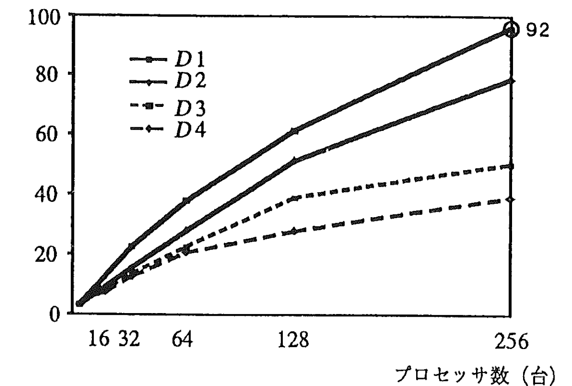


図 3.2.6 データ規模と台数効果

B. 配線率と並列度

端子が局所的に集中するデータに対しては，多数のネットを同時に配線すると，配線ネット間の競合が起こり配線率が落ちる可能性がある．そこで，$D1$ と $D2$ に関して，256 台のプロセッサを用い，同時に配線を実行するネット数 N と配線率 W との関係について測定した結果を図 3.2.7にまとめた．この図において，二つの縦軸はそれぞれ実行時間と配線率である．また，横軸は同時に配線を実行するネットの数を示している．すなわち，同時に実行するネット数が 1 ということは，並列処理の効果としては期待位置の並列計算のみである．この図から，端子位置が分散しているデータでは，配線率をあまり落とさずに処理速度も向上するが，局所的に端子が集中しているようなデータに関しては，複数ネットを同時配線すると配線率が低下してしまうことがわかった．ここで，期待位置計算部の並列処理による効果について考えてみよう．いま，$D2$ に関して，1 台のプロセッサを用いたときの実行時間は 70 秒であった．また，256 台のプロセッサを用いて 1 ネットずつ配線した結果は，図 3.2.7から約 12 秒である．よって，期待位置計算部の並列処理による効果は，5.7 倍であることがわかる．

C. 汎用計算機との性能比較

PIM の実験機である Multi-PSI を用いた実験結果 [14] によると，64 台のプロセッサを用いて台数効果が

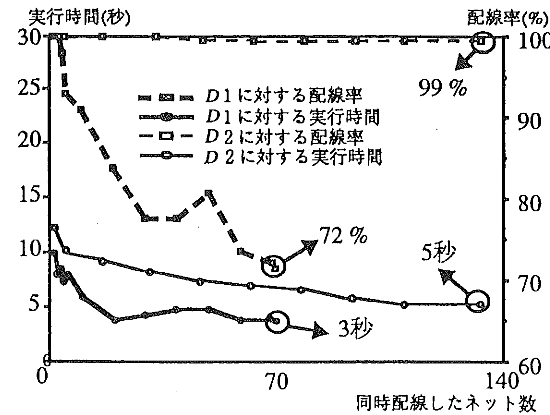


図 3.2.7 配線率と並列度

16倍出れば，15MIPSの汎用計算機 (IBM3090/400) 上でFORTRANによって書かれた同じアルゴリズムに基づく逐次プログラムと同程度の性能になることがわかっている．

PIM/mの単一プロセッサ性能はMulti-PSIの約2倍であることから，台数効果として92倍出ているデータ *D*4では，約170MIPS計算機上の逐次プログラムと同程度の処理速度であることがわかる．

3.2.6 むすび

LSIの2層配線を対象とした新しい並列配線手法を紹介した．その特徴は，すべての配線線分をオブジェクトとし，並列オブジェクトモデルに基づき並列アルゴリズムが設計され，メッセージ交換を行ないながら配線経路を決定する点にある．また，それを分散メモリ型並列マシンのPIM/m上にKL1言語を用いて実現し，LSIの実データによる性能評価結果を示した．

速度向上に関する実験では，1台のプロセッサを使用した場合と比較して，256台のプロセッサを用いると約92倍の速度向上を実現した．データ規模の増大につれて速度向上も大きくなる傾向を示しており，大規模データでは，さらに高い効率が期待される．

並列度と配線率に関する実験では，端子の位置が分散しているデータでは，台数効果および配線率とも良好な結果を得た．しかし，端子の位置が局所的に集中しているデータに対しては，複数ネットを同時配線すると，ネット間の競合により配線率が低下することが確認された．この問題に関しては，効果的なネットの同時投入量の制御，未結線ネットの自動再試行などを検討する必要がある．

また，大規模データに対しては，PIM/mを用いて92倍の台数効果を得ることができた．この処理速度は，170MIPS計算機上で同じアルゴリズムに従ったFORTRANを使ってコーディングした逐次プログラムに相当することがわかった．

今回の評価実験により得られた最も大きな課題は，いかにして並列配線時に配線率を低下させないようにするかということである．配線率が低下する原因の一つは，複数ネットを同時配線するときに配線順序が保証されなくなることである．現在ICOTでは，配線順序を保証する並列配線プログラムを開発しており，評価を進めている [16]．

本配線プログラムは，プロセスの分裂・統合，通信路の張替えなどの処理を部分的に同期を取りながら行ない経路を決定していく．そのため，複雑な同期機構を記述しなければならない．また，プログラムの処理速度を向上させるために，各プロセスのプロセッサへの割りつけ方法を変えて評価を行なわなければならない．これらのことをKL1言語を用いて実現できたということは，KL1言語のもつ並列処理記述能力の高さを示しているといえる．

3.3 論理シミュレーション

3.3.1 はじめに

本節では，ICOTの並列LSI-CADグループで開発したプログラムの一つ，並列論理シミュレータを紹介しよう．

論理シミュレータは，LSIの設計において，設計された回路が正しく動作するかどうかを検証する工程 (論理シミュレーション工程) で用いられる．この工程に費やされる時間はきわめて大きいため，高速な論理シミュレータが強く望まれている．並列計算機の利用は，高速化の手段として有望である．このことからICOTの並列LSI-CADグループでは，並列論理シミュレータの開発に取り組んだ．

ICOTの並列論理シミュレータは，タイムワープ機構 (3.3.4 B.項で詳説) と呼ばれる時刻管理機構を採用していることを大きな特徴としている．論理シミュレータではイベント法と呼ばれるシミュレーション方法を用いることが多い．この方法では，イベント処理の時刻 (順序) を管理する必要があるが，この時刻管理機構がイベント法の並列化の鍵となる．タイムワープ機構は，分散的な時刻管理機構の一つであるため，PIM/Multi-PSIのようなMIMD型分散メモリマシンに適した方法と考えられる．実際，この機構を採用することによって，並列論理シミュレータをPIM/Multi-PSI上で効率的に動作させることに成功した．

タイムワープ機構は，他の時刻管理機構に比べ，やや複雑な機構である．そもそも並列プログラムは，逐次プログラムに比べ開発コストが高いのが通常であるから，タイムワープ機構のプログラミングはきわめて困難であると考えられる．このため，従来はこの機構を用いた論理シミュレータの例はほとんどなかったといってよい．

幸いなことに，ICOTではKL1という並列プログラム言語を開発しており，筆者らはこれを利用することができた．すでに述べたように，KL1言語による並列プログラム開発は，従来のC言語などによるものに比べ，ずいぶん容易となる．タイムワープ機構を用いた論理シミュレータは，比較的短期間に開発されたが，これはKL1言語の存在に負うところが大きい[†1]．

以下，論理シミュレーション，およびイベント法の時刻管理機構について説明する．そのあと，ICOTの並列論理シミュレータの設計，および評価について述べる．

3.3.2 論理シミュレーション

まず，論理シミュレーションとはどのようなものか，すこし詳しく説明したい．

通常，LSIの設計は，仕様設計など大まかな設計から開始され，徐々に細かい部分の設計へ進む．そして，最終的にはAND，ORなどの機能をもつ素子，すなわち論理ゲート (通常，数個のトランジスタから構成される) で構成された論理回路として表わされるようになる (図 3.3.1)．

論理回路の設計過程では，何らかの設計誤りが含まれることが多い．このことは，ソフトウェア開発において，いきなりバグのないプログラムを作ることがきわめて難しいということを考えればご理解いただけると思う．もし，LSIチップを製造してしまったあとにこのような誤りが発見されれば，論理回路の設計以降の工程をすべてやり直さなければならない．これではLSIの開発期間が長くなり，設計コストが非常に高くなってしまう．

†1 並列論理シミュレータの第1版は，回路分割部も含め，わずか3か月×1人で開発された．もちろんその陰には，筆者が日夜プログラミング/デバッグに励んだことは書き留めておきたい．

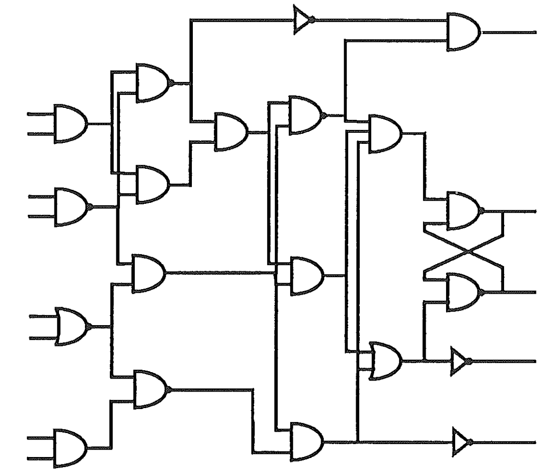


図 3.3.1 論理回路

そこで，実際のLSI設計では，論理設計からレイアウト設計に移る段階で，計算機シミュレーションによって，論理回路が正しく動作するかどうかを調べる．これが論理シミュレーションと呼ばれる工程である．

A. 素子モデル

論理シミュレーションは，素子をどのレベルで扱うかによっていくつかに分けられる．最も一般的なのがゲートレベルシミュレーションである．これは，回路をAND，OR，NOTなどの論理ゲートレベルで記述し，シミュレーションを行なうものである．

しかし，対象回路がメモリなどゲート表記が難しい素子を含む場合，ゲートレベルだけでなく，機能レベルで記述された素子も扱う必要がある．このようなものを混合レベルシミュレーションという [17, 18]．さらには，回路全体を演算器やレジスタなど，機能レベルの素子で表現する機能レベルシミュレーションと呼ばれるものもある[†2]．

反対に，ゲートレベルよりさらに細かいレベルのシ

†2 これは論理設計の前段階である機能設計などの工程で行なわれるシミュレーションである．

表 3.3.1 5 値 AND ゲートの真理値表

	0	1	↑	↓	X
0	0	0	0	0	0
1	0	1	↑	↓	X
↑	0	↑	↑	X	X
↓	0	↓	X	↓	X
X	0	X	X	X	X

ミュレーションもある．MOS 素子で構成される回路や，トライステート素子を含む回路[†1]では，一つの信号線を双方向に信号が流れる場合がある．これらは正スイッチ，負スイッチ，減衰器，電荷井戸によって表現することでシミュレーションできる．これはスイッチレベルシミュレーションと呼ばれる [19]．

B.　信号値モデル

話をゲートレベルの論理シミュレーションに戻そう．論理シミュレーションでは，2 値論理回路が対象であるわけだから，当然ながら信号値は 0 または 1 のいずれかになる．しかし，通常は 0，1 のほかに不確定の値を示す X を加えた 3 値モデルや，ハイインピーダンスを示す Z，立上がり，立下がりの遷移状態を示す ↑・↓ などを加えた多値モデルが用いられる．一例として，5 値 {0，1，↑，↓，X} を用いた場合の AND ゲートの真理値表を表 3.3.1に示す．

C.　遅延モデル

ゲートでは，入力信号が変化した場合，その影響が出力信号に現われるまでは，ある程度の時間を必要とする．これをゲート遅延という．遅延をどうモデル化するかは，シミュレーションの正確さを決める大きな要因である．

最も単純な遅延モデルは 0 遅延モデルである．これは，すべてのゲートの遅延を 0 とみなすものである．このモデルは，組合せ回路[†2]や同期回路[†3]の論理検証のみを行なう場合に用いられる．一方，回路中の全素子が同じ遅延時間をもつと仮定するものは単一遅延 (unit delay) モデルと呼ばれる．これにより，非同期回路[†4]の扱いも可能となる．一般には，素子ごとに異なった遅延をもつと考えるべきであろう．これに対応するものがノミナル遅延 (nominal delay) モデルであり，現在最も広く用いられている．

なお，さらに詳細な遅延モデルとしては，信号の立上がり時と立下がり時の遅延に異なる値を与える立上がり立下がり遅延モデル，各ゲートの最大，最小遅延時間を反映した最大最小遅延モデル，微小幅のパルス入力を出力に反映させない慣性遅延モデルなどがある．詳しくは文献 [20, 21] を参照されたい．

3.3.3　シミュレーション方法

本節では，どのようにシミュレーションが行なわれていくのか，その方法について説明する．

A.　イグゾースティブ法

シミュレーションという言葉を聞けば，流体解析など連続系の数値シミュレーションを思い浮かべる読者が多いと思う．連続系シミュレーションでは，シミュレーション時間を微小な時間に区切り，各時刻ごとにシミュレーション対象の状態を計算する．イグゾースティブ法は，このような連続系シミュレーションに近い方法である．

イグゾースティブ法では，シミュレーション時間を微小な時間 (単一遅延モデルのシミュレーションでは，ゲート遅延時間) に区切り，各時刻ごとに全ゲートについて計算してゆく (図 3.3.2)．

実際の回路では，入力信号値が変化するゲートの割合は大変小さいため，後述のイベント法に比べ，計算量がきわめて大きくなる．このため，ソフトウェアシミュレータでは，イグゾースティブ法はほとんど用いられていないが，処理が単純なため，ハードウェア化には適している．

B.　レベルソート法

レベルソート法は，回路の論理のみを検証する場合に適した方法である．

レベルソート法では，まず回路を構成するゲートにレベルづけを行なう．どのゲートについても，そのレベルは入力側のゲートのレベルより大きくなるように

†1　これらは架空のゲートを導入するなどしても表記できるが，その場合，きわめて複雑な表現になる．
†2　現在の入力のみによって出力が決まる回路．
†3　現在の入力のみならず，入力の過去の履歴によって出力が決まる回路 (順序回路) のうち，クロックなどで同期的に動作するもの．
†4　順序回路のうち，同期的に動作しないもの．

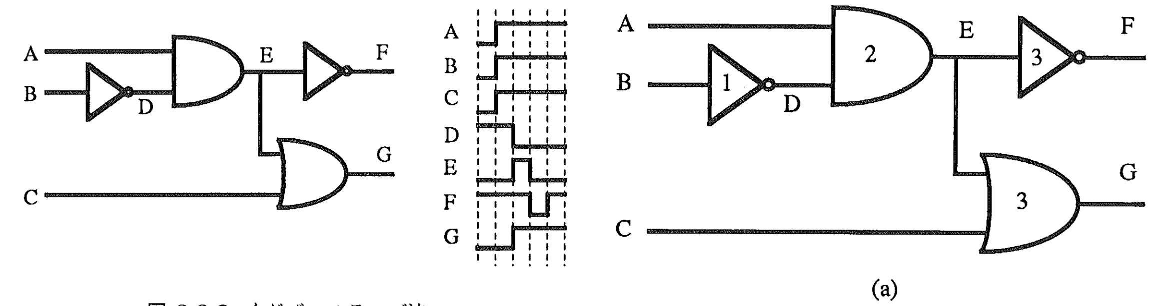


図 3.3.2 イグゾースティブ法

レベルづけが行なわれる．続いて，各ゲートの演算に応じた命令を生成し，レベル順に実行する．

たとえば，図 3.3.3(a) のような回路の場合，各ゲートはそれぞれ同図に示すようにレベルづけされる．これらのゲートに対応した命令は同図 (b) のように生成される．シミュレータは，外部からの入力信号に従い，生成された命令を実行すればよい．

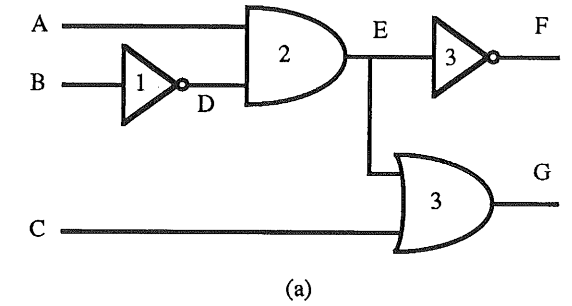
この方法は，きわめて高速であるが複雑な遅延モデルを扱えないという欠点をもつ．また，回路内にループが存在した場合，それを切断してレベルづけを行なう．このため，非同期回路などのシミュレーションは困難である．

C.　イベント法

イベント法は，入力信号値が変化 (これをイベントが発生したと見なす) したゲートについてのみ計算を行なうものである．一般に，外部からの入力信号が変化した場合，内部の信号線において信号値の変化が起こるものの割合は，高々10 %程度といわれている．したがって，イベント法はイグゾースティブ法に比べ，計算量が少なくなる．

またイベント法では，レベルソート法のようにループの切断などを行なわず，回路記述に忠実に，イベントの伝播をたどっていく．このため，非同期回路のシミュレーションが可能であり，複雑な遅延モデルも扱うことができる．現在使われているほとんどのシミュレータは，このイベント法を用いたものである．

イベント法では，発生したイベントに応じた計算を，必ず発生時刻順を守って行なう必要がある．こうしなければ，正しいシミュレーション結果を得ることができない．このため，イベント処理の時刻管理機構が必要となる．



(a)

D = not (B);
E = and(A,D);
F = not(E);
G = or(C,E);

(b)

図 3.3.3 レベルソート法

従来の論理シミュレータでは，イベントを集中的に管理する時刻管理機構が通常用いられてきた．この機構はタイムホイルと呼ばれている (図 3.3.4)．

タイムホイルはいくつかのスロットが連なった輪である．各スロットはあらかじめ決められた時間間隔 (通常，ゲート遅延値の最大公約数) ごとの時刻に対応する．発生したイベントは，その生起時刻のスロットにいったん登録される．

タイムホイルに登録されたイベントは，そのうち，最も小さい時刻のイベントから順に処理を行なう．ゲート遅延値は非負の値であるため，この機構により，必ず時刻順にイベント処理を行なうことができる[†1]．

なお，タイムホイルの大きさ (1 周分のスロット数) は，ゲートの最大遅延時間分あればよい．しかし，極端に大きな遅延をもつ素子を扱う場合，ホイルの大きさを小さく抑えておき，ホイル 1 周内に登録できないものは，別に用意されたオーバフローリストに登録するという方法を用いる場合もある [21]．

3.3.4　イベント法による並列論理シミュレーションの時刻管理

それでは，いよいよ論理シミュレーションの並列化に話を進めよう．ここでは，最も一般的に用いられているイベント法の並列化を考える．

†1　処理したイベントの結果が，過去に影響を与えることはない．

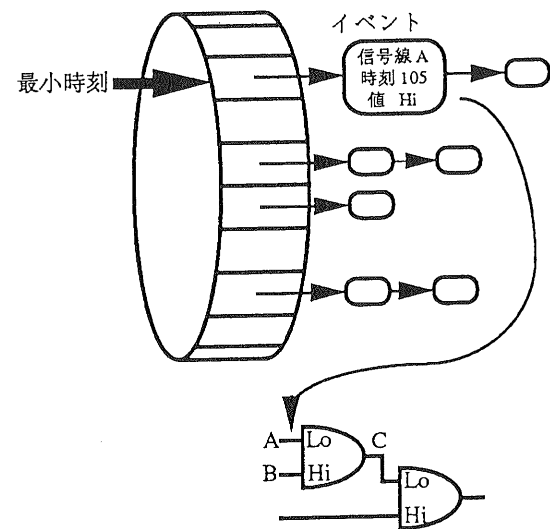


図 3.3.4 タイムホイル

論理シミュレータを並列化する場合，どのレベルで並列性を抽出するかをまず最初に考える必要がある．たとえば，一つのイベント処理のなかに存在する小粒度の並列性を利用することも可能であるし，同時に処理できる複数イベントのレベルでの並列性も利用可能である．これは，使用する並列計算機のアーキテクチャを考慮して決定されるが，ここでは後者の並列性を抽出する場合を考えよう．

タイムホイルを用いた時刻管理機構は，逐次計算機上では非常に効率のよいものである．しかし，この方法は以下に示す二つの理由から，並列処理，特に PIM のような大規模 MIMD 型分散メモリ計算機上での処理には適していない．

- 抽出できる並列性は，高々同時刻に生起するイベント数のみである．一般に同じ時刻に生起するイベント数はあまり多くない．ゲート遅延の単位をより細かくした場合，この傾向はますます強まる．
- 多数のプロセッサを用いる場合，集中管理部分がボトルネックになる．また，プロセッサ間通信コストが高いような疎結合 MIMD 型分散メモリマシンでは，集中管理部分との頻繁なメッセージ交換が大きなオーバヘッドとなる．

したがって，大規模 MIMD 型分散メモリマシン上で，効率的な論理シミュレーションを行なおうとする場合，

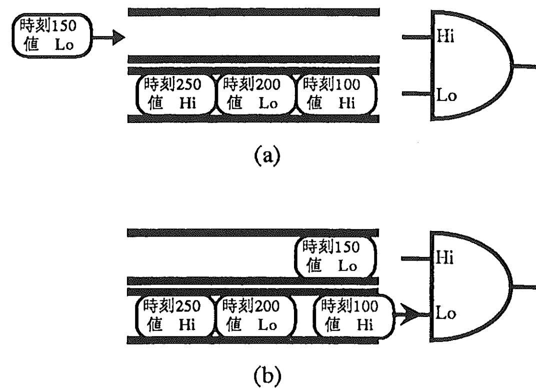


図 3.3.5 保守的な機構

時刻を分散的に管理する機構が有利であると考えられる．

分散的な時刻管理機構は，以下に示すように，保守的 (conservative) な機構 [22] と楽観的 (optimistic) な機構 [23] に分けられる．

A. 保守的な機構 (Chandy-Misra の方法)

いま，各ゲートをオブジェクトに見立て，これらがメッセージを交換することでシミュレーションが進行するモデルを考えよう．メッセージは，イベント情報とその発生時刻の情報をもつとする．また前提として，同一信号線上ではメッセージの送受信順序が保存されるとする．

保守的な機構では，ゲートオブジェクトでは，そのゲートのすべての入力信号線上に最低一つのメッセージが受信されるまで処理を開始しない (図 3.3.5 (a))．すべて揃った時点で，そのうち最小のタイムスタンプ値をもつメッセージに対して処理を行なう (図 3.3.5 (b))．前提から，以後受信するメッセージの時刻は，必ず処理されたメッセージの時刻以上であるから，この処理は正しい．したがって，このゲートでは必ず正しい時刻順にメッセージを送信する．

保守的な機構の問題点は，しばしばデッドロックが発生するということである．デッドロックの発生原因にはさまざまなものがあるが，ここでは典型的な例として，ループ構造をもつ回路の場合について説明する．

図 3.3.6(a) では，時刻 146 と時刻 150 のメッセージが到着した状態を示している．このとき，まず時刻 146 のメッセージが評価される．評価の結果，出力値が

変化しないので，このゲートからメッセージは送り出されない．その結果，このゲートの下方の信号線には以後メッセージが届かず，したがって永遠に時刻 150 のメッセージの評価ができない (図 3.3.6(b))．この状態がデッドロックである．

デッドロックの問題を解決する方法の一つに，ヌルメッセージを用いる方法がある．図 3.3.6(a) の場合において，出力値が変化しない場合でもメッセージを出す場合を考えてみよう (図 3.3.6(c))．このメッセージはループを 1 周して，下方の信号線に到着する (図 3.3.6(d))．メッセージの時刻は，この間に通過したゲート分の遅延値 8 が加算され，154 となっている．その結果，時刻 150 のメッセージが評価できることになる．このようなメッセージがヌルメッセージである．ヌルメッセージはイベントには対応しないが，ゲートの時刻を進める役割をはたす．

しかしながら，単純にヌルメッセージを送る方法では，ヌルメッセージが大量に発生してしまう問題がある．一般に論理回路の平均的な出力先ゲート数は 1 より大きいため，ヌルメッセージは指数関数的に増大する．したがって，ヌルメッセージを削減する，あるいは不要なヌルメッセージを発生させないような工夫が必要になる．このような工夫としては，タイムアウト法やバッファリング法，問合せ法などが提案されている．これらの詳細は文献 [22, 24, 25] を参照していただきたい．

B. 楽観的な機構 (タイムワープ機構)

タイムワープ機構は，ゲートごとに入力メッセージの待合せを行なわないという点で保守的な機構と大きく異なる．また，同一信号線上でメッセージ送受信順序が保存されている必要もない[†1]．

タイムワープ機構では，「メッセージは時刻順に到着するだろう」という楽観的な予測に基づいて処理を進める．すなわち，メッセージが時刻順に到着している限り，即座にそのメッセージの評価を行なってよいとする (図 3.3.7(a))．

しかし，並列処理では，プロセッサ間をまたがってメッセージの送受信が行なわれることがあり，しばしばメッセージが時刻順に届かない場合が発生する (図 3.3.7(b))．このような場合に備えるため，各ゲートで

†1 順序が保存されている場合には，後述のように，その性質を利用して処理を簡略にすることが可能である．

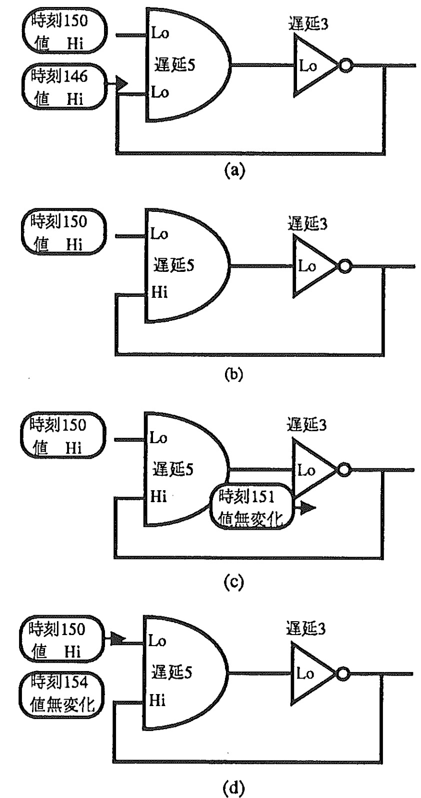


図 3.3.6 デッドロックとヌルメッセージ

はメッセージの履歴を保存しておく．そして，時刻の小さいメッセージが遅れて届いた場合，その時刻まで履歴を巻き戻す (この処理をロールバックという)．また，このときすでにいくつかのメッセージを誤って送信している場合には，これらを取り消すためのメッセージ (アンチメッセージと呼ばれる) を送信する (図 3.3.7(c))．ロールバック直後の状態は，遅れて届いたメッセージが，あたかも時刻順に到着した状態に見えることに注意していただきたい (図 3.3.7(c))．この状態から処理をやり直す．やり直された処理については正しいと見なせる．

なお，アンチメッセージを受信したゲートは，場合に応じて次の二つの処理のどちらかを行なう．

1) 取り消されるべきメッセージの処理をまだ行なっていなかった場合

アンチメッセージと取り消されるメッセージを消滅させる．

2) 取り消されるべきメッセージの処理をすでに行

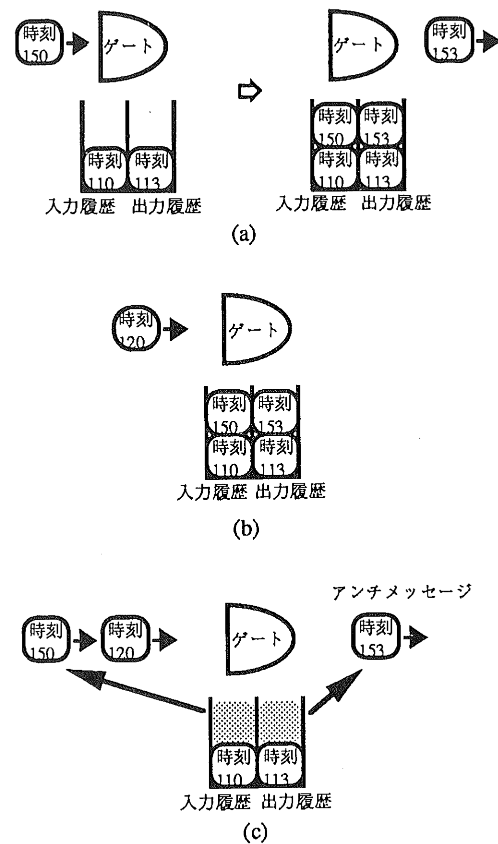


図 3.3.7 タイムワープ機構

なっていた場合

アンチメッセージの時刻までロールバックしたあと，1) の処理を行なう．

必要があれば，さらに次のゲートにアンチメッセージを送る．

アンチメッセージは，物理的な通信ネットワークのバグなどで消滅しない限り，必ず送り先に届くため，誤った出力メッセージと，その結果として発生する誤った処理は必ず取り消される．したがって，タイムワープ機構によるシミュレーションの結果は常に正しいことが保証される．

タイムワープ機構ではロールバックに備えて履歴を保存するため，大量のメモリを消費することが予想される．このため，ときどき大域的なシミュレーション時刻 (これを Global Virtual Time (GVT) という) を求め，不要となった履歴領域を解放する．GVT は，ある時点での全ゲートのシミュレーション時刻，およびゲート間を通過中のメッセージの時刻のうちの，最小値以下のものであればよい．GVT 以前にロールバックすることはないことから，GVT 以前の履歴領域は解放・再利用することができる．

GVT は終了検出にも用いられる．すなわち，GVT がシミュレーション終了時刻を越えた時点で，シミュレーション終了とみなす．

GVT を求める処理は，大域的な処理となるため，大規模な並列マシンではコストが高い．しかし，GVT 処理はそれほど頻繁に行なわれるわけではない．したがって，シミュレーション全体の時間に比べると，GVT 処理の時間は無視できると考えてよい．

3.3.5 PIM/Multi-PSI 上での論理シミュレータ：設計と実装

本項では，ICOT において，MIMD 型分散メモリマシンである PIM/Multi-PSI を対象マシンとして開発した並列論理シミュレータの詳細を述べる．特に，採用したさまざまな工夫について説明したい．

A. 論理シミュレータの対象回路モデル

ICOT の論理シミュレータの対象回路は，ゲートレベルで記述された回路のみとした．ただし，組合せ回路，同期回路のみならず，非同期回路も扱えるものである．

信号値は Hi，Lo，X(不定) の3値モデルを，また遅延はノミナル遅延モデルを採用した．スパイク[†1]についてはこれを検出し，取消しを行なう機能をもつ．

このシミュレータは，商用を目指したものではないことから，最低限の一般性をもたせた単純な仕様にしているが，機能の拡張は容易である．

B. シミュレーション方法の選択

さて，上記の仕様を満たしつつ，MIMD 型分散メモリマシン上で効率的に並列論理シミュレーションを行なうためには，適切なシミュレーション方法を選択する必要がある．

まず，ノミナル遅延モデルを対象遅延モデルとし，非同期回路もシミュレーション対象とすることから，必然的にイベント法を用いることになる．イベント法では，3.3.4項で述べたように3種類の時刻管理機構が考えられる．このうち，タイムホイルを用いた集中時刻管理機構 (以後タイムホイル機構と記す) は，分散メモリマシン上では非効率であることが容易に予測できる．したがって，分散的な機構のどちらかを選ぶことになる．

†1 現実には発生しないが，計算上現われる時間幅0の信号値変化．

筆者らはタイムワープ機構を選択した．この理由は以下のとおりである．

1) 保守的な機構での問題点である，デッドロック回避のコストが高いと判断した．ヌルメッセージを用いた単純な方法は，ヌルメッセージ数が爆発する．分散メモリマシン上でヌルメッセージを低コストで削減することは難しいと判断した．
2) タイムワープ機構では，ロールバック処理が大きなオーバヘッドとなるといわれているが，実際には，そのコストは小さく抑えることが可能と考えた．
3) タイムワープ機構は，いままでほとんど評価されていないが，これはそのプログラミングが複雑・困難であったことが一因である．このようなものを，KL1 言語によって短期間にプログラミングできることを示すことは意義が大きい．

最終的には，タイムホイル機構，保守的な機構による論理シミュレータも作り，これらとタイムワープ機構によるシミュレータとの性能比較を行なった．この結果については，3.3.6 B.項に紹介する．

C. 負荷分散

並列処理を行なう場合には一般にいえることであるが，負荷分散方法が処理速度に与える影響はきわめて大きい．したがって，どのような負荷分散を行なうかは並列論理シミュレータにとっても大変重要である．

並列論理シミュレーションは一つのメッセージ当たりの処理量が小さい，すなわち小粒度であるため，プロセッサ間通信のオーバヘッドが問題になる．また，タイムワープ機構ではロールバックの発生量も懸念される点である．この問題を分散メモリ型並列マシン上で実行する場合，(1) 負荷の均等分散，(2) プロセッサ間通信の低減，(3) 十分な並列性の抽出，の3点が負荷分散の目標となる．

最適な負荷分散のためには，この3点を適切に反映した評価関数を定義し，その値を最良にする負荷分散

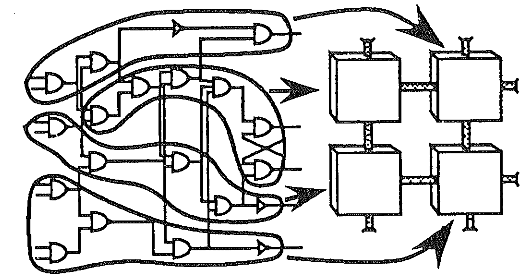


図 3.3.8 縦割り指向戦略による回路分割

方法を求めなければならない．しかし，一般にこのような問題は，現実的な時間内に解けないため，何らかのヒューリスティクスによって，そこそこ満足する解を求めることになる．

今回，上記目標の3点をある程度満足し，かつ計算時間がシミュレーション時間に比べ十分に小さい負荷分散方法として，縦割り指向戦略と名づけた戦略により回路を分割し，得られた部分回路を各プロセッサに静的に割り当てる方法を試みた．

縦割り指向戦略は，連結したゲートはできるだけ同じプロセッサに割り当てるとともに，出力先が複数分岐する部分から並列性を抽出することを意図している．この戦略では，回路入力端子から順にゲート接続関係をたどり，縦方向につながったゲートを一つのグループにまとめることによって，回路をいくつかの部分回路に分割する．ここで，複数の出力先ゲートが存在する場合，そのうちの1ゲートのみをグループ中のゲート群に取り込み，他のゲートは，新たに別のグループ化処理の開始点とする．全ゲートのグループ化終了後，小さい部分回路については，接続関係にある別の部分回路に統合する．また，極端に長い部分回路はいくつかに切り分ける．最後に，生成された部分回路をランダムに各プロセッサに割り当てる (図 3.3.8)．

D. スケジュール

通常，シミュレーション対象回路に含まれるゲートの数は，使用するプロセッサ数に比べ，はるかに大きい．したがって，シミュレーション実行中，各プロセッサには複数のゲートオブジェクトが割り当てられ，複数の (未処理) メッセージが存在する．この場合，各プロセッサ内でメッセージのスケジューリングを適切に行なうことは，ロールバック頻度低減に役立つ．

スケジューリング戦略としてはいくつかの方法が提

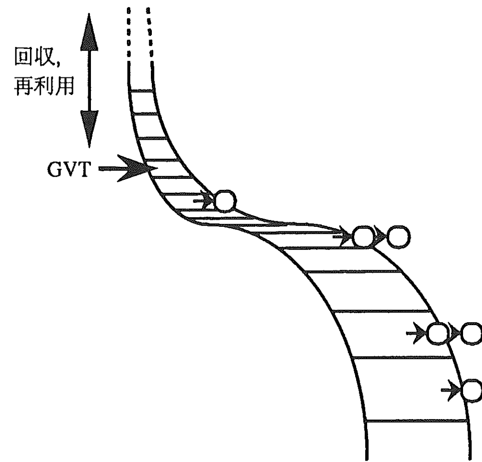


図 **3.3.9** スケジューラ

案されているが [26]，論理シミュレーションでは一つのメッセージ当たりの処理量が小さい (小粒度) ため，スケジューリング処理の軽いものが望ましいと考えられる．このことから本システムでは，プロセッサに到着している未処理メッセージのうち， 最小時刻をもつものから処理を行なう戦略 (以後，最小時刻優先戦略と呼ぶ) を採用した．

このスケジューリング戦略を採用すれば，その実現のための機構は，タイムホイルにきわめて類似したものとなる．すなわち，スケジューラは各時刻に対応したスロットをもつ．一つのスロットには，同じ時刻をもつメッセージが登録される．スケジューラは登録されているメッセージのうち，最小時刻のものから順に受信側ゲートに送る．

ただし，このスケジューラは輪構造をもつことはできない．これは，すでに評価したメッセージより小さい時刻をもつメッセージが新たに登録される場合があるからである．したがって，テープ構造をもつことになるが，GVT 以前のスロットについては回収・再利用される (図 3.3.9)．

E. アンチメッセージ削減

Jefferson によるもともとのタイムワープ機構では，同一信号線上でのメッセージ送受信において順序が保存されていることを仮定しないため，取り消すべきメッセージすべてに対しアンチメッセージを送る (図 3.3.10)．

本シミュレータのプログラムでは，信号線は KL1

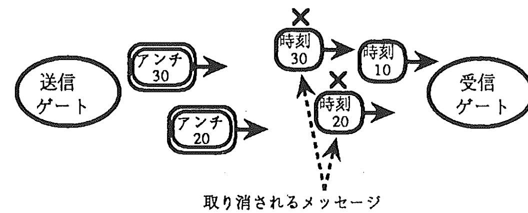


図 **3.3.10** アンチメッセージ送信

のストリームとして表現され，メッセージはストリーム上を流れるデータとして表現されている．KL1 では，同一ストリーム上のデータの送信順が保存されるため，送信ゲートと受信ゲートの間ではメッセージの送信順序は保存される．

本シミュレータでは，この環境のもとで有効な福井の方法 [27] に改良を加えたアンチメッセージ削減方法を用いている．はじめに，元となった福井の方法を述べる．

- **アンチメッセージ削減方法 1**
  取り消すべきメッセージのうち，最小の時刻をもつメッセージに対応したアンチメッセージ M_{anti}のみを送る．ここで M_{anti}のもつ時刻を $TS(M_{anti})$ とする．このとき，受信者は同一信号線上で受信したメッセージのうち，M_{anti}到着以前に受信し，かつ $TS(M_{anti})$ 以上の時刻をもつメッセージのみを取り消せばよい (図 3.3.11(a))．

本システムでは，ロールバックが発生したゲートから，アンチメッセージの送信直後に通常のメッセージが送信される場合が多くあることに着目し，次に示す改良を加えた．

- **アンチメッセージ削減方法 2**
  送信側ゲートでロールバックが発生すると同時に，取り消すべき一連のメッセージの最小時刻以下の時刻値をもつ新たなメッセージ M_{new}が発生する場合，単に M_{new}の送信を行なうだけでアンチメッセージの送信は行なわない．受信者は，同一信号線上で M_{new}以前に受信したメッセージのうち，$TS(M_{new})$ 以上の時刻をもつメッセージについて取消し処理をすればよい (図 3.3.11(b))．

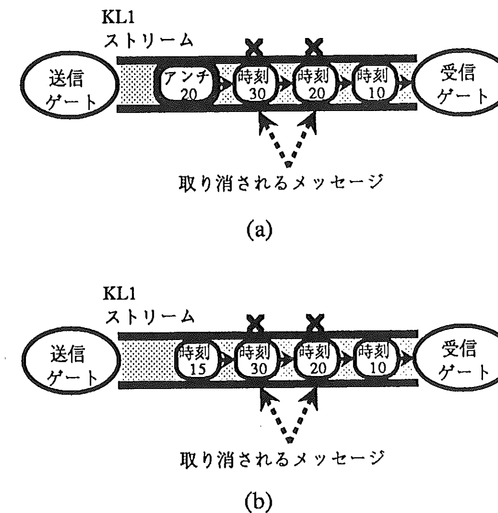


図 **3.3.11** アンチメッセージ削減方法

表 **3.3.2** 対象回路 (Multi-PSI にて)

回路	s1494	s5378	s9234	s13207
ゲート数	683	3853	6965	11965

3.3.6 評価

Multi-PSI 上で，ISCAS'89 のベンチマーク[†1]から四つの順序回路についてシミュレーションを行ない，論理シミュレータの性能，速度向上，およびさまざまなオーバヘッドについて計測した．

対象回路のゲート数を表 3.3.2に記す．なお，このシミュレータは機能素子を対象素子としないため，D フリップフロップはゲートに展開している．

今回の実験では，各ゲートにすべて 1 単位時間の遅延値を与えたが，単一遅延モデルにしたわけではないため，これによる処理の簡易化は行なっていない．また，クロックの周期は 40 単位時間とし，クロック線以外の入力端子には，クロックの立ち上がりに同期して，ランダムに信号値が変化するような入力信号を与えた．

A. 測定結果

図 3.3.12に各回路のシミュレーションにおける台数効果のグラフを示す．この図から，s13207，s5378 に

†1 これらのベンチマークはもともとテスト生成用のものである．

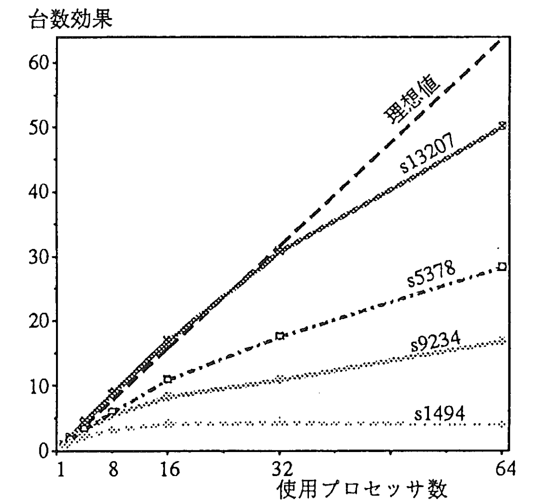


図 **3.3.12** 台数効果 (Multi-PSI にて)

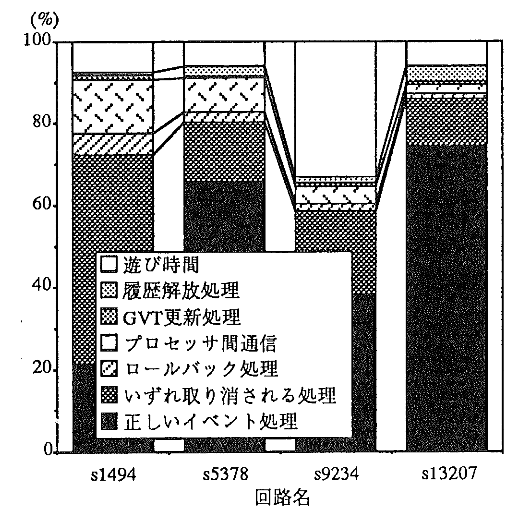


図 **3.3.13** 各種処理時間の割合 (Multi-PSI にて)

ついては良好な台数効果が得られていることがわかる．一方で，s9234，s1494 では台数効果がそれほどよくないことにも気づく．

この原因として，ロールバック処理，GVT 更新処理，およびメッセージ通信のオーバヘッドが大きいことが考えられる．全実行時間に対する各種処理時間の割合を図 3.3.13に示す．なお，これらの値は，64 プロセッサ使用時の全プロセッサでの平均値を示している．この図から，ロールバックや GVT 更新の処理時間は，64 プロセッサ使用時でも非常に短く，これら

表 3.3.3 並列性

回路	s1494	s5378	s9234	s13207
並列性	18.88	35.52	17.95	43.24

がシミュレータ性能に与える影響は小さいことがわかる．また，プロセッサ間通信が全体の処理時間に占める割合も小さいことがわかる．

それでは，いったい何がs9234やs1494における台数効果を制限しているのであろうか．筆者らは，さらに各回路のシミュレーションにおける並列性を調べた．並列性は，得られる台数効果の上限値を表わすと考えられる．ここでいう並列性は，プロセッサ間通信やロールバック，GVT更新の処理時間が無視できるような環境における台数効果とする．実際に，このような環境を生成し，並列性を測定した結果を表3.3.3に示す．この表から，s1494, s9234では，並列性がきわめて小さいことがわかる．したがって，これらのシミュレーションで高い台数効果が得られないのは当然であったといえよう．並列性が小さい場合には，多くのプロセッサは遊んでしまうか，あるいはいずれ取り消されてしまう処理ばかりを行なってしまうことになる．実際，図3.3.13を見れば，s9234ではプロセッサの遊び時間の占める割合が大きく，またs1494では結局取り消されてしまう処理にかかった時間の占める割合が大きい．

これらから，対象回路のシミュレーションでの並列性が高い場合，タイムワープ機構は効率的に並列性を引き出すことができ，懸念されたロールバック処理，GVT更新処理，およびプロセッサ間通信のオーバヘッドは小さいといえよう．

なお，スーパリニア効果(プロセッサ数よりも大きな台数効果を得ること)にお気づきの読者もいると思う．これは，GVT更新後の履歴解放処理が原因であることがわかっている†1．並列性よりも実際の台数効果のほうが高いという一見奇妙な事実も，同じ理由による．これらの詳細については文献[28]を参照していただきたい．

†1 もう少し詳しくいえば，履歴を入力線ごとに保存していること，また全体のメモリ量が使用プロセッサ数に比例していることに起因している．

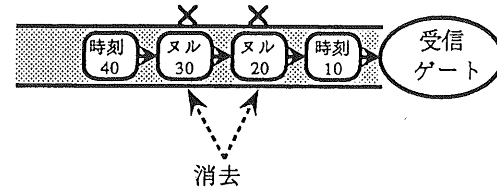


図 3.3.14 ヌルメッセージ削減機構

B. 他の時刻管理機構との比較

タイムワープ機構が効率的な並列論理シミュレーションを実現することはわかった．しかし，他の時刻管理機構を用いた場合に比べ，いったいどの程度有利なのであろうか？

この点を明らかにするため，筆者らは保守的な機構およびタイムホイル機構を用いた論理シミュレータを作り，タイムワープ機構との性能を比較してみた．

a. 保守的な機構による論理シミュレータ

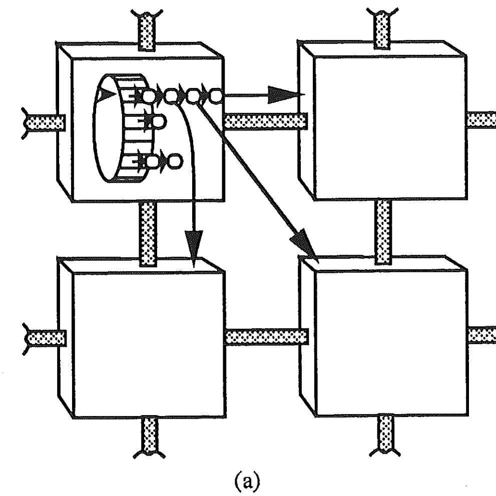
3.3.4 A.項に述べたように，保守的な機構における最大の問題点は，回路がループ構造をもつ場合などにデッドロックが発生することである．ヌルメッセージを用いればデッドロックを回避できるが，ヌルメッセージが多量に生成されてしまう点が問題となる．したがって，何らかの方法でヌルメッセージを削減しなければならない．

今回の保守的な機構による論理シミュレータでは，バッファリング法を用いてヌルメッセージを削減するようにした．例を用いて，この方法を簡単に説明する．図3.3.14は，同一信号線上でヌルメッセージを受信した直後に，さらに別のメッセージ(これがヌルメッセージであるか通常のメッセージであるかは問わない)を受信した場合を示している．この場合，ヌルメッセージを消去しても，あとに続くメッセージにより時刻を進めることができる．したがって，後続のメッセージがあるヌルメッセージはすべて消去できる．

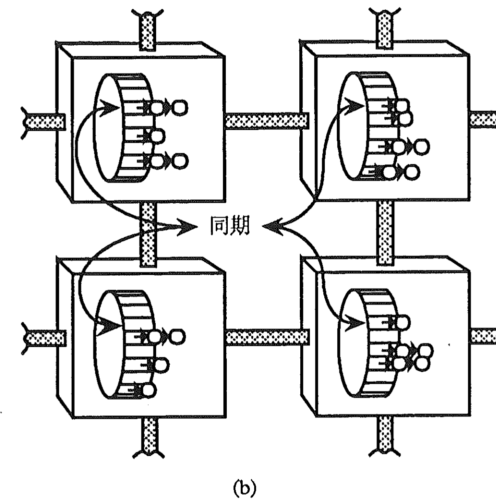
b. タイムホイル機構による論理シミュレータ

タイムホイル機構の並列化方法としては，二通り考えられる．一つは，シミュレータ全体で一つのタイムホイルをもたせ，同時刻に発生する複数イベントを異なるプロセッサで処理する方法である(図3.3.15(a))．もう一つは，すべてのプロセッサがタイムホイルをもち，それぞれ割り当てられた部分回路に関するイベント処理を同時に行なうものである(図3.3.15(b))．

前者の方法は，イベント処理ごとにプロセッサ間通



(a)



(b)

図 3.3.15 タイムホイル機構の並列化

信が必要であり，またタイムホイルへのイベント登録，取出し処理が一プロセッサに集中する．一方，後者の方法では，時刻を進めるごとに大域的な同期が必要になる．それぞれに問題点があるが，筆者らは後者の方法がMulti-PSI向きであると考え，これを採用した．

c. 性能比較

上記2種類の時刻管理機構による論理シミュレータ性能をそれぞれ測定した．これらとタイムワープ機構によるシミュレータとの性能比較結果を図3.3.16に示す．なお，対象回路としてs13207を，負荷分散戦略はタイムワープ機構のシミュレータと同様，3.3.5 C.項で述べた方法を，さらに保守的な機構でのスケジューリング戦略は3.3.5 D.項で述べた戦略を用いた．

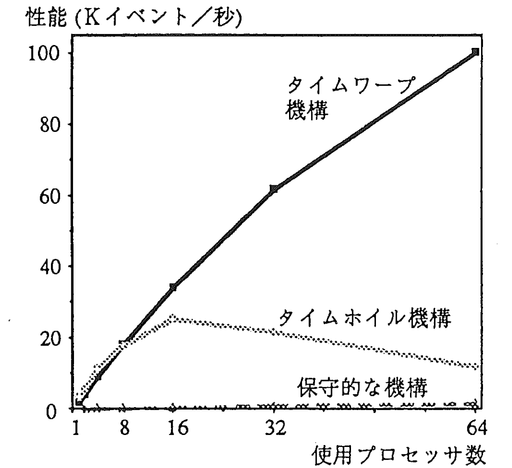


図 3.3.16 各時刻管理機構の性能比較

まず，保守的な機構について見てみよう．速度向上の面では良好であるが，絶対性能はきわめて悪いことがわかる．1プロセッサ使用時のヌルメッセージを含めたメッセージ処理性能を測定したところ，2006メッセージ/秒であり，この値はタイムワープ機構のものとほぼ等しいことがわかった．そこで，さらにヌルメッセージ数を計測したところ，なんと生成されたメッセージのうち，約97%までがヌルメッセージであり，イベントに対応したメッセージは，わずか3%であったことが判明した．このことから，ヌルメッセージを用いた保守的な機構は，低コストでヌルメッセージを十分に削減できる方法がない限り，論理シミュレーションには不適切と考えられる†1．

一方，タイムホイル機構については，使用プロセッサ数が8以下の場合，タイムワープ機構の場合よりも高い性能を示した．そして，16プロセッサ使用時で約25Kイベント/秒という最高値を示している．しかしながら，32プロセッサ以上ではかえって性能が低下している．この理由としては，(1)時刻を進める際のタイムホイル間の大域的な同期のコストが高いこと，(2)使用プロセッサすべてを効率的に稼働させるだけの十分な並列性を抽出できないこと，の2点が考えられる．タイムホイル機構は，使用プロセッサ数が少なく，かつシミュレーション時間を粗く離散化する場合

†1 共有メモリマシン上では，問合せによりヌルメッセージを生成することでその発生を抑えることに成功した例が報告されている[24, 29]．この方法を分散メモリマシン向きに改良することが，解決策の一つとして考えられる．

表 3.3.4 対象回路 (PIM/m にて)

回路	s38584	s38417	s35932	s15850	s13207
ゲート数	27965	31995	26433	13354	11965

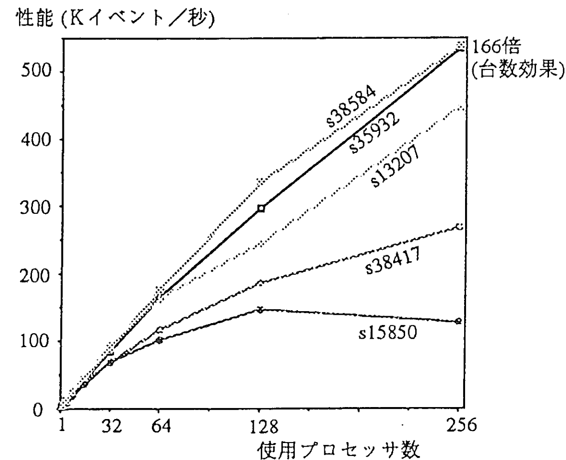


図 3.3.17 性能評価 (PIM/m にて)

のみ，良好な性能が期待できる．

C. PIM/m での性能

いままで Multi-PSI 上での論理シミュレータの評価について述べてきた．Multi-PSI は，第五世代コンピュータプロジェクトの最終目標の一つである並列推論マシン PIM の実験機である．プロジェクトの最終段階になり 5 種類の PIM が完成したため，そのうちの一つ PIM/m 上で論理シミュレータの性能評価を行なった．

PIM/m は Multi-PSI とは異なり，256 プロセッサをもつ．通常，より多くのプロセッサ数をもつ計算機を用いる場合，それに見合った大きな問題を解かせなければ，高い処理効率を維持できない．このため，筆者らは，さらにたくさんのゲートから構成される回路を対象回路としてシミュレーションを行なった．

実験回路の大きさを表 3.3.4に示す．

これらのシミュレーションの結果，得られた性能と使用プロセッサ数の関係は図 3.3.17のようになった．

最もよい場合で，256 プロセッサを使用し，500K イベント/秒以上の性能を達成した．これは，ソフトウェアシミュレータとしては良好な値であると考えている．また，このときの台数効果についても 166 倍と良好であった．

3.3.7 プログラム開発について

最後に，プログラム開発状況，特にどのようにデバッグを進めたかを中心に触れておきたい．

一般に，並列プログラムはデバッグが大変であるといわれている．これは，並列プログラムは実行するたびに実行結果が違い，バグの追跡が非常に難しくなるためである．KL1 言語は，データフロー言語であるため，メッセージの送受信に関する基本的な同期処理については，プログラマは陽に記述する必要がない．したがって，従来の言語によるプログラミングに比べ，バグの混入する機会はかなり減る．

しかしながら，KL1 では非決定処理，すなわち複数の選択肢のどちらが選ばれてもよいような処理も自然に書ける．このことは，割込み処理などが簡単に記述できるなど，KL1 の記述力を高めている．しかし同時に，タイミングに依存したバグが混入する可能性も大きくしている．

タイムワープによる論理シミュレーションでは，この非決定処理を多用している．たとえば，複数入力をもつゲートでは，それぞれの入力にメッセージがやってくる可能性があり，かつ，ときどき GVT 更新処理に関するメッセージも受信する．これらがどのタイミングで到着しても，正しく処理を行なうようにプログラムされなければならない．

もちろん「バグを入れないように細心の注意を払って設計/プログラミングする」ことが最も重要であることは間違いないが，どんなに注意深くプログラミングしても混入するのがバグである．したがって，いかに早くバグを除去できるかということが，プログラミング期間を大きく左右する．

論理シミュレータが開発されていた時点では，KL1 言語で記述したプログラムの開発/実行環境は，以下の三つであった．

1: PDSS
Unix マシン上で動く KL1 言語実行システム．シングルプロセッサ環境のみ提供．以下のシステムとは，ソースレベルにおいても，互換性が保たれていない部分がある．

2: Pseudo Multi-PSI
PSI ワークステーション上の KL1 言語実行システム．Multi-PSI 上でのシステムとオブジェクトレベルでの互換性がある．時分割による擬似マルチプロセッサ環境を提供．

3: Multi-PSI
実際の並列処理システム

一般には，以下のような順でプログラム開発が行なわれている．

1) PSI 上でコーディング
2) Pseudo Multi-PSI でデバッグ
3) 実機 (Multi-PSI/PIM) 上で動作/デバッグ

通常，2) の段階でほとんどのバグが除去される．3) の段階で発見されるバグは，データサイズやタイミングに起因するものである．

しかし，論理シミュレーションの開発においては，以下の手順に従った．

1) Unix マシン上でコーディング
2) PDSS でデバッグ
3) Pseudo Multi-PSI でデバッグ[†1]
4) 実機 (Multi-PSI/PIM) 上で動作/デバッグ

PDSS では，デバッグツールであるトレーサを用いても，実行順序は変わらない．すなわち，再現性がある．これに対し，Pseudo Multi-PSI や実機 Multi-PSI では，トレーサを用いると，実行順が変化する．筆者は，PDSS の提供する再現性を活用することで，デバッグ期間の短縮を図った．

プログラムの動作に誤りが発見された場合，同じ条件で何度か試行を繰り返しながら徐々にバグの存在部分を特定していくのが通常であろう[†2]．しかし，実行のたびに動作状況が変化することでバグの影響の現われ方が異なってくれば，この作業が大変難しくなる．一方，再現性があれば，比較的短期間にバグを特定できる．本シミュレータの開発期間が短かった大きな理由の一つは，PDSS を活用してデバッグを行なったことであると筆者は確信している．

なお，PDSS はシングルプロセッサ環境のみを提供するシステムであるが，複数のゲート群をそれぞれ管理するプロセスをうまくスイッチングすることで，擬似並列環境を実現した．さらに，PDSS⇔Multi-PSI 間のソースレベル互換性のない部分については，互換性を提供するプログラムを作成した．

3.3.8 むすび

本章では，ICOT で開発した並列論理シミュレータについて解説した．

ICOT の論理シミュレータは，時刻管理機構としてタイムワープ機構を採用していることを特徴としている．このシミュレータの性能評価を行なった結果，タイムワープ機構は，対象となる問題に含まれる並列性を効率よく抽出し，高い台数効果を達成するとともに，その結果得られた絶対性能も，ソフトウェアシミュレータとしては良好なものであることが確認できた．具体的には，PIM/m 上で 256 プロセッサを用いて 166 倍の台数効果と，500K イベント/秒を越える性能を達成した．

さらに，ヌルメッセージを用いた保守的な機構，およびタイムホイル機構による並列論理シミュレーションの実験も行ない，タイムワープ機構との性能比較を行なった．保守的な機構は，ヌルメッセージ削減機構を組み込んだにもかかわらず，なお多量のヌルメッセージが発生し，その結果，性能はタイムワープ機構に大きく劣った．また，タイムホイル機構は，使用 PE 数が少ない場合にはタイムワープ機構を上回る性能を示したが，使用 PE 数が増加するにつれ，性能が低下した．以上の比較から，タイムワープ機構は，分散メモリマシン上での並列論理シミュレーションの時刻管理機構として最も有効な方法であることが確認できた．

本シミュレータは，ソフトウェアシミュレータという観点からは良好な絶対性能を示したといえるが，256 プロセッサを用いたことを考えると，決して十分ではない．これは，基本的に 1 プロセッサでのシミュレー

†1 PDSS とのソースレベルでの互換性がない部分についてはバグが発生する可能性があるため，このステップが必要である．
†2 もちろん，もし可能であれば全履歴を保存し，試行は 1 回のみ行なうという方法も考えられる．しかし，多すぎる情報は，プログラマのやる気を失わせがちである．

タ性能が低いためであり，さらにいえば，KL1 言語のもつオーバヘッドが大きいためである．

KL1 言語は，あらゆる粒度の並列性が簡単に記述できる非常に手軽な並列プログラミング言語であるが，それゆえのオーバヘッドが存在している．論理シミュレーションのようにきわめて高速処理が要求される場合，並列処理部分のみを KL1 で記述する一方，逐次処理部分は C 言語などで記述し，これらを結合するべきであろう．これによって，超高速論理シミュレータの実現が期待できる．

3.4 遺伝子情報処理

3.4.1 はじめに

ここでは，遺伝子情報処理の主要なテーマの一つであるタンパク質の配列解析を行なう二つの並列システムについて紹介しよう．最初にマルチプルアライメントと呼ばれる配列解析はどのようなテーマなのかについて簡単に解説する．その後，マルチプルアライメントを解決する二つのシステム（ダイナミックプログラミング法によるシステムとシミュレーテッドアニーリング法によるシステム）を，それぞれの並列システム構築経験を交えて，紹介したいと思う．

3.4.2 配列解析とは

A. はじめに

ある日，まったく畑違いの分子生物学の勉強を始めることになった．並列マシンが実際に役立つことを実証するためには，ある程度大規模でまた実際的な領域を扱う必要がある．遺伝子情報処理と呼ばれる分子生物学と計算機科学の新しい境界研究領域は，この目的に向いているように思われた．

とはいうものの，筆者らが遺伝子情報処理の研究を始めた頃は，分子生物学などまったくの門外漢だった．のみならず，並列プログラミングの方法論がよくわかっているわけでもなかった．

それゆえ，分子生物学の勉強をしながら，並列プログラミング方法論の研究を行なわなければならなかった．分子生物学的に意味があり，かつ並列処理としてもおもしろい，そのような研究対象を探す必要があった．かくして，筆者らの冒険は始まった．

B. タンパク質構造とアミノ酸配列

生命現象のなかで重要な役割をはたしている物質にタンパク質がある．タンパク質の研究は，分子生物学の中心的な課題の一つといってよい．タンパク質は非常に複雑な構造をしており，この構造が，タンパク質の機能を生み出している．この構造をいろいろ調べてみることは非常に重要な研究課題であるが，現在のところ，立体構造がよくわかっているタンパク質は数百種類と非常に少ない．

一方，この非常に複雑な立体構造をもつタンパク質も，元をたどれば，アミノ酸という単位が直線状につながったものが，折れたたまってできている．簡単に述べれば，タンパク質には二つの端があり，その端を引っ張ればアミノ酸が連なったまっすぐな 1 本のひもとなる．このアミノ酸のひもは，アミノ酸配列といわれる．タンパク質の構造は，このアミノ酸配列が決まれば一意に決定される．しかし，そこにどのようなルールが存在するのかは，よくわかっていない．配列と構造の関係を明らかにすることは重要な研究テーマであるが，まだアミノ配列からタンパク質の完全な立体構造を予測する方法は存在しない．とはいうものの，何はともあれアミノ酸配列のなかにタンパク質構造ひいては機能に関する情報がすべて入っているという事実は非常に重要である．

さて，アミノ酸はそれぞれ慣習でアルファベット 1 文字で表わされることになっているので，それぞれのタンパク質は文字列として表現できることになる．実はこのタンパク質のアミノ酸配列だけを知ることは比較的容易であり，数万種類のタンパク質のアミノ酸配列が判明している．これは，立体構造がよくわかっているタンパク質が少ないことと対照的である．

筆者らは，まず，このタンパク質のアミノ酸配列にスポットを当てることにした．

C. マルチプルアライメント

アミノ酸配列は DNA の塩基配列によって指定されているから，タンパク質を作るための情報は，最終的にはすべて DNA のなかに納められている．DNA の塩基配列に変化が起こると，それがアミノ酸配列の変化として反映されるというわけである．これを調べるには，類似のタンパク質の複数のアミノ酸配列を比較する方法が有効である．この配列の比較技術のなかで，重要な


```
M-MULV    ---LLDF--LHQLTHLSFSKMKALLERSHSPYYMLNRDRTLKNITETCKACAQ
HTLV      VLQLSPA-ELHSFTHCG---QTALTLQGATT-----TE--ASNILRSCHACRG
RSV       AYPLREAKDLHTALHIG---PRALSKACNIS-----MQ-QAREVVQTCPHCNS
             ^     ^^   ^       ^^                       ^  ^
```

図 3.4.1 マルチプルアライメントの例

ものがマルチプルアライメントと呼ばれるものである．

図 3.4.1 は，マルチプルアライメントの一例である．マルチプルというのは複数本（3 本以上）の配列を比較することを意味する．マルチプルアライメントは，複数の配列間で同じアミノ酸，あるいは似ている性質のアミノ酸をなるべく縦に整列させる処理である．ところどころにハイフン "-" が挿入されているが，これはギャップと呼ばれる．これらのギャップを挿入することにより，性質の近いアミノ酸をなるべく縦に整列させるのである．このギャップの挿入は，進化的には，あるアミノ酸が DNA レベルで欠失したことを表わしている．マルチプルアライメントでは，同じ列に同じ文字が縦に並んでいるのが一番よいが，異なる文字でも，それらが表わすアミノ酸の性質が似ていれば同じ列に置くことを許容する．これは進化的には，あるアミノ酸が DNA レベルで他のアミノ酸に置き換ったことを表わしている．

さて，マルチプルアライメントを行なうと何がわかるのだろうか．まず進化がどのように行なわれたかを推定できる．アライメント結果から配列間のアミノ酸の差異を数えることができるが，この差異を用いて，タンパク質ごとに何回アミノ酸の置換が起こったかを算出することができる．アミノ酸の置換速度はヒトでもウマでも「タンパク質の種類が同じなら，ほぼ一定」であることが知られている．この事実を用いて，種の間の進化的距離を決定でき，進化系統樹を作成できる．次にアライメント結果から，タンパク質のどの部分が重要かという推測が可能である．たとえば図 3.4.1 の例では，アミノ酸がばらばらの列やギャップが多い列もあればアミノ酸がほとんど同じである列もある．ほとんどのアミノ酸が同じであるような列はアミノ酸が置換しにくく保存性が高いことを示している．ところが一方，突然変異をはじめとする遺伝子上の変化はどこでも等確率で起こると考えられる．ではいったい，なぜこのように保存性の高い部分と低い部分が存在するのだろうか．

その理由は，以下のように考えられる．遺伝子上の変化はどこでも等確率で起こるが，タンパク質には重要な部位とそうでない部位が存在する．もし重要な部位のアミノ酸に変化が起こると，普通そのタンパク質は著しいダメージを受ける．その結果，その変化を受けた個体は生存上不利になり，一般に死んでしまう．そのため，重要な部位に起こった変化は種のなかに残らない．一方，あまり重要でないアミノ酸に変化が起こっても，普通そのタンパク質は以前と同様に機能する．その結果，その個体は生き続けて子供を作り，その変化が種のなかに固定される可能性が高い．つまり「保存性の高い部分＝重要な部分」と考えられる．この考え方を用いて構造が未知なタンパク質でも，機能的，構造的に重要な部位を，配列だけから予測できるわけである．

D. 最適化問題としてのマルチプルアライメント

さて，マルチプルアライメントは，これまで生物学者がエディタなどを用いて手作業で行なうことが多かった．もし，「アミノ酸の類似性」と「ギャップの入りにくさ」に対し，生物学的，物理化学的観点から得点を割り当てることができれば，アライメント全体の評価値を定義することができる．評価値が定義できれば，次はこの評価値を最適化する，つまり最もよくする手法を考えることになる．マルチプルアライメントはいわゆる「組合せ最適化問題」として定式化できる．この定式化に従って筆者らが開発した並列 3 次元ダイナミックプログラミング法と，並列シミュレーテッドアニーリング法という二つのプログラムについて以下で解説する．アミノ酸の類似性については，Dayhoff という人が提案した類似性の表 [30] を用いて，またギャップコストに関しては，パラメータとして試行錯誤ができるようにプログラムを作成した．

3.4.3 マルチプルアライメント (1)：3 次元ダイナミックプログラミング法の並列実行

A. はじめに

まずこの章では，“並列 3 次元ダイナミックプログラミングを用いたマルチプルアライメントシステム”

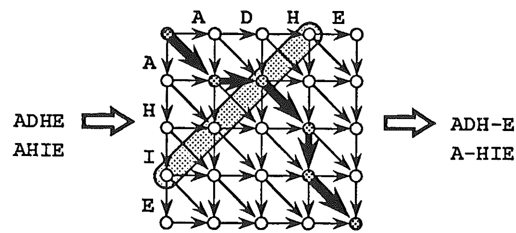


図 3.4.2　2 次元 DP マッチングによるアライメント

の紹介と，それにより得られた並列プログラミング体験について述べようと思う．この並列ソフトウェアを並列プログラミングという観点から見ると，次の章に紹介するシミュレーテッドアニーリングの並列化のような比較的粒度の粗い負荷分散とは異なり，たくさんの小粒度プロセスの負荷分散を効率よく実現したという点で興味深い．さらにその実行の挙動は複雑なため，モデルを作っての解析も試みた．

B.　ダイナミックプログラミングとアライメント

a.　ダイナミックプログラミングによるアライメント

ダイナミックプログラミング（以下，DP と略す）は段階的に決定を行なう特徴をもつ，最適化問題を解くためのアルゴリズムの一つである．一般に，コスト最小のものを選ぶという解法で解いた場合，指数オーダの計算量がかかるような問題も，この DP の手法を用いると，各段階ごとにそこまでの部分的な最適解を決定することが可能なため，多項式オーダの計算量で解を得ることができるのである．

アミノ酸配列のアライメントは，この DP を用いて行なうことができる [31]．簡単のために，配列 2 本のアライメントについて DP の概念的説明を行なう．図 3.4.2 を例に用いて説明してみよう．たとえば，ADHE，AHIE という二つの配列をアライメントする場合，この二つの配列を図 3.4.2 のような 2 次元のネットワークの辺に対応させる．各アーク（矢）には，コストが割り振られる．斜め方向のアークは，そのアークの位置に対応する二つのアミノ酸の類似度がコストとして割り振られる．この類似度には前述の Dayhoff マトリクスを用いている．また縦および横方向のアークはギャップに対応し，ギャップを挿入するときのコストが割り振られる．こうしてすべてのアークにコストが割り当てられる．この結果，ネットワーク上始点のノードから終点のノードに至るさまざまな経路は，さまざまなアライメントに対応するようになり，コストとしてアライメントの評価値をもつことになる．

このように問題を定式化すると，最適なアライメントを求めることは，このネットワーク上のコスト最小の経路を求めることに対応する．図 3.4.2 の例では，太いアークで表わされた経路がコスト最小となる．この太いアークで表わされた経路を順に見ていくと，最初のアークは A と A が同じカラムに並ぶことを意味し，次の横向きのアークは D に対応するもう片方のアミノ酸がない（つまりギャップが対応する）ことを意味する．以下，次のカラムには H には H が並び，…という具合に読んでいくことができるのである．その結果として，図 3.4.2 の右側にあるアライメントを得ることができたのある．

コスト最小の経路は，左上の端から右下の端に向かって（逆でも可能），各ノードに至る最短経路を段階的に決定していくことにより求めることができる．各段階は，図 3.4.2 の右上がりの細長い楕円の領域に存在するノード群に対応する．段階的に各ノードへ至る最短経路を求めていくと，いったん求まった部分的最短経路はもはや変更されることがない．それゆえ，この部分的最短経路を用いて次の段階の計算を行なうことができるのである．ここで述べている“段階”という言葉は，DP の一つのキーワードであるので，心に留めておいていただきたい．

さて，このように N本の配列をアライメントするには，このような N次元のネットワークにおける DP を行なえばよいわけだが，実際には配列の長さの N乗に比例する計算量となる．

b.　DP3 次元化の意義

このように N次元 DP を用いれば，N本の配列の最適なアライメントを得ることができる．しかし，この手法が必要とする計算量は多く，配列が 3 本以上では，これまで近似的にしか使用されていなかった [32, 33]．通常は配列 2 本のアライメントを繰り返して，マルチプルアライメントを行なう方法が試みられている．ところが，それでは精度が十分でなく，難しいところはもっぱら熟練した生物学者の勘に頼っている状況である．そこで，筆者らは 3 次元 DP により 3 本の完全なアライメントを求めることにターゲットを定めた．これを実現することにより，従来の手法により得られた結果の検証や繰り返し得られた解を用いて，より質の高いアライメントを短時間のうちに得ることができるのである．こうして，アライメントにおいて，DP を 2 次元から 3 次元に一つ次元を上げたことにより，単に 3 本の配列を一度にアライメントすることが可能になっただけではなく，さらにそれ以上の非常に意義のある成果が得られたのである [34]．

C.　KL1 による 3 次元 DP の並列実装

3 次元ＤＰは図 3.4.3 のような 3 次元のネットワークとして表現できる．

さて，KL1 プログラミングにおける並列の実行単位はプロセスである．したがって，並列実行の主体である各ノードを KL1 のプロセスで実現することにした．この各段階において，計算を行なうプロセスを段階が一つ進むごとに生成するのは効率が悪い．そこで 3 次元 DP におけるノードを KL1 プロセスに，アークを KL1 プロセスの通信路に，それぞれ対応させることを考えた．このように対応づけを行なうと，3 次元 DP のネットワークをそのまま反映した KL1 のプロセスネットワークを，あらかじめ生成してしまうことができる．このようにプロセスネットワークを構築すると，各プロセスが隣接するプロセスとメッセージの授受を行なうことによって全体の計算が進んでいくことになる．上記の方法を用いると，3 次元のプロセスネットワークを 3 次元メッシュに分割してプロセッサに割り当てることにより，並列実行を行なうことが可能である．各段階の処理は，波面状にネットワークのなかを進んでいくのである．

図 3.4.3 では縦，横，高さ，各方向にそれぞれ 4 分割されて，64 個のプロセッサにネットワークを割り当てた例を示す．各プロセッサにはノードに対応する KL1 プロセスが多数割りつけられている．また，この割当て方法は，縦，横，高さ方向の切断数を容易に変えられる．

筆者らは，この計算量の多い 3 次元 DP を，ICOT の並列推論マシン上で並列実行するプログラムを開発した [35]．プロセッサ 64 台で並列実行させたところ，1 台に比べ約 37 倍の高速化が実現できた．すなわち，計算量が多く，いままで本格的に扱われてこなかった 3 本の最適なアライメントを，並列処理により現実的な時間内で実行可能にしたのである．

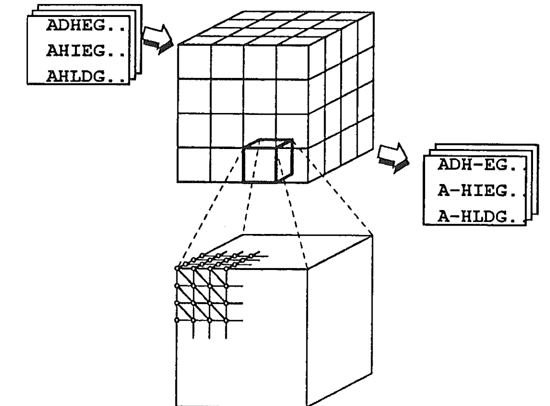


図 3.4.3　3 次元 DP によるアライメントとプロセッサへの割当て

D.　PIM における 3 次元 DP の並列実行とそのからくり

a.　並列プログラムの難しさ

並列処理システムが構築できたあとには，実際に並列処理が理論どおりに行なわれているか，行なわれていないとしたらどこに問題があるかを検討するために，実験を行なって性能特性解析をすることが重要である．それによって得られた処理モデルは，システムの並列処理性能の向上の手掛りになるだけでなく，他の類似の並列システムを実装するうえでの貴重な資料となる．この問題においても，KL1 を用いることで非常に素直に並列プログラムを記述することができた．しかし，実際に実行させてみると，予想していた実行の振舞いとは異なる，意外な一面も見せつけられたのであった．

一般に，逐次プログラミング言語から発展してできた並列プログラミング言語により，複雑な並列プログラムを実現することは非常に難しい．それに対して KL1 は，並列処理を行なうために生まれてきた生粋の並列言語である．そのため非常に容易に効率のよい並列プログラムを実現することができる．しかしその半面，逐次に処理が行なわれる部分は，意識して記述しない限り，処理系まかせになって，こちらの思いもよらない動作をしてしまうことがあり得ることを肝に命じておかなければならないのも事実である．実際，筆者らのプログラムでは強い実行の指向性が現われた．その結果，各軸方向に進む処理の速さの間に差が生じ，

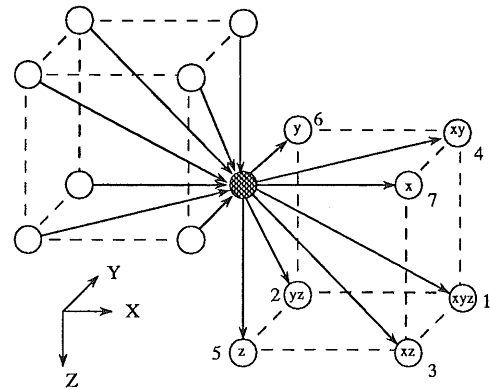


図 3.4.4　プロセスのメッセージ送受信

当初一様に進むと思われていた実行の波面は予想とは違ったかたちで現われたのだった．このことは，全体の実行効率に非常に影響を与える要因となり得るのである．

以下，並列プログラミング経験として非常に興味深いと思われるので，実際に，この 3 次元 DP がどうして予想外の振舞いをしたのかを実行のモデル化をして，検証してみたことについて述べる．いささか複雑な話になることをお許しいただきたい．

b.　プログラムの実行の様子とそのモデル化

まず，筆者らは実行の指向性について考えてみた．図 3.4.4 のような，プロセスネットワーク上に仮想的に座標軸を考えることにする．各プロセスは一般に，自分の直前の七つのプロセスからの情報をすべて受け取った時点で自分の計算を行ない，次の七つのプロセスに情報を送る．あるプロセスにおいて次のプロセスに複数のメッセージを送ることは，筆者らのマシンでは 1 命令として実行されるため，1 プロセッサ内では必ず逐次に実行される．図 3.4.4 において，xyz，yz，xz，xy のプロセスはすべてのメッセージが揃っていないため，すぐには動けない．筆者らのプログラムにおけるメッセージ送信の定義では，意識したわけではないのだが，z の位置のプロセスに実行が移るようにプログラミングされていたので，z が最初に実行されたのであった．

この考えを進めると，1 プロセッサ内では Z 軸方向に優先的に処理が進むと予想できた．さらにこの考えを発展させていくと，ある一つのプロセッサが担当する部分のネットワークにおいては，図 3.4.5 のような実行順序となることが予想される．

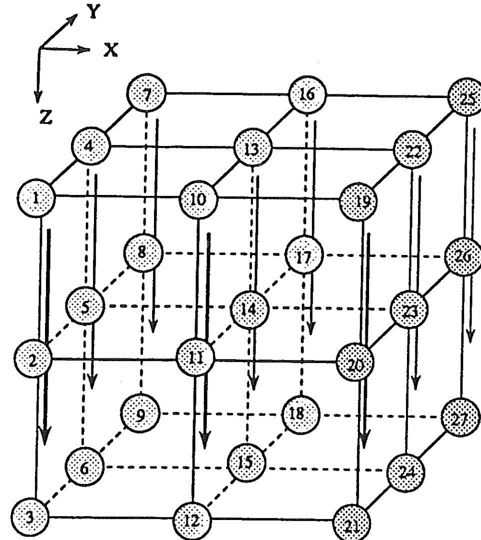


図 3.4.5　プロセッサ内での実行順序

さらに，この予想をプロセスネットワーク全体に広げていくと，処理の流れには指向性があり，並列実行可能な波面が各軸に対して均等には進まないことがわかる．図 3.4.6 に，これまでの考察をもとに，矢印 1 本を 1 プロセッサ内での Z 軸方向への一連の処理として，ネットワーク全体での処理の流れをイメージ化してみた．

並列処理に要する時間を把握するには，全体の処理時間を決定する最も遅い処理に注目する必要がある．そういう処理の流れに着目できれば，全体の処理時間がどのようなものから構成されるかを推測することができる．

このように，実際にプロセスネットワーク上をどのように処理が進んでいくかを予測したものをもとに，メッセージがプロセッサをわたる部分でのプロセッサ間通信オーバヘッドを考慮に入れて，全体の実行時間を算出する式を導き，このプログラムの実行モデルとした．ここで，プロセッサ間での通信コストの考慮は，モデル化を行なう上での「みそ」となっている．ここでは，このモデル化について詳細を述べるスペースがないので，興味のある方は文献 [36] を参照されたい．

c.　実行モデルの検証

64 プロセッサで実行した場合の，X 軸，Y 軸，Z 軸，各方向に沿った分割の違いによるパフォーマンスの差異について検討してみた．(X,Y,Z) = (4,4,4) とか，(1,8,8) とかさまざまな異なった分割に対して実行

を行ない，実行時間 T の計測を行なった．分割しない実行時間 $T1$ も計測した．先に述べたモデルでは，分割 A について，並列実行に要する時間 T は，逐次実行時間 $T1$ と一つのメッセージのプロセッサ間通信コスト C を用いて，$T = T1\mathbf{P}(A) + C\mathbf{Q}(A)$ の形で表わされているため，$T1$ および T から一定値 C を求めることが可能である．それぞれの場合ごとに，C を算出したところ，1.032 ± 0.364 ms となり，ほぼ一定と見られる値が得られた．これは，モデルが実際の並列実行に近いものであることを示している．以下，C の値は 1.0 として解析を行なった．一つの軸方向の分割を 4 として固定したときに，他の軸方向の分割を変化させたものの実測値とモデルから得た値の三つのグラフを図 3.4.7 に示す．モデルの $C = 1$ は，C を 1.0ms として，通信オーバヘッドを考慮したグラフ，$C = 0$ は通信オーバヘッドを考えない場合のグラフである．

図 3.4.7 からわかるように，モデルから得られる値と実測値は，ある程度定量的に一致しているといえる．それぞれのグラフに対して解説を加える．Y 軸方向の分割を固定した場合，X 軸方向を細かく分割するほど実行時間は実測，モデルともに悪くなっている．Z 軸方向の分割を固定した場合も，同じく X 軸方向の分割が好ましくないことで一致を得ている．そして，X 軸方向の分割を固定した場合であるが，この場合は残りの Y 軸と Z 軸についてトレードオフ点が存在し，かつ，それは Y と Z に均等に分割したときで，実測とモデルが一致している．

このようなモデルの特徴から，全体のプロセッサ数を一定にしたとき，最も効果的な分割は X 軸方向には分割をせずに，Y，Z 軸方向を均等に分割したときだと推測できる．そして，実験においても，64 プロセッサを使用して最も速く解を得られたのは，(X,Y,Z) = (1,8,8) の分割でマッピングを行なった場合で，まさにモデルから得られた推測に一致していた．

d.　最後に

一般に，プロセッサ稼働率と通信オーバヘッドはトレードオフの関係にある．つまりプロセッサ稼働率を上げることのみを考えれば，通信オーバヘッドの影響が並列効果を打ち消すほど強く出てしまうし，一方，通信オーバヘッドをなくすことのみに専念すると稼働率が下がってしまって，結局，実行効率は全体として悪くなってしまう．そのため，その間で最も実行効率

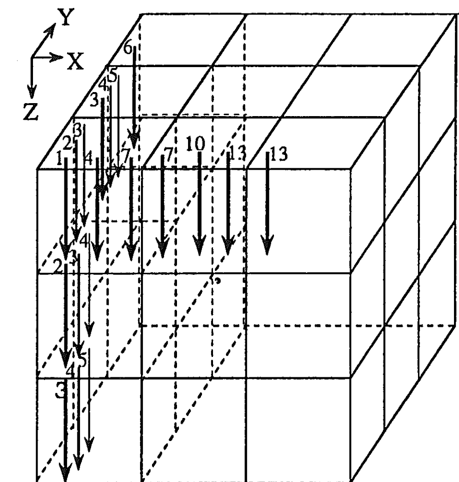


図 3.4.6　全体の実行の流れ

のよいバランスのとれた点を見つけることが重要になる．

本プログラムの解析で用いたモデル，およびそれから導かれる式は，プログラムの特性をよく表現しており，トレードオフ点を予測するという点でも大きな意味をもつものであった．特に通信オーバヘッドがないときの理想的理論値と，通信オーバヘッドの理論値が，分離された形で表現できたことは興味深い．図 3.4.7 からも，PIM のようなマシンでは通信オーバヘッドまで含めたモデル構築を行なうことが，今後ますます重要になると考えられる．

3.4.4　マルチプルアライメント (2)：シミュレーテッドアニーリング法とその並列化

A.　はじめに

筆者らは，もう一つの異なった手法によりアライメントの自動化を試みている．その手法はシミュレーテッドアニーリングという手法であり，さまざまな組合せ問題に適用可能な汎用の解決手法である．これは DP とは異なり，3 本以上のアミノ酸配列を一度にアライメントすることが可能である．ただし，実行には長時間を要してしまう．

B.　シミュレーテッドアニーリング法とマルチプルアライメント

a.　シミュレーテッドアニーリングとは

シミュレーテッドアニーリング（以下，SA と略す）

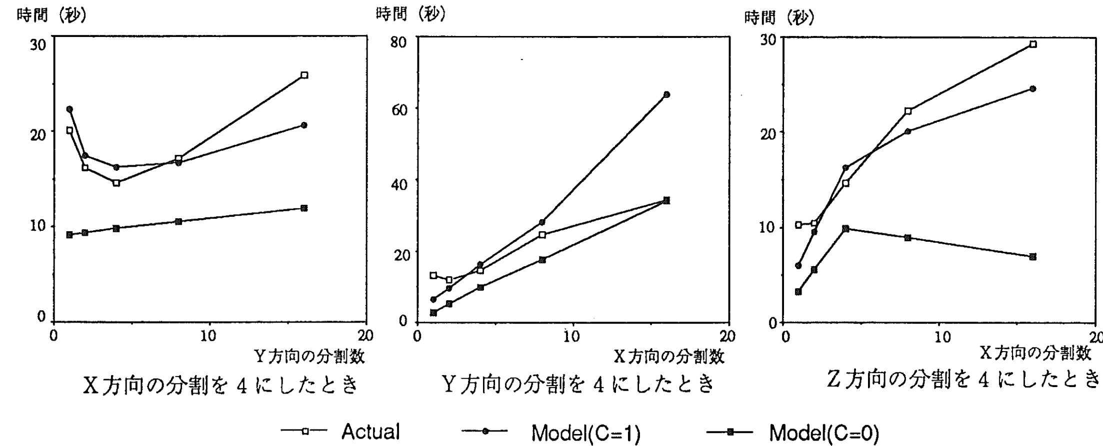


図 3.4.7 分割による性能の差

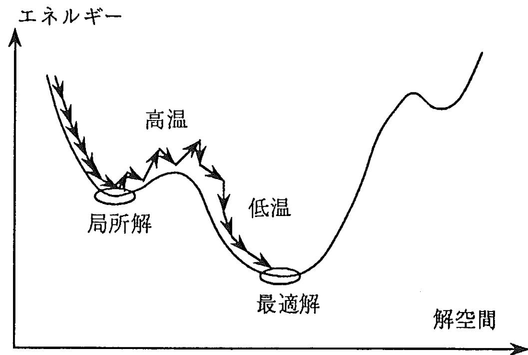


図 3.4.8 シミュレーテッドアニーリング

のアルゴリズムは，組合せ的に生成される解空間において，定義されたコスト（エネルギーと呼ばれる）最小の解を探索する確率的なアルゴリズムである．この手法の長所は，温度と呼ばれるパラメータを導入することにより，解を少しずつ変換しても，解空間上，比較的近傍にある局所的にエネルギー最小な点に陥らずに，全体でエネルギー最小の点にたどり着くことを可能にするという点である [37].

元来，アニーリングとは，物理系の焼きなまし過程を意味する．つまり，ある物質を高温から非常にゆっくりと温度を下げることにより，結果として非常に安定な物質が得られる過程を指している．シミュレーテッドアニーリングとは，この焼きなまし過程を模擬したアルゴリズムである．これには，探索許容度を示すパラメータ（温度パラメータと呼ばれる）が導入されている (図 3.4.8)．アニーリングの初期においては，温度パラメータは大きな値に設定され，コスト的にはよくないような解への変換も許される．そして，あらかじめ設定しておいたスケジュールに従って温度パラメータは下げられていく．アニーリングの後期においては，温度パラメータの値は非常に小さくなっており，この頃にはコストが大きくなるような，よくない変換はほとんど受けつけられなくなる．

具体的にアルゴリズムを説明すると，初期解 X_0 から順に次のように解系列を生成していき，徐々に最適解に近い解を得ていく．まず，ある解 X_n にランダムな微小変形を行なうことで次の解の候補 Y_n を作る．最小化を目的とするエネルギー関数を $\mathbf{E}$ とすると，エネルギー値の変化は$\Delta E = \mathbf{E}(Y_n) - \mathbf{E}(X_n)$ となる．$\Delta E \leq 0$ ならば，よい変形であるので無条件に $X_{n+1} = Y_n$ とし，また$\Delta E > 0$ ならば，解の質が悪化するような変形が行なわれたこと意味する．そのため，確率値として $P = \exp(\frac{-\Delta E}{T_n})$ を採用し，温度パラメータ T_n に依存させて，次の解を決定する．つまり，確率 P で $X_{n+1} = Y_n$ とし，確率 $(1 - P)$ で $X_{n+1} = X_n$ とする．このオペレーションをエネルギー値が収束するまで，多数回繰り返す．

ここで温度パラメータ列 $\{T_n\}$ は温度スケジュールと呼ばれ，それを適切に設定すれば，十分な時間ののち最適解を求められることが理論的には保証されている．しかし現実には，限られた時間内に準最適な解を求める場合が多い．そのための適切な温度スケジュールは扱う問題ごとに異なり，それを設計することは少々骨がおれる課題である．あとに述べる並列化は，この温度スケジュールの設計を不要にするものである．

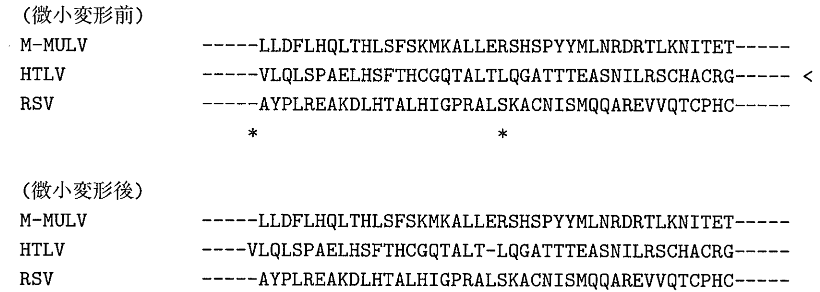


図 3.4.9 微小変形例

b. マルチプルアライメントへの適用

マルチプルアライメントに SA を適用するためには，ある解の状態から近隣の状態へと移るオペレーションである微小変形と，各状態における評価尺度にあたるエネルギー関数を定義する必要がある．

マルチプルアライメントの結果は内側に不定数のギャップを含むので，微小変形において，ギャップをどのように扱うかが鍵となる．そこで，配列の頭部や尾部に，あらかじめ十分な数のギャップを付加する方法をとった [38]．そして，状態に対する微小変形は次のように定義する．複数の配列のうちのある1本の配列に対して，任意のギャップと任意のカラム位置をそれぞれランダムに選択し，選択されたギャップを選択されたカラム位置に移動させる．そして，間の部分の配列を，移動したギャップがあったほうへ1カラム分移動する．

図 3.4.9 の例では，HTLV の配列に注目して，*印のある二つのカラムが選ばれた場合のオペレーションである．この微小変形は，相同性の高い配列群のマルチプルアライメントは比較的うまく行なえるのに対し，ギャップが固まりで入るような相同性の低い配列間のマルチプルアライメントは苦手であった．それは，ギャップの固まりが配列内部に形成されにくいためである．その欠点を補うために，ギャップを長方形の固まりで動かすブロックオペレーション [39] を導入した．ブロックオペレーションは，あるギャップをランダムに選んだならば，そのギャップの横方向や縦方向にギャップの連なりを探し，矩形ブロックの単位でギャップ群を移動させる微小変形である．

マルチプルアライメントにシミュレーテッドアニーリングを適用した場合，取り扱われる全配列にわたって同時に評価を行なうことができるという利点がある．こうした評価の値を各状態に対して与えるのがエネルギー関数である．現在エネルギー関数として，ある配列ペアにおいて，各カラムにおけるアミノ酸ペアについて Dayhoff マトリクスの値を総和し，それをすべての配列ペアについて行ない，合計したものを使用している．

c. 並列化による温度スケジューリング問題の回避

SA を並列に行なうにはいくつかの方法がある．最も単純な方法は，通常の逐次的に動く SA を，利用可能な要素プロセッサ（PE）の数だけ独立に（異なる乱数で）行ない，そのうちの最もよい解を選ぶものである（単純並列 SA）．それに対して，本システムで用いた方式 [40] は，各 PE ごとに問題に応じた高温から低温までの異なる温度を割り当てさえすれば，事前にそれ以上の完全な温度スケジューリングを行なっておく必要がない（温度並列 SA）利点をもつ (図 3.4.10)．そのため，適切な温度スケジュールの吟味されていない段階においては，温度並列 SA は単純並列 SA に比べ，非常に有効となる．また，温度並列 SA は，そのときどきにおける最良の解が刻々と得られるという利点も併せもっている．この温度並列の方式を，以下に述べる．

各 PE ごとに初期解を与え，それぞれの PE ごとに高温から低温までの異なる温度パラメータを与え，それぞれ一定温度の探索を行なう．そして，ある回数探索を行なうと，隣接する温度を担当している PE の間で，解の交換を確率的に行なう．この解交換は，よい解は必ず低温の PE に移動するようになっており，悪

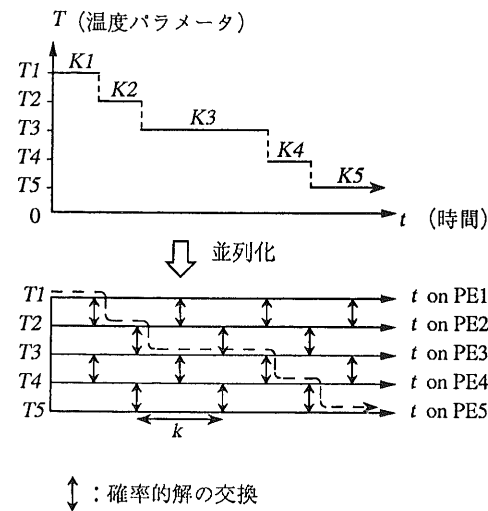


図 3.4.10 温度並列 SA

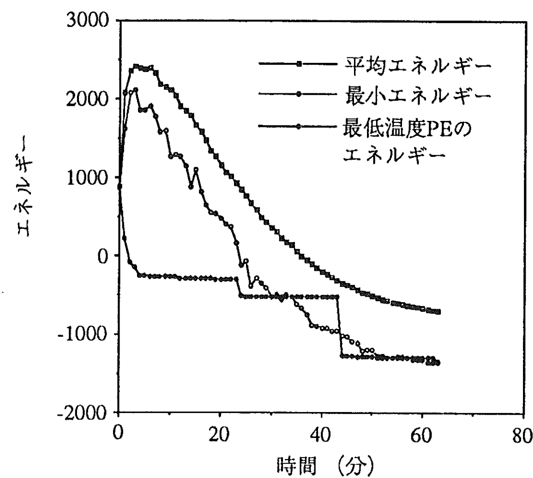


図 3.4.11 アニーリング経過の比較

い解も温度に依存した確率に従って交換されうる．それを適当な頻度で行なうように設定する．これにより，しかるべき時間のあとには，優れた解が最低温の PE に現われる．

この解の交換確率は，温度パラメータ $T1$ において得られた解のエネルギーが $E1$，温度パラメータ $T2$ において得られた解のエネルギーが $E2$ のとき，$\Delta E = E1 - E2$，$\Delta T = T1 - T2$ とおけば，次式で定義される．

$$\mathbf{p}(T1, E1, T2, E2) = \begin{cases} 1 & \textbf{if } \Delta E \cdot \Delta T < 0 \\ \exp(\frac{-\Delta E \cdot \Delta T}{T1 \cdot T2}) & \textbf{otherwise} \end{cases}$$

この関数により得られた確率値に従って，解を実際に交換するか，交換を見送るかの決定を下せば，各温度に対する Boltzmann 分布に従う平衡状態を崩さずに解の交換を行なうことが可能になり，十分に長い時間をかければ最適解が得られることが保証される．

d. KL1 による並列実装と実験結果

以上で述べたようなシステムを並列推論マシン上に構築した．高温から低温まで 63 の異なる温度を，63 台の要素プロセッサ（PE）に割り当てて，温度並列 SA を行なった．また 1 台で逐次 SA を，63 台で単純並列 SA を行なった．そして，温度並列 SA によるエネルギー低下を，時間を追って逐次 SA や単純並列 SA と比べたのが 図 3.4.11 である．図を見ると，二つの並列 SA が，逐次 SA に比べかなりよいエネルギー値を示しているのがわかる．さらに温度並列 SA は，単純並列 SA に比べても，ほとんど常によいエネルギー値を示しているのがわかる．しかし，最終のエネルギー値には両者の差異はほとんどない．これは単純並列 SA の温度スケジュールの設計が，割合によかったためであろう．演算時間についても，温度並列は解交換の手間があるにもかかわらず，この実験の範囲内では，両並列 SA には差異が見られなかった．

また，SA で問題を解く場合に，初期状態として近似解を入れ，中低温からアニーリングすると効果的なことがある．このアニーリングには，その近似解に合わせた温度スケジュールが必要であるが，こうした場合にも，温度並列 SA は温度スケジュールの吟味が不要で気軽に使用できる．このように温度並列方式は，解交換に伴う時間のロスもきわめて小さいうえ，温度スケジュールを問題に応じて設定する必要もなく簡便であるため，総じて単純並列方式よりも有用であった．

3.4.5 むすび

ICOT では，並列推論マシンの開発，並列処理の研究という，計算機分野でも新しいテーマに新しいコンセプトで取り組んできた．そうして，プロジェクトの成果である並列推論マシン PIM や並列論理型言語 KL1，およびその処理系 PIMOS が生まれた．筆者らはこれらの環境があったからこそ，大規模データを扱うことのできる高い処理能力をもったマシンのニーズのある分子生物学分野の応用に取り組んでくることができたのだった．これは，新しい計算機利用分野開拓の敷居を低くしたという大変意味のある成果である．

並列システム構築の立場からいうと，とにかく頭のなかで考えた並列処理のイメージをそのままプロトタイプ化でき，しかもあまり苦労もなくほぼイメージどおりの実行を行なってくれる．これは，KL1 および PIMOS が並列処理におけるユーザの苦労を吸収してくれているからであり，それゆえ，応用システムを研究，開発する者にとって，問題の定式化や並列処理可能な部分の切り出しといった興味深いテーマに専念できる環境が整いつつあるといえ，非常にうれしい傾向である．もちろん，より高性能なシステムに仕上げるためにはプロトタイプシステムからチューニングを行なっていく必要は生じる．そのために 現在も，並列実行の profile をとる機能など，PIMOS が多くのサポートを行なってくれている．さらに，今後このような環境上で研究が行なわれるにつれ，さまざまな意味のある開発事例が蓄積され，よりよい並列システム開発環境へと進化していくことであろう．まさにいま，並列処理は第一歩を歩み出したばかりなのであるが，その一歩は KL1 や PIMOS がきっと正しい方向を指し示してくれているといえるだろう．

さて，こうして ICOT において，生物応用の研究を始めてからいままで，わずか 2 年ほどの期間ではあったが，ここで紹介したものを含め，いくつかの並列応用システムを開発することができ，生物学者にも使っていただけるレベルに達しつつある．ここに至るまでに，「分子生物」という分野に関してずぶの素人であった筆者らは，いろいろな知識や情報を吸収，交換するために，生物分野と計算機分野の融合をはかるコミュニティに参加，推進してきた．当時は発足間もなかったこのコミュニティも少しずつ広がり，両者の話も少しずつではあるがかみ合いつつあるというのが現状である．そのなかで，分子生物分野の方々には親切にさまざまなことを教えていただき，非常に御世話になってきたので，この場を借りて心からの御礼を申し上げておきたいと思う．

このように，計算機のフィールドにおいても，また生物応用という面でも，筆者らは開拓者となることができたのではないかと思う．他の成熟したテーマほど完成された成果には至っていない部分もあるが，すべてにわたって，ゼロからの出発であった．今後，こうして植えられた種子が成長し，これらの芽がいろいろな人に引き継がれ，より大きな木に育っていってくれれば，ICOT のいうところの新しい並列処理の文化も，生物研究に対する計算機サポートも，必ずやより大きな成果という形の花を咲かせていくことであろう．

3.5 法的推論

3.5.1 はじめに

並列推論マシン (PIM) 上の法的推論実験システム HELIC-II について紹介する．法的推論とは，法律分野で専門家が立法や契約や裁判での論争などをするときに行なう推論方式のことである．法的推論は，人工知能の応用として最も古い歴史をもち，しかも，法令文と判例という複数の知識源を利用するという特徴がある．

法的推論は意味処理や高次推論に関するさまざまな技術的課題を内在しており，知識処理の技法の有効性を試す格好のテーマといえる．また，法的推論は大量の判例データベースの検索や利用を必要とするから，本質的に実行時間がかかるので，実用システムの開発には並列処理による高速化が必要とされる分野である．したがって，法的推論システムは並列推論の応用として適している．

まず 3.5.2項では，法的推論の概要を説明し，3.5.3 項では HELIC-II の構成について述べる．そして 3.5.4 項では，刑法での問題解決の実例を紹介し，3.5.5項では，並列化のための工夫について述べる．3.5.6項で開発過程について述べる．

3.5.2 法的推論とは何か

法律家は，訴訟の代理，契約書の作成，損害額の算定，紛争の仲裁，節税の相談など，日常生活のいろいろな問題にかかわっている．そのなかで最も重要な仕事は，裁判において一方の代理人として論争することである．

法律の分野では，法令文という知識の体系ですべての情報が与えられているように見える．法令文は通常，「もし～ならば～である.」の形式をしているので，これをルールや論理式で表現することでルールベースが

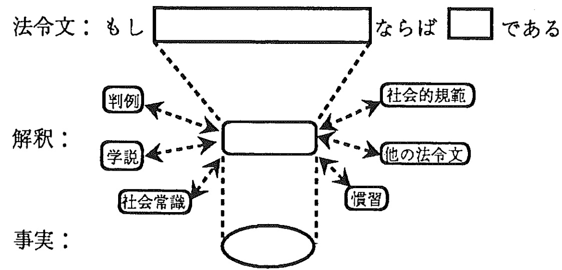


図 **3.5.1** 法律の解釈

できるからである．扱っている事件がこれらのルール（論理式）と同じレベルの用語（述語）で記述されているならば，演繹的にルール（論理式）を適用すれば法的な結論が得られるわけである．たとえば，「人を殺した者は殺人罪である」という法令文があったとしよう．これを

IF 殺人 (A,B) *THEN* 殺人罪 (A)

と表現してみる．新しい事件で

殺人 (太郎, 次郎)

という事実のデータがあれば，上のルールを適用して

殺人罪 (太郎)

という結論が得られる．

しかし，現実には法令文は抽象的な概念を用いて記述されていて，現実の事件の事実データとは用語のレベルに差があるのが普通である．たとえば，「太郎がさいなことで次郎と口論になり，次郎を軽くなぐったところ，次郎は興奮して心臓マヒで急死した」という場合，「口論」「なぐる」「興奮する」「心臓マヒ」「死亡」という概念は出てくるが，これだけで「殺人 (太郎, 次郎)」が成立するかどうかはダイレクトにはいえない．「人を殺す」とはどういうことかを定義するより詳細なルールがないと，演繹的に結論が出せないのであるが，そのようなルールは存在しないのが普通である．

このように法令文 (ルール) と事実とのギャップがあることが，法律のルールにおける特徴である．これは，少ない数のルールで世の中のいろいろな問題に対処するためにやむをえないことである．そもそも，ありとあらゆる事態を想定して，その状況をいくつかの場合に分類し，もれのないように法令文を作成することなど不可能であるし，そのような抽象性があるからこそ，時代の流れに応じて法律をダイナミックに適用していけるわけである．

実データが抽象的な法律概念に該当するかどうかを判断するときに行なわれるのが，法律の「解釈」である．解釈というのは，その法律概念が現在の事件に該当するかを判断する前に，その概念の意味を明確化する操作をいう（図 3.5.1）．たとえば，「馬は通行するべからず」というルールは「厳密に馬だけ」に当てはまるのか，それとも，「馬は重量物のたとえ」に使われているので牛にも適用されるのかを判断するには，このルールにおける馬の意味を解釈しなければならない．

このように法的推論とは，「法令文にある法的概念の意味を明確にし (解釈)，与えられた事実が該当するかどうかを判断し，該当する場合に演繹的に法的結論を生成する」ことであるといえる．

解釈を行なうには

- 社会的に要請される規範：(例，公共の福祉の保持，基本的人権の尊重)
- その法令文の目的
- 他の法令文との関係
- 過去の判断例
- 法律の解釈に関する学説
- 慣習，社会常識，産業政策など

などのいろいろな知識が必要とされる．これらの知識による制約が矛盾することもあり，それらを完全に満たす解釈は存在しないことがある．その場合，異なる価値基準に立てば異なる解釈が成立し得るわけで，それらの間のバランスをとった判断をすることが法律的なセンスになる．解釈のうちで，適用される状況のパターンがかなり定まったものは，法令文のルールの詳細ルールとしてまとめることができる．しかし，解釈によっては，具体的な事実データが与えられてから動的に行なわれるものもあり，そのような場合には事前にルールとして与えることはできない．

さて，このように法律の知識は法令文以外にさまざまなものがあることがわかった．しかしながら，これらの知識ごとの知識ベースを構築し，それを用いた推論をモデリングすることはほとんど不可能といえる．ある知識は抽象的すぎてその判断は主観的にならざるを得ないし，ある知識は量が多すぎて記述しきれないであろう．

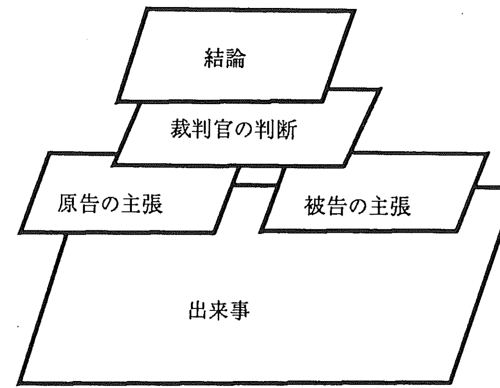


図 **3.5.2** 判例に含まれる情報

そこで，「判例」に着目してみよう．判例は，過去の特定の事件について，どのような状況が存在し，原告と被告はどのような主張をし，裁判官はどのような観点からどのような判断をしたかが記述されている (図 3.5.2)．判例には，上記のさまざまな知識がその事件についてどのように用いられたかという情報が含まれているので，判例を参考にすれば，他のさまざまな知識ベースを構築することなく，これらの知識をカバーできる可能性がある．過去の類似の判例を探して，現在の事件の法的推論の参考にすることは，実際に法律の専門家も行なっている．もちろん，法的判断は，それぞれの事件の特殊事情に影響されるから，判例はあくまでも参考資料にすぎないことに注意しなければならない．

3.5.3　法的推論システム HELIC-II

前述のように，法的推論は法令文を利用した推論と判例を利用した推論の二つのモジュールとしてモデル化できる．法令文と判例の関係は，「漠然としたルール」と「そのルールの運用実績」の関係に等しいから，このモデルは世の中の幅広い問題に適用可能である．このモデルを並列推論マシン PIM 上のソフトウェアツールとして開発したのが HELIC-II である．

HELIC-II は「ヘリクツ」と読む．法律において，過去の判例を引用して，自分に有利な論理を展開するのが目的だからである．

HELIC-II は図 3.5.3のように二つの推論エンジン（ルールベースエンジンと事例ベースエンジン）と概念辞書，ルールベースと事例ベースからなる．法令文はルール形式で記述され，ルールベースに格納されている．判例は「出来事の客観的な記述」と「原告/被告の双方の主張」の組として，事例ベースに格納されている．

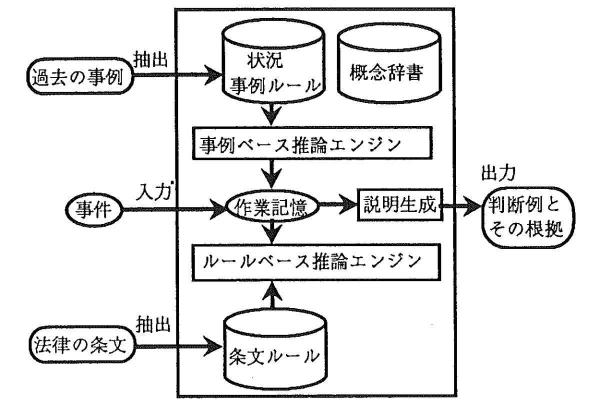


図 **3.5.3** HELIC-II の構成

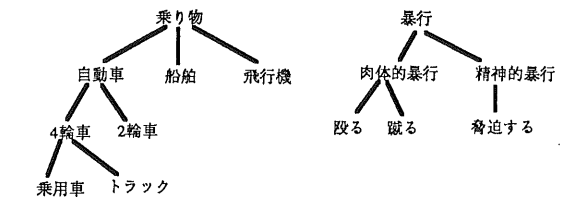


図 **3.5.4** 概念間の階層関係

概念辞書中の概念の間には，上位／下位の関係に基づく階層構造が成立している (図 3.5.4)．概念間の類似性は，この階層構造上の距離で定義される．たとえば，「乗用車」と「飛行機」の関係より「乗用車」と「トラック」の関係のほうが類似性が高いことになる．

以下では，これらの二つの推論エンジンの役割を紹介する．

A.　判例の表現と判例に基づく推論

a.　判例の表現

まず，判例をどのように表現し，事例ベースに格納するかについて説明する．

前述のように，HELIC-II では，判例の情報を「出来事の記述」と「双方の主張」に分けて表現する．それぞれの情報の記述の単位となるのがオブジェクトである．オブジェクトは，個々の人物や権利や出来事などを表わすものであって，以下のように「オブジェクトの上位概念を表わす述語」と「オブジェクトの識別子」と「そのオブジェクトに関する性質 (属性名=属性値 の対のリスト)」で表現したものである．

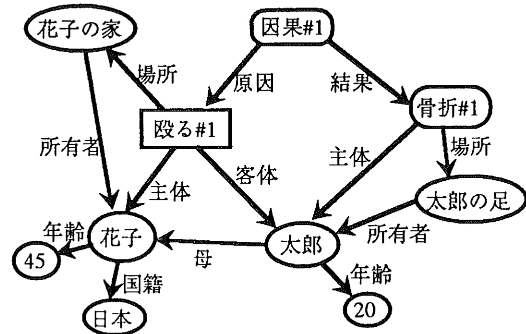


図 **3.5.5** 出来事の記述

自然人 (太郎, [年齢=20, 母=花子])
自然人 (花子, [年齢=45, 国籍=日本])
家 (花子の家, [所有者=花子])
殴る (殴る#1, [主体=花子, 客体=太郎,
　　場所=花子の家])
骨折 (骨折#1, [主体=太郎, 場所=太郎の足])
因果関係 (因果#1, [原因=殴る#1,
　　結果=骨折#1])

「出来事の記述」は対象とする事件の概要をオブジェクトの集合で記述したものである．この情報は，図3.5.5 のように，オブジェクトをノードとし，属性をリンクとした意味ネットワークとして考えることもできる．

「双方の主張」は，その事件の原告／被告の主張を「～の場合は～とすべきだ」の形のルールで表現したものである．このルールは一般のルールと異なり，変数を含まないので「事例ルール」と呼ぶことにする．たとえば,「交通事故の加害者である甲が，被害者である乙を一時的に置去りにした行為は，あとで乙を捜しに現場に戻った事実があるにしても，保護責任者遺棄罪における遺棄行為にあたると考えるべきである」のような主張が事例ルールである．

事例ルールの条件部は，その事件の状況の意味ネットワークのサブセットである (図3.5.6)．事例ルールの条件部のなかで，その結論を出すのに不可欠の情報とそうでない情報があるので，ネットワークのリンクに重要度の重みを付与する．重みには exact, important, trivial の三つがあり，それぞれ「絶対にマッチしなければならない条件」「マッチすることが重要な条件だが，必ずしもマッチしなくともよい」「マッチすることが望ましいが，それほど重要ではない」を表わす．事例ルールは，事件の状況記述のうちで，論争をするのに着目すべき事実は何かを表わしているといえる．

裁判での原告／被告の論争は，このような事例ルールや法令文ルールの連鎖である [41]．

b.　判例に基づく推論

事例ベース推論エンジンの役割は，新しい事件が与えられたとき，過去の類似の事例ルールを利用して，どのような法的概念が主張できるかを列挙することにある．これは以下の二つの段階で行なわれる．まず第1段階として，新しい事件と各種の判例の出来事の粗い比較を行ない，類似する判例をピックアップする．次に第2段階として，選択された判例に含まれている事例ルールの条件部を新しい事件と比較して，類似するものがあればそのルールの実行部を実行して仮定的な法的概念を生成する．

新しい事件と判例との類似性は，双方の意味ネットワークの比較によって行なわれる．図3.5.7の例では，(a) は「不発弾の処理を依頼されていたにもかかわらず，放置し，子供のいたずらで爆発したときは，役所に責任がある」という事例ルールの条件部であるとする．(b) は「野犬の処置を依頼されていた保健所が十分な対応をする前に，幼児に噛みついた事件」の記述である．両方のネットワークを比較し

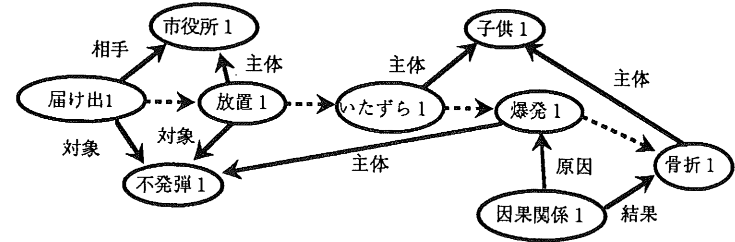
不発弾 1	と	野犬 2
届出 1	と	処置要請 2
放置 1	と	放置 2
爆発 1	と	噛みつく 2
因果関係 1	と	因果関係 2
子供 1	と	通行人 2
骨折 1	と	裂傷 2

なるノードと，それらをつなぐリンクを対応させてみよう．すると，二つのネットワークに類似の構造が存在することがわかる．このように二つのネットワークをなるべく多くのリンクがマッチするように，ノード同士とリンク同士の対応をとることによって，両者の類似度を判断することができる．ここで両者の類似度は

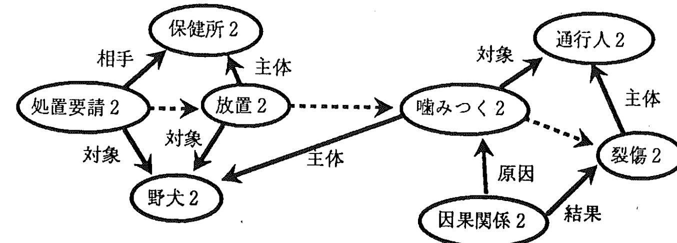
$$\text{類似度} = \sum(-\text{対応するノードの距離} + \text{対応するリンクの重み})$$

IF　　交通事故 (事故#1, [主体=甲/exact, 手段=自動車#1/trivial]),
　　怪我 (怪我#1, [主体=乙/exact]),
　　因果関係 (因果#1, [原因=事故#1/exact, 結果=怪我#1/exact]),
　　置去り (置去り#1, [主体=甲/exact, 客体=乙/exact]),
　　..........
THEN　遺棄 (遺棄#1, [主体=甲, 客体=乙, 対象行為=置去り#1]).

図 **3.5.6** 事例ルール



(a)　爆弾のケース



(b)　野犬のケース

図 **3.5.7** 状況の類似

で定義される．ここで,「ノードの距離」とは，前述したように概念辞書における概念間の距離である.「リンクの重み」はリンクにつけられた exact, important, trivial をそれぞれ重みに変換したものである．

この類似度があらかじめ設定されたスレッショルド以上であれば,「事例ルールの条件部が満足された」と判定され，事例ルールが発火する．したがって，事例ルールは，条件部のすべてが満足されなくても発火することがある．発火したルールは，その実行部が実行され，新しい概念が意味ネットに追加される．

事例ルールは状況の記述を抽象化するのにも使うことができる．たとえば,「...のような状況のもとでは，殴る，脅す，金品を奪う，の一連の行為が強奪するに該当する」という事例ルールは，三つの行為を一つの抽象的な行為に抽象化するものである．したがって，個々の判例の状況記述の詳細度に差があっても，事例ルールを用いて抽象化していく過程で類似性が見いだされることがありうる．

B.　法令文の表現と法令文に基づく推論

a.　法令文の表現

法令文は論理式で記述することができる．たとえば,「過失に因り人を死に致したる者は千円以下の罰金に処す」は，過失致死罪の規定であるが，この前半部は図3.5.8のように表現することができる．

法令文の論理式のなかでは，2 種類の否定 (logical not と negation as failure) を用いることができるようにしており，それぞれ，“∼” と “not” で表現される．“∼ 殺意” は「殺意がなかった」ことを表わし，“not 殺意” は「殺意があったことが証明できなかった」ことを表わす．法令文には，二つの否定がしばしば現われる．また，例外規定は二つの否定の組合せで表現す

自然人 (A), 自然人 (B),
行為 (Act, [主体=A]), 過失 (Neg, [主体=A, 行為=Act]),
因果関係 (Cause, [行為=Act, 結果=Death]), 死 (Death, [主体=B])
　→ 構成要件該当 (K, [行為=Act, 罪=過失致死罪])

図 **3.5.8** 法令文の表現 (1)

自然人 (A), 行為 (Act, [主体=A]), 構成要件該当 (K, [行為=Act, 罪=Crime]),
罰則規定 (P, [罪=Crime, 刑=Punish]), not (～ 罰則 (P2,[刑=Punish]))
　→ 罰則 (Penalty, [刑=Punish]).

心神喪失 (N, [主体=A, 行為=Act]) → ～ 罰則 (P2,[刑=Punish]).

図 **3.5.9** 法令文の表現 (2)

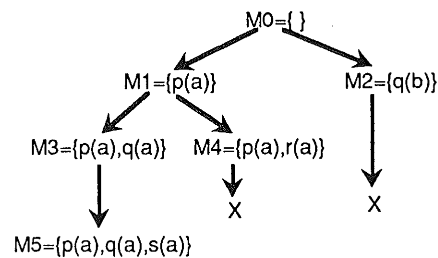


図 **3.5.10** MGTP の証明木

ることができる．たとえば，「心神喪失者の行為は之を罰せず」は，上記の過失致死罪などのような刑罰の定義規定の例外規定となるから，図 3.5.9のように表現することができる．

b. 法令文に基づく推論

推論は，並列定理証明プログラム MGTP[42] を拡張した推論エンジンを用いて前向き推論を行ない，考えられる法的結論をすべて列挙するようになっている．

MGTP の働きを以下の例を用いて説明しよう．

$true \rightarrow p(a); q(b).$　　$C1$
$p(X) \rightarrow q(X); r(X).$　　$C2$
$r(X) \rightarrow s(X).$　　$C3$
$q(X) \rightarrow false.$　　$C4$

証明は空モデル $M0 = \{\}$ から始まる．$M0$ に対して $C1$ を適用すると，二つのモデル $M1 = \{p(a)\}$ と $M2 = \{q(b)\}$ が生成される．$M1$ に $C2$ を適用すると，$M3 = \{p(a), q(a)\}$ と $M4 = \{p(a), r(a)\}$ が生成される．ここで $M3$ に $C4$ を適用すると，内部に $false$ を含むので，このモデルは棄却される．$M4$ に $C3$ を適用すると，$M5 = \{p(a), r(a), s(a)\}$ が生成される．$M2$ に $C4$ を適用すると，$false$ が生成されるので棄却される (図 3.5.10)．

このように MGTP では，論理式を適用することによってモデルを次々に生成し，最終的に残ったモデル M5 がすべての論理式を満たすモデルとなっている．

筆者らは，オリジナルの MGTP を法的推論のエンジンとして利用するために，いくつかの拡張を行なった．その一つは前述の Negation As Failure の実現であり，もう一つは以下のような複数の仮説における推論の実現である．

事例ベース推論エンジンは，いくつかの事例ルールを利用して推論を行なう．元の事例ルールには原告や被告の主張が含まれているから，相互に矛盾するものがある．たとえば，いまの事件が二つの論点 p, q をもち，事例ベース推論エンジンが異なる判例を利用して，原告有利の $p(a)$ と $q(b)$，および被告有利の $\sim p(a)$ と $\sim q(b)$ を生成するかもしれない．このような場合に，法令文による推論では，初期状態を四つの仮説パターン $[p(a), q(b)]$，$[\sim p(a), q(b)]$, $[p(a), \sim q(b)]$, $[\sim p(a), \sim q(b)]$ に分類し，それぞれのパターンごとに法令文の適用を試みる (図 3.5.11)．

3.5.4 刑法への HELIC-II の応用

前項で HELIC-II の基本的な機能を紹介した．本項では，これらの機能で，どの程度の問題が解決できるのかを考察する．

筆者らは機能を実証する実験として，刑法の論理構築システムを開発している．刑法では，さまざまな犯罪の定義が規定されている．しかしながら，法的概念の詳細な定義や判断基準が与えられていないため，実問題では，ある概念に該当するか否かがしばしば問題となる．

たとえば，殺人の定義は法律では与えられていない．ある行為が殺人に該当するためには，その行為を行なうときに「殺意」があったことと，その行為によって他人の死が生じたこと (「因果関係」があったこと) が立証されなければならないことには異論がないであろう．しかし，殺意や因果関係の判断はしばしば論争の原因になる．人違いで拳銃を撃った場合や，ささいな行為をしただけなのに予想もしない事態が生じて死亡したような場合がその典型例である．

そこで筆者らは，司法試験の刑法の事例問題をいくつか選択して，その推論過程を解析し，過去の判例がどのように利用されているかを調べた．その後，関連の判例を集めて HELIC-II にのせ，この事例問題がどのように解かれるかを実験した．

以下は，実験に用いた事例の一つである．

> 「ある冬の日，生活に疲れた甲女は，生後 4 か月の太郎を道端に置去りにした．通りがかりの乙は太郎を警察に届けようと，車に乗せた．しかし，途中で交通事故を起こし，太郎は大怪我をする．乙は太郎が死んだと思い，その場に放置して逃げたところ，太郎は凍死してしまった．甲と乙の罪は何にあたるか．」

甲においては

- 置去りにした行為は保護責任者遺棄罪の遺棄にあたるか．
- 置去りにしたとき，太郎の死は予測できたか．
- 置去りが太郎の死を引き起こしたか．(因果関係)

が論点となり，乙については

- 保護責任者遺棄罪の保護責任者に該当するか．
- 置去りにした行為は保護責任者遺棄罪の遺棄にあたるか．それとも死体遺棄罪の遺棄にあたるか．(故意)
- 交通事故が太郎の死を引き起こしたのか．(因果関係)

などが論点となる．これらについて，HELIC-II は過去の判例を引用して，検察に有利な判断や被告に有利な判断を複数生成した．たとえば，以下の例は引用された判例の一つの状況である．

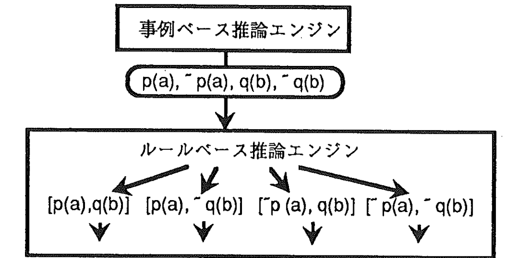


図 **3.5.11** 初期状態の分割

> 「浦田某はかねてから同居している市の助を殺そうと思い，就寝中に首を絞めた．市の助は気絶したが，浦田は市の助が死んだものと思い，犯行を隠すために海岸に連れていって置去りにした．市の助はうつぶせにおかれたので，砂を吸って窒息死した．」

判例によると，検察側は，「首を絞めたことによって死にはしなかったが，海岸に置去りにしたことよって死んだのである．置去りは首絞めを隠すための行為だから，両者は一体であり，首絞めと死の間には因果関係がある．よって，殺人罪である」という論理を展開した．一方，弁護側は，「首を絞めた行為は殺人未遂罪にあたる．また，海岸に捨てた行為は，殺意がないから，過失致死罪にあたる．したがって，殺人罪にはならない」という論理を展開した．

乙の行為と浦田の行為の類似性から，HELIC-II は双方の論理 (事例ルールで表現されている) を利用して，正反対の論理を生成した．

それぞれの判断の結果は推論木 (図 3.5.12) と自然言語で出力される．木のルートのノードは推論結果を表わし，末端のノードは初期の事実データである (図 3.5.12は実際の推論木より簡略にしている)．

浦田事件を参考にしたことによる論理をまとめると，「交通事故によって死にはしなかったが，道路に置去りにしたことによって死んだのである．置去りは事故を隠すための行為だから，両者は一体であり，交通事故と死の間には因果関係がある．よって，業務上過失致死罪である」と，「交通事故は業務上過失傷害罪にあたる．また，道路に置去りにした行為は，過失致死罪にあたる」というものであった．

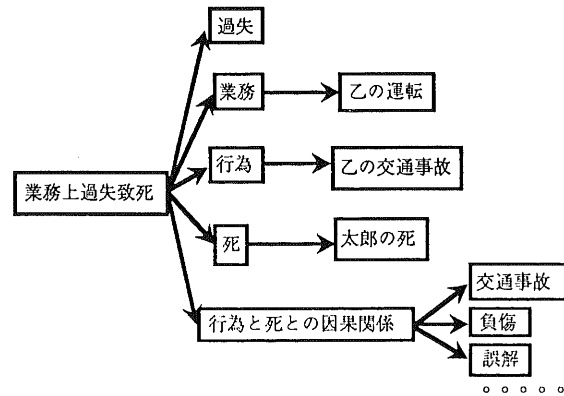


図 3.5.12 生成された推論木の例

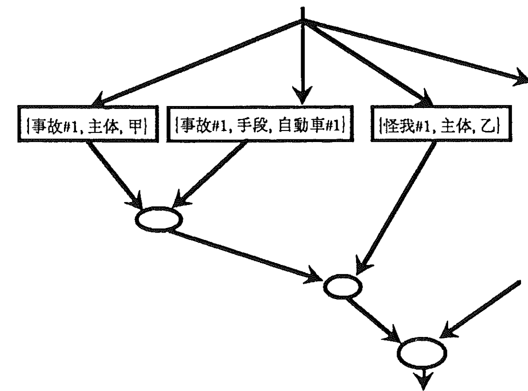


図 3.5.13 条件部のネットワーク

3.5.5 並列推論マシンによる高速化

前項までに法的推論システム HELIC-II の概要を説明した．この推論メカニズムをコンピュータで実現すると

- 判例の数が多いこと
- 判例の類似判定に時間がかかること
- 仮説の組合せが膨大なこと
- 法令文の数が多いこと

などにより，計算時間が非常にかかることが予想される．いままでに，欧米を中心に法的推論システムのいろいろなモデルが提案されてきているが，どのモデルも同じ問題を抱えている．筆者らは，このシステムを並列推論マシン PIM の上で高速化することにより，計算時間の問題の解決を図った．

以下に並列処理を効率的に行なうためにとった手法をいくつか紹介する．

A. 判例に基づく推論の並列処理

判例に基づく推論は，類似判例の検索と事例ルールを用いた論理構築の 2 段階からなる．類似判例の検索の基本的なアイディアは，判例をあらかじめ複数のプロセッサに分配しておき，検索を並列に行なうことである．現在の事件と個々の判例の間の類似判断を行なった結果は一つの要素プロセッサに集められ，一定の基準以上の判例がまず選択される．

個々の判例は，通常 5，6 程度の事例ルールをもっている．選択された判例の事例ルールは要素プロセッサに分配され，それぞれの要素プロセッサにおいて，条件部の類似判定が並列になされる．前述のように類似判定は，二つの意味ネットワークのノードやリンク間のマッピングを行なうことによる．このマッピングは，事例ルールの条件部をいわゆる Rete ネットワークと類似のネットワークに展開することによって行なわれる．

たとえば，図 3.5.6の事例ルールの条件部は図 3.5.13 のように展開される．

このネットワークにおいて，四角のノードは，入ってきたデータがその条件に合致するかどうかを選択するものである．また楕円のノードは，2 か所から入ってきたデータが矛盾がないかどうかをチェックするものである．

このネットワークのルートから，以下のような新しい事件のネットワークの情報

自然人 (太郎, [年齢=20, 母=花子])
自然人 (花子, [年齢=45, 国籍=日本])
家 (花子の家, [所有者=花子])
殴る (殴る#1, [主体=花子, 客体=太郎, 場所=花子の家])
骨折 (骨折#1, [主体=太郎, 場所=太郎の足])
因果関係 (因果#1, [原因=殴る#1, 結果=骨折#1])

を流すと，対応のとれたノードのマッピングデータがネットワークから出力される．

図 3.5.14は，論理構築モジュールの並列処理の効果を示したグラフである．横軸に使用したプロセッサの数を示し，縦軸に実行時間を 1 台プロセッサのときの実行時間で割ったときの値（台数効果）を示している．

図で示すように，並列処理により，ほとんど使用プロセッサ数に比例した台数効果が得られている．

B. 法令文に基づく推論の並列処理

前に述べたように，法令文に基づく推論のエンジンは，並列定理証明プログラム MGTP に基づいている．MGTP は入力された個々の論理式を KL1 の節に変換し，それを KL1 プログラムとして実行することによって並列処理を行なっている．MGTP の並列性は，推論木 (図 3.5.10) の分岐の数が多ければ多いほど引き出される．

筆者らは，3.5.3項で述べたように，MGTP に (i) negation as failure の実現と (ii) 複数の仮説における推論の実現，の拡張を行なった．これらは結果として，推論木の分岐数を増やすこととなり，並列処理の効果を一層推進することとなった．

(1) Negation as failure の実現

MGTP で以下のような否定の論理式が現われたとする．

$$p(X), not(r(X)) \rightarrow s(X).$$

この論理式は以下のように K オペレータを用いて別の論理式に変換される．

$$p(X) \rightarrow k(r(X)); \sim k(r(X)), s(X).$$

"$k(r(a))$" は，「将来，$r(a)$ が現われることを表わす仮説」である．したがって，すべての論理式を適用したあとで，最終的にモデルのなかに

- $k(r(a))$ が含まれているのに，$r(a)$ が存在しない
- $\sim k(r(a))$ と $r(a)$ の双方が存在する

ようなものは棄却される [43]．

(2) 複数仮説の推論

また，複数の仮説における推論の実現においては，事例ベース推論エンジンの推論結果を待たなければ，ルールベース推論エンジンが実行できなくなり，並列処理の観点からは無駄な待合せが生じることになる．そこで，事例ベース推論エンジンの出力が一部だけしか得られていない段階でも，ルールベース推論エンジンが実行できるように，新たに s オペレータを導入した．

これは，論争の対立する可能性のある述語 p のセットをあらかじめ登録しておき，事例ベース推論エンジンが，たとえば $\sim p(a)$ のように，肯定あるいは否定の一方のデータを生成すると，対立する見解 $p(a)$ もあとから生成されるものと予測して，$\{\sim p(a)\}$ と $\{s(p(a))\}$ という二つの初期モデルを生成して，ルールベースエンジンを実行させる．証明の最後に

- $s(p(a))$ が含まれているのに，$p(a)$ が存在しない
- $s(p(a))$ と $s(\sim p(a))$ の双方が存在する

ようなモデルは棄却される．

このような s オペレータの導入により，ルールベース推論エンジンの並列処理の効果を上げることができた．図 3.5.15は，ルールベース推論エンジンの台数効果のグラフである．

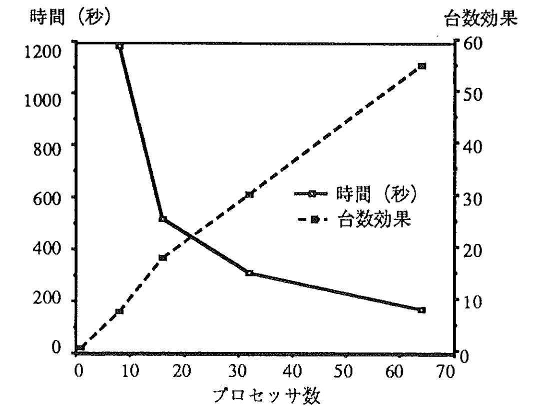


図 3.5.14 論理構築モジュールの台数効果

3.5.6 開発過程

本研究は，1989 年後半に並列推論マシン上のプロダクションシステムと仮説推論システムの研究としてスタートした．KL1 でこれらの推論システムの並列プログラムを開発し，その応用として法的推論システムの開発を行なったものである．

1990 年に事例ベース推論エンジンが完成し，「持病をもった人が，仕事の途中で死亡したときに，過労死として認められるか」を判定するシステムを試作した．労働省を訪問して，過労死の判定基準の資料をいただいたり，国会図書館で判例を集めたりして，事例ベースを試作した．個々の判例を読んで，事例ルールを抽出するのは容易に見えて骨の折れる仕事である．同じ判例であっても用いる概念によって，まったく異なる表現が可能である．そこで，当時のグループリーダの

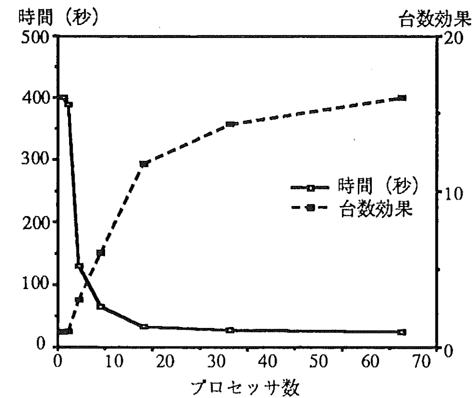


図 3.5.15 ルールベース推論エンジンの台数効果

石川にすべての判例に目を通してもらって，どのような概念でまとめるかの統一基準を作成してもらった．その後，判例をグループのメンバーに分担して，その基準に従って事例ルールの抽出を行なった．

1991 年に MGTP を拡張したルールベース推論エンジンを完成させ，HELIC-II の原型が完成した．同時に対象を労働災害から刑法に移行した．これは，労働災害補償法では十分な解説書がないこと，判例に比べて法令文の数が少なく，バランスがとりにくいことなどによる．刑法を選択するにあたって，法律の専門家の方々のご意見をうかがった．多くの専門家の方は，刑法よりも民法や商法を推薦されたが，素人にもわかりやすいことから刑法を選択した．ただし，HELIC-II の推論の枠組自体はどの法律を選択するかには依存しない．

3.5.7 むすび

法的推論システム HELIC-II の機能とその並列処理を中心に紹介した．法的推論は，膨大な判例検索と高度な推論を必要とするので，並列推論に向いた研究テーマである．たとえば，本稿で仮説推論や非単調推論などの高次推論の技術が使われていることが理解されるだろうし，ここでは述べていないが，時間推論のモジュールも組み込まれている．判例と法令文を用いた推論は，法的推論だけでなく，幅広い分野に適用可能な汎用的なメカニズムである．今後は，HELIC-II の推論メカニズムの形式化と原告／被告の論争のモデリングなどに研究対象を広げていく予定である．

3.6 定理証明系 MGTP

3.6.1 概説

A. はじめに

定理証明とは，「三角形の一辺は他の二辺の和より短い」とか「素数は無限に存在する」といった命題を表現する論理式が，公理と呼ばれる論理式群から三段論法などの推論規則に従って論理的に演繹されることを機械的に証明する技術であり，自動推論技術とも呼ばれている．ここで，このように公理から論理的に演繹される (帰結される) 命題を定理と呼んでいる．

定理証明の究極の夢は，フェルマの定理のような前人未到の問題を自動的に解くことであるが，数学の定理を証明すること以外にもさまざまな有益な用途がある．たとえば，1) データベース中の事実を基に，論理式で表わされた質問に応答すること，2) プログラムの実行を論理式で表現し，意図どおりにプログラムが動作するかどうかを検証したり，また，論理式で仕様を与えてこれを満たすようなプログラムを合成すること，3) ある制約条件のもとで，計画や設計をしたりすることや，最近では 4) 遺伝子の構造や性質の解析などに利用することも考えられる．

これらの問題は，$\forall x A(x)$ (すべての x に対して $A(x)$ が成立する)，$A \wedge B$ (A かつ B) とか $A \rightarrow B$ (A ならば B) といった形を組み合わせた一階述語論理の論理式で素直に表現することができる．

定理証明系 MGTP は，一階述語論理の自動推論を行なうものである．一階述語論理は，知識を論理式として表現する場合の枠組みとしてひろく利用できるが，そこで表現された知識を公理と考えると，定理証明系は知識を基に自動推論を行なう汎用的な推論エンジンととらえることができる．

われわれは MGTP の適用範囲として，数学的定理証明に留まらず，FGCS プロジェクトのなかだけでも演繹データベース，仮説推論，法的推論，ソフトウェアやハードウェアの検証・合成，自然言語処理など，知識処理領域の大半をカバーすることを考えている．実際に法的推論システム HELLIC-II では，法令のルールベースとして組み込まれ使用されている．

このような定理証明系について，まず研究開発の背景を眺めることにしよう．

B. 研究開発の背景

一階述語論理の定理証明は，第五世代コンピュータの基本概念である「論理に基づく推論」を実現する一つの重要な方法である．

Prolog や KL1 などの論理型言語は，一階述語論理のサブセット (一部分) をなすホーン論理の自動証明の研究過程から生まれたものであり，元をたどればここで紹介する定理証明系ときわめて近い起源をもっている．ここで扱う一階述語論理は，ホーン論理を包含しており記述能力の点ではより優れているが，そのために定理証明系の実行効率はホーン論理に限定した場合より悪く，実用性の観点からはこれまであまり顧みられることがなかった．

しかし，近年の論理プログラミング技術の進展を背景に，これを見直す動きが出始めている．その典型例は Prolog 技術をうまく使った PTTP[44] や SATCHMO [45] などである．前者は，証明したい命題の否定 (ゴール) から始めて，公理を適用して部分ゴールに細分化していくことによって矛盾を導く (反駁する)，後向き推論を行なう定理証明系である．一方，後者は公理から始めて，証明したい命題が得られるまで，次々に定理を生成していく，前向き推論に基づいた定理証明系である．

筆者らは，かかる動きに触発され，並列論理型言語である KL1 をベースに，平成 2 年より一階述語論理のモデル生成型定理証明系 MGTP の研究を開始した．MGTP 研究の動機は二つある．一つは，論理プログラミングをベースとした一階述語論理の定理証明システム構築の研究を通じて，ホーン論理を基盤とする現在の論理プログラミング技術を一般の一階述語論理へと拡張していくことである．もう一つは，自動推論の研究から派生し成長してきた論理プログラミングの技術を自動推論分野に逆に再適用することによって，この分野に革新をもたらすことである．すなわち，一度は離反した論理プログラミングと自動推論の融合がわれわれの目指すところである．

本研究の当面の目標は，KL1 の特長を活用して高速な並列自動推論システムを並列推論マシン PIM 上に構築することである．定理証明の過程には多大な並列性が内在しており，膨大な計算量，大量の記憶域を必要とするので，定理証明系自身，KL1 および PIM にとって格好の応用の一つでもある．

C. MGTP について

定理証明系 MGTP では，証明の方式として SATCHMO のモデル生成法を採用した．その理由は，以下のとおりである．

1) モデル生成法は，基底 (グランド) アトムのみがモデル要素として生成される場合，ユニフィケーションが不要でマッチングで十分であるので，KL1 が提供する高速なマッチング (ヘッド・ユニフィケーション) を利用して，効率よく実装できる．
2) 数学的定理のように深い推論 (長い証明) が必要となる問題を解く場合，モデル生成法のようなボトムアップ型の定理証明系は，探索空間を狭めるための各種の技法，すなわち補題化，包摂テスト，削除戦略等を容易に組み込むことができる．

現在，変数を含まない基底アトムのみを扱うグランド版と，変数含みのアトムを扱うノングランド版の，2 種の MGTP が開発されている．

グランド版 MGTP の場合，ユニフィケーションが不要でマッチングですむので KL1 との親和性がよく，簡潔かつ効率のよい証明系が得られた．また，グランド版は，ノンホーン節のケース分割によって OR 並列性が容易に抽出できるという利点をもっている．

一方，ノングランド版 MGTP の場合，オカーチェックつきユニフィケーションをユーザが (KL1 で) 書く必要があり，1) で述べた KL1 の利点を活用できない．したがって，グランド版ほどの高速性は期待できないが，KL1 によるユニファイアの実装は，KL1 の実用性を見る上でよい評価材料となった (C で書かれたものと比べて，2〜3 倍程度の遅さという良好な結果が得られている)．また，ノングランド版では，ケース分割が生じない (OR 並列性のない) ホーン節に対象を限定しているので，これからいかに (AND) 並列性を抽出するかという，AND 並列化方式の確立が研究の主眼であった．

現在のところ，両版とも PIM/m-256PE 上で 200 倍以上の台数効果を達成している．グランド版 MGTP を用いて，有限代数の問題を試みた結果，カナダの数学者 Bennett により提示された準群に関する未解決問題の一部を解くことに成功した [49, 50, 51]．また，ノングランド版 MGTP は，PIM/m の単一 PE 上で，米国アルゴンヌ国立研究所 (ANL) 開発の OTTER と

同程度かあるいはこれをしのぐ性能を達成しており，OTTERでは解けなかったハードな数学的定理を解くことにも成功している [52].

MGTP の研究過程で，SATCHMO に含まれていた連言照合の冗長性を除去する手法，ケース分割の爆発を防ぐ手法 [53]，アトムの過剰生成を抑制して計算量・記憶域を削減する手法 [54]，などのモデル生成アルゴリズムの改良技術や，論理プログラミングで開発された「失敗による否定」を組み込む技術 [55]，さらに様相論理システムや仮説推論システムなどの構築技術 [56, 57] が生み出されてきた．

誌面の関係上これらのすべてを紹介できないので，本別冊の趣旨に沿って，本稿では，MGTP の並列化方式とその実現手法に焦点をあてる．以下では，まずモデル生成法とは何かを概説し，MGTP の証明手続きおよび基本的な処理構造について簡単に述べる．そして MGTP の証明過程における並列性の所在について言及し，OR 並列化方式および AND 並列化方式の考え方と実現方法を紹介する．最後に実験結果と今後の課題について述べる．

3.6.2 モデル生成法

本文を通じて節は次のように含意式の形で表現される．

$$A_1, A_2, \ldots, A_n \rightarrow C_1; C_2; \ldots; C_m$$

ここで，$A_i (1 \leq i \leq n; n \geq 0)$ および $C_j (1 \leq j \leq m; m \geq 0)$ はアトムである．→ の左側を節の前件，右側を後件という．前件は $A_1, A_2, \ldots, A_n$の連言 (‘,’)，後件は $C_1, C_2, \ldots, C_m$の選言 (‘;’) である．式の意味は，$A_1, A_2, \ldots, A_n$ のすべてが成り立てば，$C_1, C_2, \ldots, C_m$ のうちいずれかが成り立つ，というものである．

$n = 0$ のとき，前件部を特に $true$ と書き，$true \rightarrow C_1; C_2; \ldots; C_m$ を正節と呼ぶ．一方 $m = 0$ のとき，後件部を特に $false$ と書き，$A_1, A_2, \ldots, A_n \rightarrow false$ を負節と呼ぶ．それ以外の節 $(m \neq 0, n \neq 0)$ は混合節と呼ばれる．さらに，正節および混合節を generator 節，負節を tester 節とも呼ぶ．

モデル生成法は，与えられた節集合に対するモデル[†1]を，空集合から始めて構成的に求める証明手法である．

モデル生成法には次の二つの規則がある．規則中 M は構成途中のモデルを表わし，これをモデル候補と呼ぶことにする．

- モデル拡張規則： 混合節もしくは正節の前件がモデル候補 M のもとで充足しており，後件が充足していないとき，後件で M を拡張するもので，もう少し詳しく述べると次のようになる．混合節もしくは正節 $A_1, A_2, \ldots, A_n \rightarrow C_1; C_2; \ldots; C_m$ において，ある置換σ のもとに，各前件アトム $A_i\sigma$ があるモデル候補 Mで充足されており，いずれの後件アトム $C_j\sigma$ も M で充足されてないとき，各 $C_j\sigma$ を M に加えてモデル候補を拡張する[†2]．
- モデル棄却規則： 負節 において，ある置換 σ のもとに，各前件アトム $A_i\sigma$ があるモデル候補 M で充足されるとき，M を棄却する．

ここで，$(A_1, A_2, \ldots, A_n)\sigma$を得る過程を，節の前件とモデル候補要素との連言照合という．正節の前件 $(true)$ はどんなモデルでも充足されることに注意されたい．

M を拡張する際に $C_j\sigma$ が M で充足されているか否かを検査しているが，これは包摂テストと呼ばれる検査の一種である．二つの項 u, v 間に $u\sigma = v$ が成り立つとき，u は v を包摂するといい，このように，より一般的な項を残す操作を包摂テストという．モデル拡張によって生成されたアトムがモデル候補要素に包摂されるか否かを検査することを前向き包摂テスト，逆にモデル候補要素を包摂するようなアトムかどうかを検査することを後向き包摂テストという．

モデル生成法における証明は，以下のように行なわれる．まず，初期モデル候補集合として $\mathcal{M} = \{\phi\}$ から始め，いずれの規則も適用することができなくなった時点で，最終的なモデル候補の集合 $\mathcal{M}$ を出力する．このとき，すべての $M \in \mathcal{M}$ について，M はすべての節を満足しているため，与えられた節集合のモデルとなる．もし，$\mathcal{M} = \phi$ であれば，すべてのモデル候補が

†1 ここでモデルとは，真であると解釈されるアトムの集合のことである．モデルに含まれないアトムは，偽であると解釈される．

†2 $M \cup \{C_j\sigma\} (1 \leq j \leq m)$ を新たなモデル候補とする．

棄却されており，モデルが存在しないため，その節集合は矛盾している (あるいは充足不能である) ことを示す．

例として次の節集合 S1 を考える．

$C1:\ p(X), s(X) \rightarrow false.$
$C2:\ q(X), s(Y) \rightarrow false.$
$C3:\ q(X) \rightarrow s(f(X)).$
$C4:\ r(X) \rightarrow s(X).$
$C5:\ p(X) \rightarrow q(X); r(X).$
$C6:\ true \rightarrow p(a); q(b).$

S1 問題に対する証明木を図 3.6.1に示す．まず，初期モデル候補集合として $M_0 = \{\phi\}$ から始める．$C6$ にモデル拡張規則を適用することにより，M_0 は $M_1 = \{p(a)\}$ と $M_2 = \{q(b)\}$ に場合分けされる．次に，$C5$ によって M_1 は $M_3 = \{p(a), q(a)\}$ と $M_4 = \{p(a), r(a)\}$ に場合分けされる．さらに，$C3$ によって M_3 は $M_5 = \{p(a), q(a), s(f(a))\}$ に拡張される．さて，M_5 においてモデル棄却規則が $C2$ に適用できて，M_5 は棄却され，このケースの処理はここで終了する．一方，M_4 は $C4$ によって $M_6 = \{p(a), r(a), s(a)\}$ に拡張されるが，$C1$ によって棄却される．同様に，M_2 は $C3$ によって $M_7 = \{q(b), s(f(b))\}$ に拡張されたあと，$C2$ によって棄却される．いまや，モデルを構成する可能性のすべてが絶たれたので，S1 は充足不能であると結論できる．

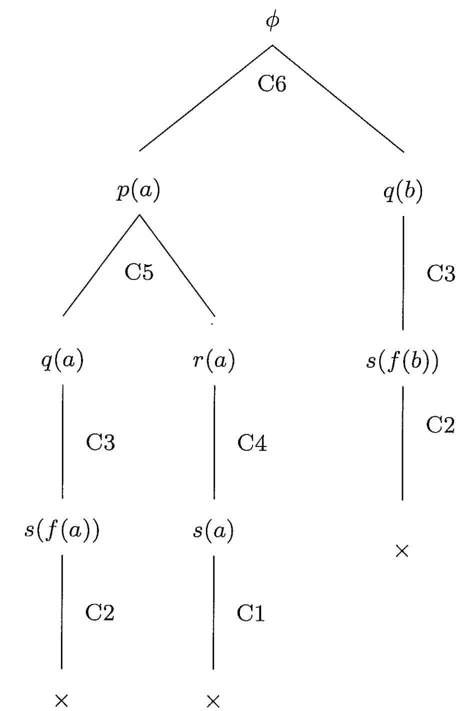


図 3.6.1 S1 に対する証明図

3.6.3 MGTP の基本構造

A. モデル生成アルゴリズム

MGTP で採用しているモデル生成アルゴリズムを図 3.6.2 に示す．ただし，本図は証明木の一つの枝の探索を行なう手続きを示したものであり，場合分けに対する手続きは含まれていない．ここで，Mはモデル候補を，Dはモデル拡張候補 (モデル拡張規則の適用の結果として Mに付加されるべきアトムの集合) を，ΔはDの要素をそれぞれ表わす．MとDの初期値はそれぞれ空集合および正節の後件アトムの集合である．

図 3.6.2 において，(4),(6),(8) がモデル拡張規則，(7) がモデル棄却規則に対応している．**while** ループを一回りする間に，(3) D から Δ を一つ選び，(4) Δ と M を使って (混合節 $generator$ による連言照合を行なうことにより) 新たなモデル拡張候補アトムの集合 new を生成する．次に，(6) new の包摂テストを $M \cup D$ に対して行ない，包摂されなかったアトム集合を new' とする．そして，(7) new' と $M \cup D$ を使って棄却テスト (負節 $tester$ による連言照合) を行ない，成功すれば手続きは終了する．さもなければ (8) D を new' で拡張する．このサイクルの始めに Dが空ならば反駁に失敗，すなわちモデル M が見つかったことになり，このアルゴリズムは停止する．

モデル生成法にとって連言照合は基本的操作であるので，これについて少し詳しく述べることにする．n 個の前件をもつ節 $A_1, A_2, \ldots, A_n \rightarrow \ldots$ の連言照合をモデル候補 M に対して行なう場合，M の要素の n 個の組 $B_1, B_2, \ldots, B_n$ すべてについて B_i と A_i $(i = 1, \ldots, n)$ との照合を行なう必要がある．この照合の後，新たなモデル拡張候補 Δ が D から選ばれ (図 3.6.2 (3))，連言照合が再び行なわれる (同図 (4)) 場合を考えよう．今度は M ではなく $M \cup \{\Delta\}$ に対して連言照合が行なわれるので，$M \cup \{\Delta\}$ の要素の n 個の組すべてについて照合を行なわなければならない．この照合のうち n 個の要素がすべて M の要素であるような組についての照合は，以前に行なっているので冗長である．つまり，$M \cup \{\Delta\}$ の要素の n 個の組すべてについて照合を行なう必要はなく，n 個のうち

```
(0)  M := φ;
(1)  D := {A | (true → A) ∈ a set of given clauses};
(2)  while D ≠ φ do begin
(3)        D := D \ {Δ};
(4)        new := CJM_generator(Δ, M);
(5)        M := M ∪ {Δ};
(6)        new' := subsumption(new, M ∪ D);
(7)        if CJM_tester(new', M ∪ D) ∋ false then return(unsat);
(8)        D := D ∪ new';
(9)  end return(sat)
```

図 3.6.2 MGTP のアルゴリズム

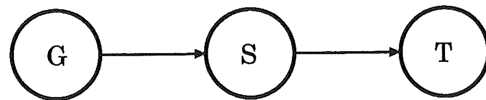


図 3.6.3 G，S，T 間のデータの流れ

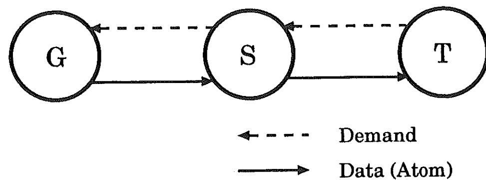


図 3.6.4 G，S，T 間のデータの流れ

少なくとも一つは Δ である組についてのみ照合を行なえば十分である．図 3.6.2 に示した連言照合 CJM は，このように冗長性を除いた操作を表わしている．

B. MGTP の処理構造

MGTP のモデル生成アルゴリズムの主な操作は以下の三つである．

1) 混合節 *generator* による連言照合を行ない，アトム集合を生成する．
2) 生成されたアトム集合の包摂テストを行ない，包摂されないアトムをモデル拡張候補に追加する．
3) 負節 *tester* の連言照合により，モデル候補の棄却テストを行なう．

MGTP は，これら三つの操作に対応して 3 種類のプロセスから構成されている．それらのプロセスをそれぞれ，生成 (G) プロセス，包摂テスト (S) プロセス，棄却テスト (T) プロセス と呼ぶ．

この手続きを並列化する際には，複数の G,S,T プロセスが生成されるので，それぞれが重複なく仕事をするために，排他制御が必要になる．

G,S,T の関係は生成されたアトム (データ) の流れに着目すると，図 3.6.3 のようになる．この図より容易に想像できるように，MGTP の計算機構は G プロセスを generator，T プロセスを tester とした generate-and-test 型になっている．一般に generate-and-test の計算では，generator の走りすぎによるアトムの過剰生成の危険性がある．これは tester にかける必要のない無用アトムを生成してしまい[†1]，それらに対して高価な包摂テストが無駄に行なわれることを意味し，メモリ空間の爆発を引き起こすことにもなる．特に並列環境下においては，generator に対する制御が何らなされない場合は，generator が暴走し，いっこうに tester が実行されないという事態も起こしかねない．この危険を回避するため筆者らは，「tester が必要とするときのみ generator を起動しアトムを生成する」という要求駆動的考え方に基づいた，遅延モデル生成法と呼ぶ新たな方法を導入した [54]．

遅延モデル生成法では，データの流れとは逆向きに要求が流れる (図 3.6.4)．G プロセスは S プロセスか

†1 ある深さで false を導くような反駁アトムが見つかるものとすると，それ以上の深さのアトムまで生成してしまうこと．

らの，S プロセスは T プロセスからの要求があってから動き始める．たとえば，G プロセスは S プロセスからの要求個数分だけアトムを生成すると，S プロセスからの要求待ち状態になる．

このような要求駆動制御により，generator と tester のスピード差が均一化され，生成の行きすぎによる無駄をなくすことができ，計算量およびメモリ消費量のオーダを大幅に削減することができる．

3.6.4 MGTP の並列化

MGTP の節は，前件 (リテラルの連言) と後件 (リテラルの選言) から構成されており，前件における連言照合操作の並列性を AND 並列，選言の場合分けによる並列性を OR 並列と呼ぶ．

MGTP の証明過程には，主に次の三つの並列性の源がある．

1) 複数のモデル候補の探索
2) 連言照合
3) 包摂テスト

1) は，後件部が複数のアトムから構成される非ホーン節で，モデル拡張を行なった場合に発生する並列性である．これは複数のモデル候補 (証明木の各枝) を同時に探索することであり，与えられた節集合が値域限定であれば，ほかとの交信なく独立に探索を行なえるので，OR 並列性とみなすことができる[†1]．

2) についてもう少し詳しく述べると，(1) 各節の連言照合は独立に行なえ，さらに (2) 一つの節に対する連言照合も並列に実行できる．しかし，節中に共有変数がある場合，これらの変数は矛盾しない値をとらなければならない．後件部が一つのアトムのみからなるホーン節を扱う場合は，モデル候補が一つしか存在せず (証明木は一本の枝からなる)，連言照合が並列性の大きな源となる．この場合，ホーン節集合のすべてのアトムは唯一つの解に関与するので，ホーン節に対する並列性は，AND 並列性とみなせる．

3) の計算は，元来かなりの逐次性を含んでいる．モデル拡張候補 D 中のアトムに対する包摂テストを過不足なく完全に行なうためには，包摂テストの順序を固定する必要があり，あるアトムの包摂テストは，それ以前のアトムの包摂テストが完了しないかぎり，完了できない．しかし，次の包摂テストは前の包摂テストの完了を待たずに開始でき，あるところまで進めておけるので，並列実行はある程度可能である．しかし，この包摂テストは依然として最も大きな並列実行阻害要因となっている．

非ホーン節を含む問題の場合には，複数の解を同時に探索する OR 並列性を利用することによって十分な並列効果が得られる．しかしながら，ホーン節のみからなる問題の場合はこのような OR 並列性はないので，AND 並列性を引き出さなければならない．AND 並列で台数効果を得るのは OR 並列に比べて格段に難しい．実際，MGTP の並列化の労力の大半は AND 並列化に投入された．以下に，OR 並列化方式と AND 並列化方式について述べるが，誌面の大半が AND 並列化に割かれているのは，このことを反映している．

3.6.5 OR 並列化方式

複数のモデル候補の探索を並列に行なうときには，どのプロセッシングエレメント (PE) でどのモデル候補を探索するかを指示する方法 (PE 割つけ法) が，台数効果を高める上で重要である．この指示が適切でないと，PE の負荷が不均等になり期待していた効果が得られない．ここでは実際に筆者らが実験で用いた割つけ法を二，三紹介する．

最も安易な割りつけ法は，ある PE で複数のモデル候補が得られたら，一つを残して他のすべてのモデル拡張候補の探索を相異なる他の PE に割り当てていく，というものである．しかし，これには親元の環境のコピーが必要であり，また一般的にモデル候補数は爆発的に増えていくので，プロセス間通信が多発し現実的ではない．そこで，このように無闇やたらに PE を割り当てるのを抑制することが現実的な解法となる．筆者らは，この OR 並列探索の起動を抑制する方法を，有界 OR 並列と呼んでいる．

基本的な考え方は，利用可能 PE 数を埋めるのに十分なプロセスを発生させたあとは，他 PE に割りつけることなしに自 PE 内で以降の処理を行なうというも

†1 「後件部の変数はすべて前件部に出現する」という値域限定性 (range-restrictedness) を節集合が満たすときは，生成されるアトムは変数を含まないので場合分けの際に共有変数の問題が生じず，証明木の各枝の探索は独立に行なうことができる．しかし，この性質を満たさない場合には共有変数の束縛値の一致性を検査する必要があり，単純な場合分けができず，AND 並列性と同種の問題が生じる．

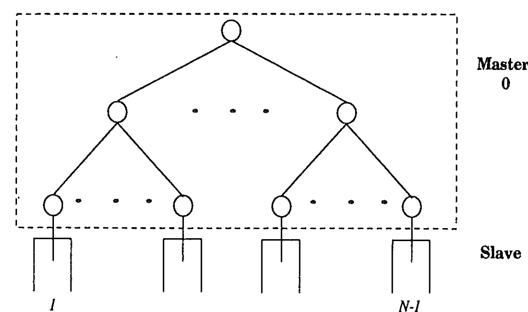


図 3.6.5 PE 割りつけ法

のである．この方法は OR 並列探索では異なる枝間では通信が生じないという性質を利用したものであり，これによって PE 間通信が軽減できる．

一つの実現法としてはマスタースレーブ方式が考えられる．マスター PE が空モデルから始めてモデル候補を拡張し，モデル候補の数が利用可能 PE 数を越えると，続きの探索をスレーブ PE に分配する，という方法である (図 3.6.5)．スレーブ PE は割り当てられたモデル候補を自分で探索するのみで，他の PE にさらにモデル候補を分配するようなことはしない．

また，マスターを利用しない実現法も考えられる．これはある深さに達するまでは，モデル候補の分岐のたびに何らかの方法で (たとえば modulo 計算によって) 負荷が均等になるように割りつけ PE を決定し，その深さ以降は自 PE 内で処理を継続するというものである．

上記二つの方法は探索木が均衡している場合に適している．この場合，どのモデル候補の探索も大体同じ計算コストになるので，PE の負荷が均等になりやすい．またこれらの方法は，通信コストを最小化できるという点で簡便な方法である．

有界 OR 並列にこだわらない割りつけ法としては，モデル候補の分岐のたびに割り当てる PE を確率的に決定する方法もある．探索木が不均衡な場合は各枝の探索コストは予見できないので，このような確率的方法が有効になる．しかし，前にふれたとおり，通信コストの点では問題が残る．

3.6.6 AND 並列化方式

一つのモデル候補の探索を並列に行なう AND 並列化では，OR 並列化に比べ, より細かな制御が必要になる．ここでは，一つの方式を提案するが，まずその方式を考える際に考慮した事項について述べる．そして，負荷分散法，粒度，並列性阻害要因とその対処法などについて触れたあと，本方式の動作概要について述べる．最後に本方式を実現する際に用いた KL1 コーディング技術を簡単に紹介する．

A. 設計指針

AND 並列化にあたって，以下の方式上のオルタナティブを検討した．

1) PE 台数にかかわらず証明が変わらない証明不変方式と PE 台数に応じて変わる証明変動方式.
2) モデル共有 (分散メモリアーキテクチャでは各 PE が同じモデル候補をコピーしてもつ) 方式とモデル分散方式.
3) マスターありとマスターなし.

証明不変方式は，モデル拡張候補 D からの Δ の取り出し順，アトムの生成順，包摂テストの順番を固定し，PE 数によらず同一の証明を得ようとするものである．一方，証明変動方式はこれらの順番を限定せず，先着順で処理する．したがって，使用する PE 数が異なると得られる証明は変わる．

証明変動方式では，超線形台数効果が得られる可能性がある一方，使用した PE 台数に見合う台数効果が常に得られるとは限らない．一方証明不変方式では，使用した PE 台数に見合う台数効果が期待できるが，それは高々線形である．

モデル共有方式の利点は，連言照合や包摂テストといった最もコストのかかる計算を最小の PE 間通信で行なえることである．一方，一つのモデル候補を各 PE に分散配置するモデル分散方式では，メモリスケーラビリティを得ることができるという効果がある．しかしながら，生成されたアトムは包摂テストのために全 PE を一順しなければならないため，PE 台数が増えるに従って通信量が増大してしまう．

マスタースレーブ方式では，逐次版 MGTP をスレーブ PE 上に置き，単にそれらとマスター PE とをスター状につなぐことによって簡単に並列システムを構築することができる．しかし，マスターの負荷をできるだけ小さくする工夫が必要となる．逐次版 MGTP をリング状に結合するマスターなしの方式も考えられるが，各 PE を協調的に制御することは難しくなる．

以下では，基本方式として採用した証明不変，モデル共有，マスタースレーブ型の AND 並列版 MGTP について述べる．ここで証明不変方式を採用した理由は，速度向上が並列化によるものか，あるいは戦略 (探索空間の変更) によるものかを，明確に区別したかったからである．

本システムでは，マスタープロセスおよび複数の生成 (G) プロセス，包摂テスト (S) プロセス，棄却テスト (T) プロセスからなり，G プロセスと S プロセス は要求駆動的に動作し，T プロセス はデータ駆動的に動作する．動作概要については 3.6.6 E. 項で述べる．

B. 負荷分散

AND 並列の実現にあたっては，G，S，T の論理的プロセスを物理的 PE にいかに配置するかが重要な課題となる．この配置法は大まかにいって，(1) 一つの PE に 1 種類のプロセスしか配置しない「機能分散」法と (2) 各 PE に 3 種類のプロセスを重ねて配置する「負荷分散」法の 2 種類がある．

(1) の方法では，時々刻々と 3 種類のプロセスの負荷比率が変動する問題では，性能を上げるのは難しい．たとえば，G プロセスが忙しい時間帯には S プロセスや T テストプロセスが配置されている PE は暇になる．このような場合には，G プロセスをより多くの PE に配置するために，すでに配置されている他のプロセスをそれらの PE から排除するような操作が必要となるが，そのタイミングや再配置の度合を適切に決めることは非常に難しい．むしろこの方法が真価を発揮したのは，性能を上げるためのパフォーマンスデバッグのときであった．各 PE の稼働状況を視覚的に表示する rmonitor で観察することによって，時間とともに変化するプロセスの負荷比率がわかり，問題の傾向のみならず並列化方式の良し悪しも把握することができた．たとえば，一つの PE だけ忙しくて他の PE はまったく稼働していない極端な場面に遭遇することもあったが，これは方式あるいは実現手法に何らかの欠陥がある場合が多かった．このようなやり方で，性能劣化の要因となるボトルネックがいくつか明らかになり，性能改善を図ることができた．

さて，(2) の場合は 3 種類のプロセスを重ねて配置するわけであるから，(1) でのそれぞれの負荷が重ね合わせられることが期待される．G プロセスが暇なときは，T プロセスが穴を埋めてくれるようなことが期待されるわけである．これはプロセスの「動的変身」による動的負荷分散効果を狙ったものであるが，この方法ではプロセススケジューリングが性能上最も重要になってくる．緊急度の高いプロセス[†1]がある PE で発生したときに，その PE では他のプロセスが走行しているためなかなか実行に移されず，その結果，他 PE のプロセスがアイドル状態になる事態が発生する．これに対処するには，最も緊急度の高いプロセスから実行すればよいわけであるが，この緊急度というのはシステム全体を見わたさないとわからないから制御は厄介なものになる．最適なスケージューリングは，机上だけではほとんど予想できないので，試行錯誤の連続で性能を上げていくことになる．このようなときに rmonitor のようなチューニングツールが有効であるのはいうまでもない．

C. 粒度

G，S，T の各プロセスの処理の単位を，1 アトム (Δ) ごととした場合の各プロセスの粒度を見積もってみよう．粒度が粗ければ並列性がでないが，細分化すると今度は分割によるオーバヘッドが生じるので，粒度の大小はシステムの性能を大きく左右する．

連言照合を行なう G，T プロセスの粒度は，節前件部のリテラル数に依存する．簡単のためリテラル数が 2 の場合を考えると，i 番目のアトムとそれ以前のアトム集合 $(1,\ldots,i-1)$ との連言照合は $i\times(1,\ldots,i-1)+(1,\ldots,i-1)\times i+i\times i$ の掛け合わせを行なうので，その回数は $2(i-1)+1$ となり，粒度は i に比例したものになる．

i は G プロセスでは M の要素数に，T プロセスでは $M\cup D$ の要素数に相当するので，G プロセスの粒度は $|M|$ に比例し，T プロセスの粒度は $|M\cup D|$ に比例することを意味する．一般にリテラル数が n のときには G プロセスの粒度は $|M|^{n-1}$ に比例し，T プロセスの粒度は $|M\cup D|^{n-1}$ に比例する．

一方包摂テストは，すでに包摂テスト済みの要素 $(M\cup D)$ に対して総当たりによる検査が必要になる．よってこの粒度は $|M\cup D|$ に比例する．

†1 そのプロセスの実行が終わらないと他の (多くの) プロセスの実行を開始できないようなプロセスは緊急度が高いといえる．

以上の考察は線形探査による操作を前提にしているので，何らかのインデックス機構を導入すれば 1 回当たりの仕事量は小さくなり得ることに注意されたい．

当初，1 アトム (Δ) を処理の基本単位とすれば十分な並列性が引き出せるものと考えていた．というのは，証明が進むにつれて $M \cup D$ が成長し，粒度もしだいに大きくなるからである．ところが，連言照合においてユニフィケーションに失敗すると，Δ に対するそれ以降の処理は行なわれなくなるので，このため負荷の不均一が生じ，$|M \cup D|$ が数千から数万になる問題では，それがより顕在化した．これとは逆に，計算量が大量なわりには $M \cup D$ がなかなか成長しないような問題もいくつかあり，基本処理単位が 1 アトムでは，十分な並列性を引き出すことは困難であることが少しずつ明らかになった．この対処法については次項**並列性阻害要因** を参照されたい．

D. 並列性阻害要因

本並列化方式には原理的に二つの逐次性があり，並列性能を追求する上での阻害要因となっている．一つは前にも少し触れたが包摂テストの逐次性であり，もう一つは証明不変を保証するための逐次性である．

a. 包摂テストの逐次性

包摂テストの逐次性は，直前のアトムの包摂テストを終了しないと，次のアトムの包摂テストが終了できないことに起因する．この逐次性は二項間の包摂関係に全順序性がないことから生じるものであり，互いに包摂し合わないアトム集合を得るには，勝ち抜き戦ではなくリーグ戦，つまり総当たりが必要となる．これは原理的な問題であり，実現手法によるものではない．

筆者らは，包摂テストを局所的包摂テスト (LS) と大域的包摂テスト (GS) に分離することにより，逐次性の低減を図った．生成されたアトムに対する包摂テストは，それ以前に生成されたアトム集合 A に対して行なわれなければならない．LS では，これをすでに包摂テスト済みのアトム集合 $M \cup D$ に対して行ない，GS ではその残り $A \setminus (M \cup D)$ に対して行なう．

現方式ではモデル共有方式を採用しているので各アトムに対する LS は他に影響を与えることもなく，各 PE 内でそれぞれ独立に行なうことができる．GS には上で述べたような逐次性があるので，パイプライン実行により並列性を抽出している (図 3.6.6)．

いま，n 個の大域的包摂テストプロセス $GS_1, GS_2, \ldots, GS_n$ が動いているものとする．各 GS_i は受けもちアトムΔ_i の処理中であるとする．GS_1 で行なわれていたΔ_1 の包摂テストが終わると，その結果が他の $GS_i (1 < i)$ に直ちに知らされる (同図 a)．各 GS_i の動きは，この包摂テストの結果によって異なる．Δ_1 が包摂されなかったときは，Δ_1 に対してΔ_i の包摂テストを行ない，包摂された場合は特に何もしない (同図 b)．いずれの場合も，次に GS_2 は Δ_2 の包摂テストの結果を他の $GS_i (2 < i)$ に知らせる (同図 c)．このように，各 GS_i の待ちは，高々(自分より以前の)プロセス個数分だけで済む．

包摂テストの実装法としては，各項を木構造として「項メモリ」に保持し，共通部分項をくくり出すことによって検索効率向上を図る方法と，各項をそのまま「リスト」形式で保持しリニアサーチによって検索する簡単な方法があるが，LS では項メモリによって，GS ではリニアサーチよって包摂テストを実現している．

b. 証明不変による逐次性

証明を不変にするためには，生成されたアトムの取り出し順序を固定しなければならない．この順序を固定することは，逐次性が発生することにほかならない．たとえば，二つの G プロセス G1 と G2 が並列に動いているとき，G1 の生成アトム集合 New_1 と G2 の生成アトム集合 New_2 は並列に生成されるが，早くできたアトムから取り出すわけにはいかない．どちらが早くできるかは非決定的であるから，あらかじめ決められた順序で取り出さなければ，実行のたびに証明が変わり得る．さて，あらかじめ New_1 から取り出すことになっていたとすると，New_1からの取り出しが完了しなければ New_2 から取り出されることはない．したがって，New_1 の生成が何らかの理由で遅れた場合，New_2 がすでに生成されていたとしても，次のアトムを取り出すことができないので S や T プロセスが動けない状態になってしまう．もちろん，G1 の実行が滞っていたとしても，G2, G3, ... と次々に G プロセスを並列実行させれば見かけ上暇な PE をなくすことはできるが，これは無駄なアトムの生成を引き起こす可能性があり，遅延生成の考え方に反する．

実際に走行させてみても，ある PE の G プロセスが生成したアトムを取り出している間，他の PE の G プロセスは，指定個数のアトムを生成したあとに，

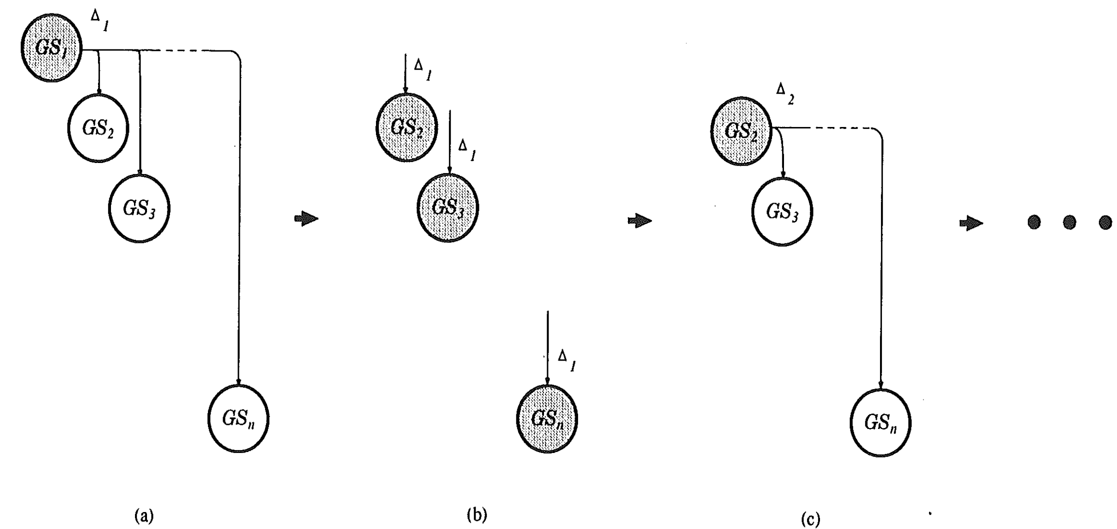


図 3.6.6 大域的包摂テストのパイプライン実行

待ち状態に入り，各 PE の G プロセスの稼働状態が，先頭 PE から 最後 の PE へと順次推移していく尺取り現象がみられた．

そこで上記の逐次性を低減するために，G プロセスにおける仕事の単位を k 分割し，連言照合 $CJM_{generator}(\Delta, M)$ を $|M|/k$ (k は定数) 個の G プロセスで行なうようにした．このようにすることにより，G プロセスの粒度がより細かくなり，かつ不均一さも顕著ではなくなり，PE の平均稼働率が 90 % 以上という性能向上に成功した．

E. AND 並列化方式概要

図 3.6.7 は，AND 並列化した際の各プロセスの結合関係を示したものである．中央のマスター (M) プロセスを介して，上部の Generator プロセス群と下部の Tester プロセス群が連結している．

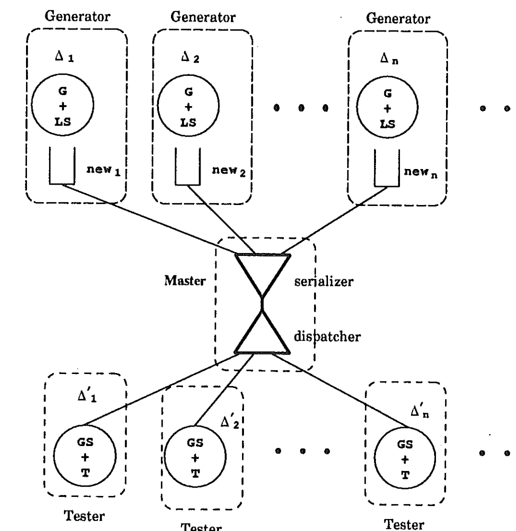


図 3.6.7 MGTP の AND 並列化

マスタープロセスの基本的な仕事は，Generator プロセスが生成したアトム (LS 済み) を Tester プロセス (GS も行なう) に分配することである．各スレーブプロセスは，暇になると次の仕事をもらうためにマスタープロセスに要求を出す．マスタープロセスは，1) 要求に先着順に応答し，2) 各スレーブプロセスが同じ仕事をしないように仕事を排他的に割り当て，3) Generator プロセスの生成したアトム数と Tester プロセスに分配したアトム数の差 (生成したが Tester プロセス によって消費されていないアトム数) を一定に保つことにより，Generator プロセスの走行のしすぎによるアトムの過剰生成を抑制する．1) により自然に動的負化分散も行なわれる．

要求 1 回当たりのマスターの仕事量は，基本的には生成アトム Δ の配分しか行なわないので，非常に軽い．

各 Generator プロセスが生成したアトム列 new_i は，serializer によって順序づけられる．これは証明を PE 数によらず不変にするためである．この serializer を merger に取り替えるだけで証明変動方式の MGTP を容易に作ることができる．serializer で整列したアトムは，dispatcher によって Tester プロセスに分配される．

Generator，Tester プロセスの大まかな振舞いは次のとおりである．

Generator: マスターから指示のあった範囲 (前述の k 分割された仕事の単位) で連言照合を行なってアトムを生成し (G)，これらのアトムに対して局所的包摂テストを行なう (LS)．包摂テスト済みのアトムはマスターの serializer に送られる．

Tester: マスターから与えられたアトムに対して大域的包摂テストを行ない (GS)，包摂テスト済みのアトムについて棄却テストを行なう (T)．

包摂テストを専門に行なう S プロセスが，Generator プロセスにおける LS と Tester プロセスにおける GS に分離されている点に注意されたい．S プロセスを LS と GS に分けることは，S プロセスの逐次性を低減するほかに，LS において生成要素のふるい落としが行なわれることによる PE 間通信の低減も図れる利点をもつ．

F.　KL1 コーディング技術

MGTP の並列化にあたっていくつかの KL1 コーディング技術が比較検討された．

a.　インプリシット・コピーイング

情報を PE にまたがって共有したい場合 (モデルコピー方式)，(1) その情報の複製／転送をプログラムの上で，ストリームを用いて陽に記述するか，もしくは，(2) 構造データの頭 (D-リストの先頭) を各プロセスに共有させ，必要な項目までたどるのはファームレベルの動作に任せるか，のいずれかがある．

(1) の方式が優れていると考えられるのは，転送すべき情報のひとまとまりが本当に一度に PE 間をわたることが保証されているときである．特に，ある PE に所在している構造データを他の複数の PE に転送したいとき，ストリームでチェイン接続した PE を順番に転送するように書けば，発信元の PE に対するメモリ競合の問題がなくなる．さらに，発信元が複数あれば，異なるタイミングで複数の情報が異なる PE 間で同時に転送でき，ネットワークのスループットが向上する．一方，(2) の方式によると，発信元の PE に対するメモリ競合の問題があり，台数の増大とともにボトルネックが生じやすくなる．

ところが，KL1 では 2 以上の深さをもつ構造データ (ベクタ) が一度に転送されることはないので，(1) の方式でも必ずしもメモリ競合の問題がなくなるとはいえない．構造データを平坦な別のデータ形式 (たとえばストリング) に変換して転送すればよいが，変換のコストが無視できない．

(2) の方式は，元来共有メモリ型のアーキテクチャに適したものであるが，分散メモリ型の PIM のアーキテクチャにおいても (1) に比べてきわめて悪いということはない．むしろ，通信記述に伴うオーバヘッドが軽減されると考えられる．また，(2) の方式では，PIM のようなローカルメモリ型のマルチプロセッサ上であたかも共有メモリを使用しているようなコーディングが可能となる．これは，KL1 プログラミングの大きなメリットであり，これにより，コーディング作業量および開発期間が大幅に短縮された．

b.　優先度制御

一つの PE に異種のプロセス群を配置する負荷分散方式では，プロセスのスケジューリングが重要であると述べた．しかし，KL1 のシステムスケジューラを直接操作することはできないので，ゴールの優先度を制御することによりスケジューリングを間接的に行なっている．

さて，図 3.6.7 に示した AND 並列化方式では，実際には Generator は，モデル拡張 (G) と局所的包摂テスト (LS) をまとめて行なう一つのプロセスとして，一方 Tester は，大域的包摂テストを行なう GS プロセスと棄却テストを行なう T プロセスの二つのプロセスで構成されている．これらのプロセスにどのような優先度を与えるかについて考えてみよう．Generator プロセスがアトムを生成しないと他のどのプロセスも動けないという点で Generator プロセスの緊急度は高いが，不要なアトムを作りすぎると遅延生成の原則に反することになる．GS プロセスは，ある要素の包摂テストが終了しないと他の PE で行なっている包摂テストが終了しないし，棄却テストも開始できないので，これも他のプロセスへの影響が大きい．T プロセスについては，それが終了しないからといってほかのプロセスの実行が開始できないというわけではないので緊急度は低いが，いわゆる全計算の終了判定を行なっているので実行をいつまでも後回しにするわけにもいかない．

また，実際には複数の Generator プロセスと複数の GS および T プロセスを一つの PE に割りつけているので，同種の複数のプロセス間の優先度も考えなければならない．これには，以下に述べるスライド優先度割当て法を用いている．

すなわち，同種のプロセス間では，個々のプロセスは降順の優先度を付加されて起動される．ある優先度 Pri をもつプロセスの計算が終了したら，Pri 以下の優先度が付加されたプロセスの優先度を一つずつ上げる．これにより，最低優先度 Pri_{min} をもつプロセスの優先度も一つ上がり，次に起動するプロセスの優先度として Pri_{min} を割り当てることができる．このように優先度を再利用することによって，有限の優先度しか提供されていない KL1 処理系で，あたかも無限の優先度があるかのような環境を模擬することができる．

以上のことを踏まえ，優先度は高い順に 1) GS プロセス，2) Generator プロセス，3) T プロセスとしている．ただし，このような優先度では GS プロセスと Generator プロセスのみで暴走する可能性がある．そこで，1 PE 当たりの未終了の T プロセス数がある閾値を越えると，これらの T プロセスのいくつかの実行が終了して閾値の範囲に収まるまで GS プロセスの起動を遅らせるようにしている．このようにして，GS，Generator プロセスによる暴走を防いでいる．

c.　要求駆動制御

KL1 には元来，必要なデータが揃うと実行を開始する同期機構が備わっており，要求駆動を実現するには適したプログラミング言語である．

遅延モデル生成の心は，新しいアトムが必要なときにのみ生成を行なう，つまり，G プロセスを動かせばよいというものだから，これは G プロセスを要求駆動に動作させることにより実現できる．

G プロセスにおいて，Δ とそれ以前のアトム集合 M との連言照合を行なう最内側ルーチンは，要求駆動制御になっていなければ，大体次のような KL1 プログラムになる．

本ルーチンは連言照合に必要な Δ と M の要素の重複を許した組合せパターンと節パターンの組 (Cmb) を受け取る．

```
g([], Out) :- % すべての連言照合が終了したら,
    Out = [].   % 出力ストリームを閉じる.
        g([Cmb|Cmbs], Out) :-
              % あるパターン Cmb について
    tryCJM(Cmb, Rst),  % 連言照合を試みる.
      (Rst = success ->  % 成功したら
            newConc(Cmb, Conc),
              % 後件部 Conc を生成して
            Out = [Conc|Out1],
              % 出力ストリームに流す.
            g(Cmbs, Out1);
       Rst = fail -> g(Cmbs Out)).
              % 失敗したら何もしない.
```

これに，要求を伝達するストリーム Dmds を加えて，

```
g([Dmd|Dmds],[Cmb|Cmbs], Out) :-
                    % 要求があったら
tryCJM(Cmb, Rst), % 連言照合を試みる.
            ...
```

という風にすると要求駆動制御になる．

実際には，このプログラムのように本当に必要になってから作り始めたのでは遅いので，ある程度のバッファリングが必要である．バッファが空にならないように，かといって作りすぎないような微妙な制御が必要になる．この制御もマスタープロセスの大きな役割である．

3.6.7　実行結果

以上述べてきた MGTP の AND/OR 並列化方式の効果を PIM/m 256 台を用いて計測した．

A.　OR 並列化

OR 並列実行による速度向上効果を見るため，pigeon hole と Bennett 問題 [50] の性能測定を行なった．pigeon hole 問題は，定理証明の分野においてベンチマークとしてよく用いられている．Bennett 問題

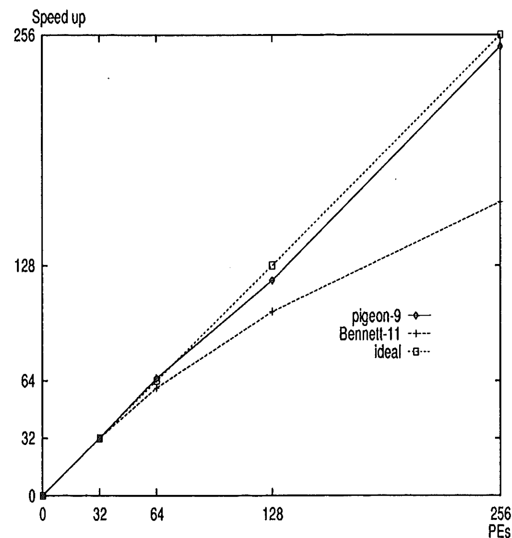


図 3.6.8 速度向上比 (OR 並列)

図 3.6.9 速度向上比 (AND 並列)

は有限準群に関するもので，ラテン方陣の数え上げ問題と見なすことができる．いずれも生成されるアトムは基底アトムのみであるが，OR 分岐の数が組合せ的に増大する問題である．

証明図の形状に関していえば，pigeon hole 問題はよく均衡しており，Bennett 問題は均衡の度合が悪い．したがって Bennett 問題では，探索する証明の枝が不均一なので，pigeon hole 問題に比べると負荷を均等に割り当てるのが困難である．図 3.6.8 は，速度向上比をグラフ化したものである．pigeon hole 問題 (hole 数 9) ではほぼ理想線に近く，また負荷の均等割当が困難な Bennett 問題 (ラテン方陣の位数 11) でも 170 倍程度の良好な速度向上が得られている．

B. AND 並列化

図 3.6.9 は，[58] に述べられている分離の規則に関する問題 (condensed detachment problem) 112 題のうち，5 題に対する PIM/m での速度向上比をグラフ化したものである．分離の規則に関する問題というのは，いくつかの単位正節 $true \rightarrow p(A)$ と分離の規則を表わす混合節 $p(X), p(i(X,Y)) \rightarrow p(Y)$ と一つの単位負節 $p(G) \rightarrow false$ からなる．混合節の後件部は連言を含まないので，枝分かれによる OR 並列性はまったくない．

これらの問題の証明は，OTTER [59] でさまざまな戦略を用いて試されている．これらの戦略には 1) 項の重さによるソーティング，2) 1) と幅優先探査を混合して用いる，3) ある種のパターンを含むアトムを捨てる，などがある．MGTP では，幅優先探査を基本として，項の重さによる足切りと 3) を用いて証明を行なった．

MGTP の単一 PE での性能は，遅延モデル生成アルゴリズムによる計算量の低減もあり，Sparc-2 上の OTTER に比べ 2 ～ 5 倍程度高速である．

問題 44，49 の証明にかかる時間は，32 PE でそれぞれ 9700 秒，18600 秒であり，問題 79，82 の 2500 秒，1900 秒に比べかなり時間を要している．問題 22 の証明時間は 8600 秒と問題 44 と大差ないが，$|M \cup D|$ の大きさは，問題 44 では 15100 に対し問題 22 では 36500 と倍以上である．

台数効果をみると，問題 44，49 では約 230 倍以上，一方問題 22，79，82 では 170 から 180 倍程度の速度向上が得られている．

問題 44，49 の台数効果が他の問題に比べて優れていることから，時間を要する問題ほど，また $|M \cup D|$ の成長速度 ($|M \cup D|$/sec) が遅い問題ほど良好な台数効果が得られていることがわかる．$|M \cup D|$ の成長速度が速いと，1) 大域的包摂テストにかけられるべきアトムと 2) 各 PE で蓄えられるアトムの単位時間当たりの数が多くなる．これによって，1) 大域的包摂テストの逐次性が顕著になり，2) 単位時間当たりのアトム転送頻度が多くなる．この逐次性と転送ネックが，$|M \cup D|$ の成長速度が速いと並列性能が劣化する原因として考えられる．

全体として，128 PE 以上では，グラフの傾きがやや緩やかになる傾向が見られるものの，256 PE までは速度向上の飽和現象は見られない．ほぼ，線形な台数効果が得られ良好な結果を示しているといえる．

3.6.8 むすび

第五世代コンピュータ プロジェクトで開発した並列定理証明系 MGTP について，並列化方式とその実現手法を中心に紹介した．本研究では，大規模並列定理証明系の優位性を示すため，PIM 上で「線形に近い台数効果」を達成した．これまで逐次マシンでは計算時間・メモリ容量の観点から不可能であった「ハードな問題」に挑戦し，これを解くことを第一目標に掲げた．主に，分離の規則に関する問題や準群問題のような数学的定理証明の問題に取り組んできたが，いわゆる未解決問題を解くことにも成功するなど，当初予想してなかったような成果が得られた．しかし，並列化方式の観点からは，解決すべき課題もいくつか浮き彫りにされてきている．

現在の AND 並列化方式は 256 PE 上で 200 倍以上の速度向上を達成し，かなり良好な並列性能を示しているが，包摂テストの逐次性の問題は完全には解決しておらず，スケーラビリティの点で限界がある．今後，項の重さによるソーティング等の機能を組み込む予定であるが，同種の問題が予想される．また，ホーン節，ノンホーン節対応にそれぞれ個別に AND 並列化方式および OR 並列化方式を開発したが，一般にはこれらの節が混在する場合が多く，AND 並列性と OR 並列性の両方を同時に取り出すことができる AND/OR 統合方式を確立する必要がある．

MGTP 研究の最終目標は，論理プログラミング技術と自動推論技術を融合することである．MGTP は KL1 が失った全解探索機能を自然な形で回復しており，従来のトップダウン型とは異なる，ボトムアップ型をベースとする新しい論理プログラミングパラダイムとなる可能性をもっている．今後，AI 応用向けに MGTP の機能拡張を進めるとともに，MGTP の特長を生かした応用領域を開拓していく予定である．

3.7 自然言語解析

3.7.1 はじめに

人は言葉を使ってものごとを理解したり，他の人に情報を伝えたりする．ヘレン・ケラーの物語のなかで，視覚も聴覚も失った子供時代のヘレン・ケラーが水に手を触れながら「water」という言葉の意味を知るという感動的なエピソードがある．このときヘレン・ケラーが発した「water」という言葉は「水をください」という意味でも「これは水です」という意味でもなく，「ああ，水だ」という人間が何かを認識したときに感じる気持ちの表われである．

人間が理解している世界は言葉と深いつながりがある．現代人が認識している現実は，人体の五感で受け取る情報だけでなく，テレビや新聞や本やラジオや電子メールなどのメディアから毎日受け取っている大量の言葉が大きな役割をはたしている．そして人間自身もこういったメディアの一つである．

自然言語とは，人間が毎日使っているこういう言葉のことである．プログラミング言語や記号論理などの人工言語と対比して，こういう呼び方をする．人工言語は，書いてあることだけを見れば意味が完全にわかるようにつくられているのに対して，自然言語ではそうはいかないことが多い．自然言語にも言葉の使用法に関する決まりがたくさんあるが，厳格な人工言語の文法と柔軟な自然言語の文法は本質的に違う．また，自然言語では状況に応じて同じ表現でもいろいろな情報を伝えることができる．

こういった変幻自在な自然言語の柔軟さは機械で処理するときにはやっかいなものだが，人間のコミュニケーションの道具として見たときには自然言語の優れた特長なのである．そして，そういう特長を可能にしているのは人間の認知能力やコミュニケーション能力である．このため，自然言語処理の研究は，自然言語の文法的構造の解明だけでなく，人間の言語認知能力やコミュニケーション能力の解明を含む必要があるということが以前より指摘されている．

しかし，これは現代の技術レベルをはるかに越えたスローガンであるといわざるをえない．現代の自然言語処理の研究は，人間とは似ても似つかない処理モデルの上で進められているが，工学的立場からは妥当な

選択であるといえる．実際，これまではこのような処理モデルのもとで自然言語処理技術は大きな進歩をとげてきた．

しかし，自然言語処理は実用化技術の面では，現在「泥沼」と表現されるような苦しい技術的壁に突き当たっている状況にあるのも事実である．「近い将来に，本当に実用的な自然言語処理システムが開発されるでしょうか？」という問に対して，自然言語処理研究者たちは否定的な答えしか返せないのである．

そのような自然言語処理の研究にとって，いま一筋の光明が並列処理である．従来の工学的立場は逐次処理マシンを前提にしていた．本物の並列マシンを目の前にすると，これまで考えもしなかったような処理方法にも挑戦してみようという意欲がわいてくる．正直にいうと，研究者としてはこれが最もおもしろい．並列処理の観点から考えていくと，人間の認知能力やコミュニケーション能力のモデルに少しは接近できそうに思えてくる．

もちろん，並列処理は処理を速くするのが第一義的な目的である．しかし，本稿で一番伝えたいのは，並列処理の楽しさと，この技術が自然言語処理に与えるブレークスルーへの手掛かりである．

3.7.2 自然言語解析とは

自然言語解析とは，人間が発した言葉が表わしている情報を，コンピュータに処理できる形式として取り出す処理である．自然言語の解析は，形態素解析，構文解析，意味解析などの処理から構成されている．

A. 形態素解析

形態素解析とは，入力された文字の並びから単語の並びを選び出す処理である．例をあげると，(すもももももももものうち）という文字列を（李，も，桃，も，桃，の，うち）という単語の列として認定する処理である．

この処理は一見簡単そうだが，実際はなかなか難しい．仮名漢字変換はすでに一般になじみ深い技術になっているが，この技術は形態素解析を含んでいる．形態素解析の難しさを理解するには，少し長めの文を人間の介入を一切入れないで仮名漢字変換させてみるとよいだろう．

文字列から単語を切り出す可能性をすべてにわたって調べると，莫大な数の検査を行なう必要がある．このため，普通の形態素解析ではさまざまな手段を使って探索する領域を減らす工夫を行なっている．

文節の区切りを決定する代表的なものとして，最長一致法といって，文節の長さが最も長くなるように選んでいく方法がある．「さらしなそばはおいしい」という文字列は，「さら」「しな」「そば」「しなそば」などの単語に区切られる可能性をもっているが，左から右に一番長い区切りを選んでいくと，「さらしなそば，は，おいしい」というように正しく分割される．

また，漢字，平仮名，片仮名，数字，句読点，アルファベットのような文字の種別を，文節の区切りを発見する手段とする方法も利用される．

形態素解析に関する文法規則もある．これは，左右に隣接する単語と単語の間の接続可能性を検査する文法である．形態素解析用の辞書には通常，この検査のための接続情報が記載されている．また，隣り合う要素だけにしか依存しない文法は，構文解析に使用されるもっと遠く離れた要素の支配を受ける文法よりもはるかに効率がよいアルゴリズムが知られている．

こういったさまざまな工夫を複合的に利用することにより，形態素解析はかなり高速で高精度の結果が得られるようになっている．しかし，このような工夫は本来出てきてほしい解を誤って削ってしまう可能性も含んでいる．また，形態素解析文法は，構文解析の情報が利用できれば本来は必要ないものである．

また，単語とはいかなるものであるかというもっと難しい問題もある．たとえば，「新世代コンピュータ技術開発機構」という文字列を一つの単語と見るか，それとも構文構造や意味構造をもつものと見るかという問題である．これは辞書に記載されていない単語をどう扱うかという問題や，そもそもそれが登録されていない単語だということをどうやって認識するかというような問題とも関係する．

B. 構文解析

構文解析は，従来はどちらかというと，自然言語処理というよりはコンパイラのような人工言語処理か，あるいは言語学で利用されるような概念であった．ちなみに，機械翻訳システムなどの商品化されている自然言語処理システムの多くは意味解析主導の解析方式をとっており，実際には構文解析は補助的にしか行なっていないことが多かった．しかし，言語学の構文論や工学分野での構文解析手法のめざましい発達とともに構文解析が実用技術となってきており，実用的な自然言語処理システムにも取り入れられるようになってきている．

言語学の構文論とは，単語の並び方の観点から正しい文と間違っている文を判定する理論である．

「ルネサンス，は，変化，の，時代，であった」という文は構文的に正しい．しかし，「ルネサンス，であった，時代，の，は，変化」は構文的に間違っている．構文的に正しいと判断されるには，単語の並び方が規則にあってなければならない．

辞書に記載されている単語の品詞の観点から単語の並び方を決める規則を構文規則という．「ダビンチ，は，変化，の，時代，であった」は意味的には奇妙だが，「ルネサンス」も「ダビンチ」も品詞は名詞であり，単語の並び方は規則にあっているので，構文的には完全に正しい文である．

構文解析という処理は構文規則を利用して，どれが主語でどれが目的語であるかというような関係や，単語と単語の間の修飾関係などを判断する処理である．構文解析も形態素解析と同様にアルゴリズムの研究が進んでおり，効率的なアルゴリズムが知られている．

しかし，自然言語の文は構文的に非常に多くの曖昧さがある．たとえば，「ためらいがちに，かけた言葉に，驚いたように，振り向く君に，季節が，頬を染めて，過ぎてゆきました」という文は，「〜に」という文節が四つあるが，意味や常識を除外して構文規則だけで修飾関係を決めると，それぞれが右側にある動詞すべてに係りうることになる．そうすると，動詞は5個あるので，$5\times4\times3\times2=120$ 個の解が出てきてしまう．構文規則を精密化することによって解の数をもう少し減らすことは可能だが，少し長い文を解析しようとすると組合せの数が爆発的に増加し，どうしても解の莫大な数になってしまう．構文解析の解の数を減らすためには，意味的情報などを利用しなければならない．

C. 意味解析

自然言語解析の目的は，言葉の意味をコンピュータが処理できる形式にすることであるが，意味解析はまさにそのアウトプットの意味構造を構成する処理である．意味構造の記述形式としては，論理式や意味的要素の関係をネットワーク構造で表現するなどの方法がある．

意味解析には，意味的な適切さを判断する処理も含まれている．意味的な適切さの判断では，もの概念の上位・下位・同義などの関係や述語概念の意味役割などを利用する．たとえば，「食べる」という述語概念は行為者という意味役割をもつ．この行為者という役割をはたすものが，動物というもの概念として判断できなければならないとする．これを，次のように書くことにする．ただしXは変数である．

食べる ([行為者=X:動物])

変数Xに「太郎」という人間が代入され，動物が人間の上位概念であると，「太郎が食べる」という文が意味的に認められることになる．

多義語や同音異義語を翻訳するときなどに，意味判断は特に切実な問題となる．たとえば，「かける」という動詞が，水をかけるのか，洋服をかけるのか，野原をかけるのか，数と数をかけるのか，命をかけるのか，何かにお金をかけるのか，博打に何かをかけるのか，電話をかけるのか，アイロンをかけるのか，レコードをかけるのか，お皿がかけるのか，魔法をかけるのか，などというような違いを認識することは翻訳には不可欠である．

実際の意味解析システムの構築では，莫大な量の概念の記述やそれに対応する意味情報を辞書へ記述することによって，意味情報をあらかじめ準備しようとするのが普通である．しかし，人間の間のコミュニケーションでは，この概念の枠組み自体が文脈や話題や状況や文化や分野などに依存して変動するのが普通であり，あらかじめ用意された枠組みで意味処理をしようとすると陳腐な結果になることが多い．

また，意味解析は構文解析の結果を選別する機能も担っているが，この構文構造は意味的に正しいがこれは正しくないというような判断を行なう基準が明白に存在しているわけではない．たとえば，「プログラムがメモリを食う」とか「車がガソリンを食う」というような表現は，行為者が動物ではないので，前の「食べる」の意味判断の条件では不適格になってしまう．

それでは，このような例に対応するために，動詞の辞書に構文関係の要素との意味関係を記述しておけばよいかというとそう単純ではない．たとえば人間と虎

はともに動物であるが，虎は人間を食べるが人間は虎を食べない．このように，文の意味的適切さは主語と目的語の関係などによってダイナミックに変化する．

意味とは何か，そもそも記述可能なものなのかという疑問は未解決の問題として残っている．

3.7.3 自然言語を並列処理する理由

自然言語処理一般について簡単に説明してきたが，本稿の主眼はもちろん並列処理である．

並列処理ソフトウェア技術はそれ自体非常に難しい技術である．自然言語処理はすでに難しい課題をたくさん抱えているので，わざわざ並列処理の問題までもち込む必要性はないのではないかと考えるのは自然である．それでもなお，自然言語を並列処理する意義は何なのであろうか．

A. 並列処理はおもしろい

従来の処理方法を並列処理の観点から見直してみると，これまでよりもっと自然なやり方が見えてくる．これがおもしろい．そして実際に並列マシンを使って並列処理プログラムを動かしてしてみると，その強烈な計算パワーに次第に夢中になっていく．本当である．カラオケのアンプのボリュームを上げていくような力の増幅の快感がある．

並列処理のおもしろさを反省してみると，その源泉はいろいろな自然現象をコンピュータ上でシミュレートできることにあると思う．並列処理ソフトの開発は，お手本がないので尻込みする人が多いが，実際にやってみると，自然界のさまざまな現象をモデルにすることで，逐次処理よりもむしろ自由な思考が可能になる．ぜひチャレンジしてみてほしい．

B. 並列処理が示す自然言語処理のブレークスルー

しかし，並列処理は楽しいというだけでなく，実用的価値もちゃんとある．それは並列処理からくる自由を利用して，自然言語処理が突き当たっている壁を突き破る方法を模索できることである．

人間が易々とやってしまう自然言語の理解が，コンピュータにはなぜそれほど難しいのだろうか．そして，並列処理はどのようにして，そういった壁を突き破る手段になりうるのだろうか．まず，現在の自然言語処理が突き当たっている壁の一端を紹介する．

a. 自然言語処理が突き当たっているさまざまな壁

(1) 大規模化の壁

自然言語処理では，小さな玩具的システムではさまざまな先進的試みが可能なのだが，実用化しようとしたとたんに大規模の障壁に阻まれることになる．この障壁の問題点は，開発の困難さと計算がたいへんだということである．

辞書や文法を大規模化する目的は，解析精度を精密化することにあるのだが，一貫性を保ちながら辞書や文法の記述を蓄積していくことは非常に難しく，努力がむくわれないことが多い．

そういう状況を一層複雑にしているものに文法規則の制御の問題がある．実用システムでは，文法規則が許容するすべての解を計算していると時間がかかりすぎるので，文法規則の適用方法を手続きの付加によって制御するのが普通である．たとえば，宣言的だといわれる論理型言語による構文解析システムでさえ，構文規則のいたるところに補強項と呼ばれる手続きを記述するのが普通である．しかし，こういう制御はやっかいな副作用をもたらすことも多い．特に文法が大規模化したときに開発が泥沼化する原因の一つが，こういう制御の管理が人間の手に負えなくなることである．

(2) 自然言語の曖昧さ

自然言語では，一つの単語が使用される状況に応じて非常に多様な意味をもつ．例をあげると，日本語の「前」という単語を英語に翻訳しようとすると，front, before, last, preceding, previous, former, under, ago, once,in front of, to, ..., などなど微妙に意味や用法の異なる単語に訳し分ける必要がある．これは日本語が英語に比べて特別に曖昧なわけではなく，英語の単語を日本語に翻訳するときも同様の訳し分けが必要になる．

自然言語の単語は生きており，その意味は厳密には使用されるたびに微妙に異なるといってよい．生きている言葉を辞書に完全に記述することは本質的に不可能なのである．自然言語処理の研究では非常に大規模で精密な辞書を作成して，単語のあらゆるニュアンスの意味を使用される状況とともに記述しようと試みてきたが，このような大規模な辞書の使用は，逆に，辞書に記述された数々の意味のなかからいかにして最も適切なものを選択するかという別の問題を引き起こす．

このような問題は，構文規則や意味構造の研究などについても同様である．

人間がこのような微妙なニュアンスの違いを認識できるのは，言葉として現われている情報だけでなく，その言葉が使われている状況や文化的背景や，相手の表情や語り口などのような多様な情報チャンネルを通じて，情報を総合的に利用しているからであると考えられる．

(3) 間違いや省略を含んだ表現の処理

放送局のアナウンサーのような特別の訓練を受けた人でも，話している内容を書き留めてみると驚くほど非文法的な表現をたくさん使っている．それにもかかわらず，人間はちゃんと情報を受け取ることができる．自然言語処理を実用技術にするためには，間違いや省略などがあっても適当に補正する能力をもたなければならない．

人間の言語理解は，相手が言いたいことを推定する過程であるともいえる．すなわち，言語理解には文法的判断や意味的な判断だけでなく，同時に相手が伝えようとしている情報内容を推理する能力も必要であり，この判断と推理の両者の統合的処理ができなければならない．

b. 壁を突き破る技術としての並列協調

ここであげた規模の問題，曖昧性の問題，間違いや省略の問題などに対して並列処理は強力な武器になる．ここで鍵になるのは「並列協調」である．

たとえば，曖昧さの問題や計算量の問題の源泉は，自然言語処理のそれぞれの処理が利用できる情報が，別の情報チャンネルに対して閉じていることにある．形態素解析では，最長一致法や字種に違いや接続関係などを使って曖昧さを削ろうとしていたが，こういう処理は構文規則からの制約がフィードバックされれば不要である．つまり，形態素解析と構文解析との協調があれば，いらない処理をいろいろやっているのである．

これは規則の制御の問題に関しても同様である．規則の制御に利用されている情報は，他の処理レベルからの情報を得られれば不要なものが少なくない．また，構文解析と意味解析の関係を考えてみると，構文解析でいったん大量の解を生成してしまってから意味的判断で解を削るのは明らかに無駄である．形態素解析や構文解析や意味解析などが並列協調すれば，曖昧性の爆発を余計な処理を混入させることなく未然に防ぐことができ，したがって計算量を無用に増やすことを押さえられるはずである．

当然であるが，並列処理がすべての問題を一網打尽に解決するわけでない．辞書や文法の開発の問題には，息の長い地道な研究の継続が絶対に必要である．しかし，並列処理は文法開発にとっても福音をもたらす．並列処理はぜいたくが許される．あとで無駄になるかもしれない処理も，けちけちしないで，並行してどんどん計算してしまってもかまわない．つまり，逐次処理システムの文法開発で苦労している文法規則の制御を，あまり気にしなくて済むという利点もあるのである．

ただし，並列処理だからといって，まったく制御がいらないわけではない．効率的に処理を行なうには優先度の制御が必要である．しかし，並列処理では優先度制御の意味を少し違った見方で実現できる．筆者は，並列処理における優先度制御を，自然界における生存競争をモデルにして考えてみた．この優先度制御は，優先度制御と負荷分散手法を組み合わせたもので，競合関係にある規則たちが互いにプロセッサなどの計算機資源をめぐって競争を行なうので，「競争に基づく優先度制御」と呼んでいる．

システムの頑強性の問題は，実際の大量の例文から，既存の規則の適用範囲外の現象をこつこつと集めて辞書や文法規則の整備を進めていくことがまず先決問題であり，これらに対する地道な努力の必要性をまず強調しておかなければならない．

しかし，もう一方で必要なのは，大づかみに相手が言おうとしていることを推定することである．個々の言語表現は，大きな談話の流れのなかで一定の役割を担っているものとして解釈されるのが自然であり，たとえ文法的に間違っていても，表現からその時点で必要とされる情報が抽出できさえすれば解析は可能なのである．

このような推定の処理は解析とは逆向きの処理になる．解析は入力された文字列を文として判断する処理だが，推定は入力された文字列が，一種のパターン処理としてどのような文であるかということを調べる処理である．しかし，推定された文の種別からそれが要求する情報を抽出するには解析が必要なので，結果として推定と解析は協調して処理されるべきである．

3.7.4 並列協調の実現方法

並列協調が実現できれば，自然言語処理の壁を突き破る技術になるということを説明してきたが，次に並列協調の実現方法について説明する．

筆者らが並列協調の実現原理としたのは型理論である．型理論は洗練された美しい理論である．型理論は，プログラムの仕様の検証やオブジェクト指向やデータベースやプログラミング言語自体の原理など，コンピュータサイエンスのさまざまな分野で注目を集めている理論である．そして，この美しい型理論が並列協調の原理としても使えるのである．ただし，型理論そのものについては詳しくは解説しない．興味のある方には参考文献を参照してほしい [60, 61, 62].

A. 並列協調は何が難しかったのか

並列協調の実現方法の詳細に入る前に，これまでの技術的問題点について説明しておく．

自然言語解析において，形態素解析や構文解析や意味解析などの処理をそれぞれ独立した処理にするのではなく，全体を統合的に処理すべきであるという主張は，そう新しいものではない．

しかし，そういうやり方が自然言語処理の主流にならなかったのにもわけがある．その理由がモジュール化の問題である [63]．大規模ソフトウェアの開発は，より簡単で小規模のモジュール群に還元して行なうのが今日の常識である．しかし，モジュール化と処理の統合は対立するのである．モジュールは内部の情報を隠蔽する．モジュールにとって外部とのかかわりは入力と出力だけである．並列協調を実現するには，処理は常に外部とのかかわりをもちつつ進行する必要があるので，どの処理も独立したモジュールにしてはならないのである．したがって，並列協調システムを具体化するためには，まったくモジュール化を行なわないで大規模で複雑なソフトウェアを開発する方法論をもたなければならなくなる．型理論がこの方法論の基礎になる．

ただし，モジュール化しないというのは処理方法に関することであり，自然言語の文法の研究として形態素解析，構文解析，意味解析に分離して，それぞれに現われる現象や規則を研究することは意義のあることである．

この辺の事情は，西洋医学と東洋医学の違いと共通するところがある．西洋医学では人体を部品の集まりと考え，それぞれの部品の機能を分析するが，これはソフトウェア開発におけるモジュール化に対応する．これに対して東洋医学では，人体や病気を常にからだ全体のものとしてとらえるが，こちらは筆者らの並列協調システムの考え方に近い．人体の臓器などが本当に機械の部品のようなものかどうか疑問であるが，科学的分析には西洋医学の方法論が圧倒的に有利である．

筆者らのシステムでも，文法の開発やデバッグでは形態素解析，構文解析，意味解析を分離した形で実施している．最終的に，そうして開発された文法や辞書を混ぜ合わせて解析を実行すると，自動的に並列協調が実現されるのである．

B. 型理論と自然言語解析

型理論とはどのようなものか，型理論がなぜ自然言語解析に使えるのかということを簡単に説明する．

型の概念を説明するために，分数「1/2」は数だろうか，ということについてちょっと考えていただきたい．もちろん，「1/2」は数ではあるが，「それと2をかけると1になるような有理数」という記述につけられた名前だという見方もできる．よく考えてみると，計算で「1/2」を扱うときは常にこの記述にパラフレーズしていることに気づく．つまり，この記述は「1/2」に対する可能な操作を余すことなく伝えてくれるのである．

この記述のなかにでている「有理数」というのも，さらにまた有理数の公理系という記述の集まりにパラフレーズできる．このパラフレーズによって，有理数とはどういう操作対象なのかという記述が「1/2」に継承されることになる．

型とは，「1/2」とか「有理数」のような操作に関する記述の集まりにつけられた名前のことである．ある‘もの’が「有理数」という型として判断されると，そのものは有理数に対して可能な操作がすべてできるということを意味する．

ここの操作という概念は，別のいろいろな概念に呼び変えることができる．たとえば，プログラムと呼び変えると，型の判断は，あるデータがどのようなプログラムで処理されるべきかということを示すことになる．また，構文規則と呼び変えると，ある単語が構文規則の対象としていかなるものであるかということを

示すことになり，型は単語の品詞を示すことになる．型理論とは，ものと型の間の正しい関係を判断するための理論である．

C. 「生きたリンク」と並列協調

システムが型推論の機能をもっていれば，データからプログラムを呼び出すことができる．これが並列協調の原理のポイントである．そして，これはオブジェクト指向のシステムの原理にほかならない．たとえば，ワープロソフトで作成した文書データは，そのワープロソフトのプログラムによって処理されるべきオブジェクトである．そしてこのとき，そのワープロプログラムが型になる．この文書データを少し修正したいと思ったとき，型推論の仕組みがあれば，文書データを指定するだけでワープロプログラムを起動することができる．ものと型の関係を“:”という記号で表わすと

文書データ:ワープロソフト

というような関係になる．この関係を型割当てと呼ぶ．次に，もっと複雑な構造の文書データを考えてみる．文書のなかにスプレッドシートで作成したグラフを張りつけたとする．このような文書データをいわゆるレコード構造，つまりラベルと値の対から構成される次のようなデータ構造をもつものとする．

[text=システムが型推論の機能をもっていれば... ,
グラフ=数値データ:スプレッドシート
]:ワープロソフト

このデータの型は全体としてはワープロソフトだが，内部構造に text というラベルがついた要素とグラフというラベルがついた要素があり，グラフというラベルの値の数値データというオブジェクトもスプレッドシートという型をもつ．こういうレコード構造を型つきレコード構造と呼ぶ．型つきレコード構造をもつデータは，内部構造の部分データをそれぞれ違うプログラムで処理することができる．たとえば，文書に張りつけられているグラフをちょっと手直ししたいなと思ったときに，その張りついているグラフを単なるグラフィックデータではなく，元のスプレッドシートプログラムを使って処理し直すことができる．これは，最近多くのオブジェクト指向システムで実現されてきている「生きたリンク」と呼ばれるものである．筆者らのシステムの自動的な並列協調の原理は，この「生きたリンク」の並列処理版である．

型つきレコード構造の要素のなかに変数を入れると，変数は通信路としての役割をはたし，より密接な並列協調が可能になる．たとえば，グラフの元になるデータが変数で，その変数が通信回線で株式市況をリアルタイムで通知するプロセスによって具体化されるとすると，ワープロの文書を開いた画面上で，グラフは刻々と変化する株式市況とともにリアルタイムでアニメーションのように動くようになる．このように，型理論を使ってデータがプログラムをコントロールすることにより，きわめて簡単に，非常に複雑な並列協調処理を実現することができる．

D. ものと型の関係の推論

型推論の仕組みについて少し説明する．型推論とは，型割当ての正しさを判断する推論である．型宣言とは，辞書や文法規則のようなものであるが，型推論は型宣言を元にして推論を進める．基本的オブジェクトに関する‘オブジェクト: 型’という型割当ての集合が型宣言である．たとえば，John : 名詞, walks : 動詞 という辞書は型宣言である．文法規則にあたるのは，型と型の間の順序関係である．

名詞句 → 固有名詞

という表記は，固有名詞は名詞句になるという構文規則を意味しているが，これを次のような型の間の順序関係に書き換えることにする．

固有名詞 < 名詞

この順序関係は，もしあるオブジェクトが小さいほうの型として判断されたならば，そのオブジェクトは大きいほうの型としても判断できるということを意味している．次の式は，文は名詞と動詞から構成されているという構文規則を表わしている．

文 → 名詞, 動詞

これは，構文規則の一番右端の要素を鍵となる要素ととらえて，次のような型の間の順序関係にする．

動詞 < (名詞→文)

ただし (名詞→文) は関数の型を表わしている．たとえば，先の型宣言で，walks という単語は 動詞と判断

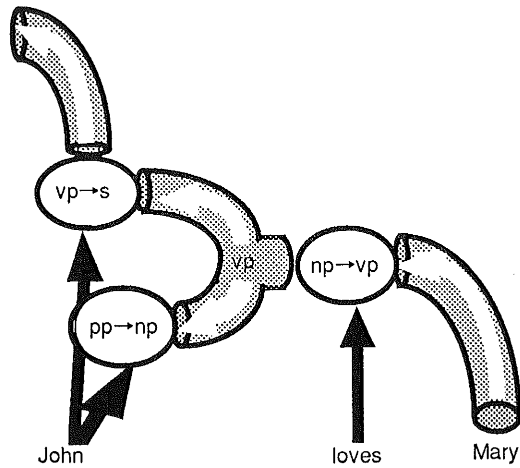


図 3.7.1 Laputaの動作原理

できるので，walks：(名詞→文) としても判断できることになる．これは，walks というのは，何か名詞という型をもつ単語を引数にとるとその結果は文という型になっているような関数であるということである．こういう関数に引数をわたした結果を，単語を左に接合することにすると，walks という関数の引数に John という値を適用した結果は [John , walks] となり，これが文という型として判断されることになる．

ここでは構文規則を例にとって説明したが，文字列をオブジェクトとして単語を型とすればその型判断は形態素解析になり，語や句の意味表示をオブジェクトとして，その意味的カテゴリーを型とすれば型判断は意味解析になる．

3.7.5 並列アルゴリズム

自然言語解析の並列処理の研究で，最もよくわかっているのが並列構文解析のアルゴリズムである．筆者らの型推論システムは，京都大学の松本裕治氏が開発した並列構文解析システム PAX を基本にしたアルゴリズムを利用している [64]．

A. 並列構文解析システム PAX と Laputa

まず，PAX のアルゴリズムを説明する．構文解析は，構文木と呼ばれる木構造を作っていく過程と見ることができる．PAX では，構文木を単語という葉っぱから始めて，文という根に向かって構成していく．このような解析法をボトムアップ解析と呼ぶ．

木を構成していく過程で小さな部分木がたくさん構成される．PAX では，そのような部分木をプロセスと呼ばれる並列処理の処理単位にしている．プロセスとプロセス，つまり部分木と部分木はストリームと呼ばれる通信路を使って通信を行なっている．ストリームのなかを流れるデータは，部分木と部分木を結合してさらに大きな木構造を構成するための構文規則である．

筆者らが開発した並列自然言語解析システム Laputa の型推論アルゴリズムは，PAX のアルゴリズムのプロセスと構文規則の関係をちょうど反転させたものである．つまり，部分木を通信データにし，構文規則をプロセスにしている．筆者らは型推論のための効率的な並列処理方法として Laputa のアルゴリズムを考案したが，筆者らと独立に同じアルゴリズムが ICOT の瀧和男氏 (現・神戸大学) によっても構文解析アルゴリズムとして考案されていた [65]．

Laputa の並列解析機構がもつ並列性には，規則の適用に曖昧性ができたときに複数個のプロセスを起動することによるＯＲ並列性と，構成に成功した部分木を次々に別のプロセスつまり構文規則に送信してより大きな解を構成しようとするパイプライン並列性がある．パイプライン並列関係にあるプロセスは，順次的にデータを伝達する通信路であるストリームによって貫かれていて，ストリームを通じて解を受け取る．

B. レイヤードストリーム法

レイヤードストリーム法は，PAX と同じく元 ICOT の松本裕治氏 (現在、奈良先端科学技術大学院大学) によって考案された探索問題のための並列プログラミング手法である．PAX や Laputa はレイヤードストリーム法を利用している．

並列論理型言語 KL1 では，複数の述語の間で共有される変数が通信路の役割をもち，そこにリスト構造で要素を順に代入していくとストリームになる．レイヤードストリームは，ストリームを流れる一つひとつの要素が部分解とストリームの対であり，この対になっているストリームに流れる要素も，再び部分解とストリームの対であるような再帰的構造になっている．レイヤードストリームという名前は，ストリームが入れ子になって階層化されていることを意味している．

レイヤードストリーム法は，このようなストリームの構造を使って探索空間を表現する手法である．部分解と対になっているストリームは，その部分解と and 関係で成立する解の集まりを意味しているので，レイヤードストリームは探索空間を部分解ごとに木構造にコンパクトにまとめた構造になっている．このため，いったん計算した部分解は，その部分解と対になっているストリームのなかの計算において共有されるので，同じ部分解を再度計算する必要がないという利点がある．これは構文解析においては，いったん作った部分木は，それよりも上の木構造を構成するときに複数の構造に分化したとしても，部分構造として共有されるということに対応している．ゆえに，この方法は効率的な構文解析アルゴリズムとしてよく知られているチャート法と基本的には同じ処理である．

3.7.6 負荷分散手法

負荷分散については結局第五世代プロジェクトの間に実現することはできなかったが，実現方法の検討を行なっていたので，これについて解説する．

並列処理の処理モデルを考えるときに，自然現象や社会構造などがよい参考となることが多い．筆者らの並列優先度制御は，自然界における生存競争を参考にして考案したものである．

自然界における生存競争の究極的な機能は，劣性な個体を淘汰して優性なものを選別することである．計算機による優先度制御の究極的機能はよい解を選別するという機能であり，自然淘汰の機能と酷似している．筆者らの「競争」という概念は，よい解が自動的に有利に計算される仕組みを作り出すためのものである．

適用可能な規則が複数個ある場合，Laputa の並列処理系はそれらの規則をＯＲ並列的プロセスとして扱うが，競争を導入するとこういったプロセス同士がプロセッサや通信権などの資源をめぐって競争を行ない，優先度の高いプロセスに優先的に資源が与えられるために，よい解がより有利に計算されるという仕組みである．

競争は，このようによい解を優先的に計算することを究極的な目的とするものであるが，競争の形態そのものも重要な機能をもっている．自然界の生存競争では，テリトリーの争奪という形態の競争がある．これは一定のテリトリーを確保できた個体のみが子孫を残すことができるというものだが，たとえば森林の面積などのような資源の量が変動しても人口密度を一定に保つという機能をもっている．並列処理では，プロセッサの負荷や通信量を一定のレベルに保つことが並列ソフトウェアの処理効率をチューニングするうえで最も重要なポイントであるが，負荷レベルの調節機能はテリトリー争奪競争の人口密度の調節機能と類似している．筆者らは，この類似性に着目して負荷分散方式の着想を得た．

競争に基づく優先度制御のもつ特長として強調すべき点がもう一つある．それはこういった制御が自律的に行なわれるという点である．並列処理の制御は逐次処理の制御に比べて人間が動作状況を把握するのは非常に困難である．したがって，並列処理の制御はできるだけ人間が意識することなしに自律的に行なわれるのが望ましい．この自律的な優先度制御という方法を経済的構造に類比させると，市場経済に類比させることができる．また他方で，人間が完全に制御を与える方式は計画経済に類比させることができる．原理的には人間が完全な制御を与える方式のほうが効率的なように思えるが，システムが複雑になるにつれて制御の複雑さは人間の手に余るようになるので，結果として自律的に制御が決まる方式のほうが効率的になりうるということは昨今の世界情勢からも推察できることである．

優先度の利用目的には，少なくとも二つの観点がある．一つは処理効率を高める手段であり，もう一つは複数個得られた解から最良の解を選択する基準である．前者の優先度は規則に与えられるべき情報であって，規則の適用制御に利用される．後者は解もしくは部分解に与えられるべき情報であって，解の選択に利用される．両者を区別するために，解につけられた優先度を評価値と呼ぶことにする．競争に基づく優先度制御は，この両方の機能の実現を目的としている．

優先度制御はプログラムの意味には影響を与えないので，優先度制御を追加しても全解探索を行なうことに関しては変わりない．プロセッサの負荷や単位時間当たりの通信量の最適化を行なう機能によってシステムの処理全体の効率化を図るという機能もあるが，結局，優先度制御の追加による効果は，よりよい解がより速く得られるということにつきる．ただし，これは最初に得られた解が最良の解であるということを保証するものではない．このため，得られた解にはすべて評価値が付与されている．

3.7.7 開発の経過

筆者らの並列自然言語解析システム Laputa の開発は ICOT 後期から始まったが，実質的にはさらにさかのぼって，ICOT 中期の成果を示した FGCS'88 のために，並列構文解析システム PAX を Multi-PSI の上にのせる作業を手伝うことから始まった．

A. FGCS'88 での PAX の開発

Prolog プログラマが KL1 でプログラミングを始めると，最初，一種のパニック状態に陥ることがある．筆者がそうであった．KL1 と Prolog は構文が一見似ているが，しかしまったく違った言語であるということを理解できると，次第にのめり込むようになっていった．それはいいとして，PAX はなんとか Multi-PSI で稼働するようになったが，性能的には問題があった．64 台のプロセッサを使用しても，1 台での処理の 3 倍程度しか速くならないのである．Multi-PSI は，プロセッサ間の通信コストが大きい．PAX は解析中に非常にたくさんの通信を行なうので，どうしても通信による処理の遅延が発生してしまうのである．

負荷分散の問題も深刻であった．PAX の処理単位は非常に計算量が小さなプロセスであり，こういうプロセス同士が複雑にストリームでつながって処理を進めている．このため，プロセスをどのようにプロセッサに割り当ててみても，プロセッサ内部の処理量とプロセッサの間の通信量とのバランスを適切な関係にできないのである．

B. プロセスと通信されるデータの関係を反転させる

PAX では，プロセスは実際に構成された部分木に対応しており，プロセス間で通信されるデータは，適用可能な構文規則の候補と部分的に完成した構文規則に対応していた．適用可能な規則がたくさんのチェックの末に部分木としてまとまるので，明らかに通信されるデータのほうがプロセスの数よりも多い．Multi-PSI では，処理の粒度を大きくして通信量を削減するほうが処理の上で有利なので，プロセスと通信されるデータの関係を反対にすることを考えた．この方式は筆者らとは別に，ICOT の瀧和男氏によっても着想されていた．

この PAX を反転させる方法にも，いくつかのバリエーションがあることがあとでわかった [65]．筆者らの方法は，プロセスすなわち構文規則が OR 並列的に動作するようにしたが，三菱電機の佐藤裕幸氏は，論理型言語の高速化技術であるクローズインデキシングが有効に働くようにする方法を開発した．

C. 性能解析ツール ParaGraph で処理のボトルネックを見つける

本稿では，並列処理はモデル化に有効であるということを強く主張してきたが，処理速度が速くなるというのもやはり重要である．というよりも，速くならなければ誰も相手にしてくれない．特に企業からの期待は，性能に関することに集中しているといっても過言ではない．

並列処理を行なうと，たとえば 32 台のプロセッサを使えば 32 倍速くなるべきだと思われるであろう．しかし，実際には 32 台使っても，なかなか 1 台のときより速くならないのが現実なのである．筆者らは，許容できるぎりぎりの目標として，32 台使うと 10 倍速くなるということを設定した．

この目標を目指して，プログラムを数度にわたって全面的に書き換えたりしたが，なかなか性能は上がらなかった．

そこに性能向上のための強い味方として現われたのが，PIMOS の性能解析ツール ParaGraph である．ParaGraph は，プロセッサごとの処理量や処理待ちの発生頻度，時間とともにプロセッサの負荷がどのように推移したかというような情報を収集してくれる．

並列処理では，どこか 1 か所でもボトルネックになる処理があると，ほかのいろいろな場所で処理待ちが発生してしまい，全体の性能低下につながってしまう．性能改善の作業は，このようなボトルネックになっている部分を発見しては一つひとつその処理を改良していくことであった．あらゆる箇所のアルゴリズムを見直して，アンバランスに遅い部分をなくさなければならなかった．

もう一つのポイントは，データをどこかのプロセッサに集中させないことである．たとえば，サーバ - クライアントモデル的な処理は，サーバがいるプロセッサにどうしてもデータが集中してしまい，そのデータにアクセスするために，計算負荷も通信もそのプロセッサに集中してしまう．したがって，データをうまくたくさんのプロセッサに分散するように，プログラムの改良を行なう必要があった．

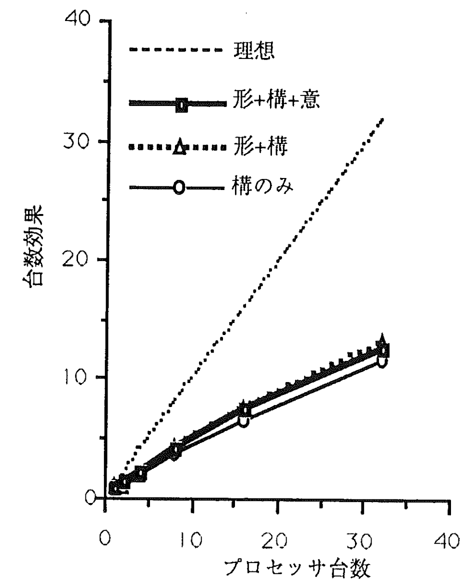


図 3.7.2 プロセッサ台数と台数効果

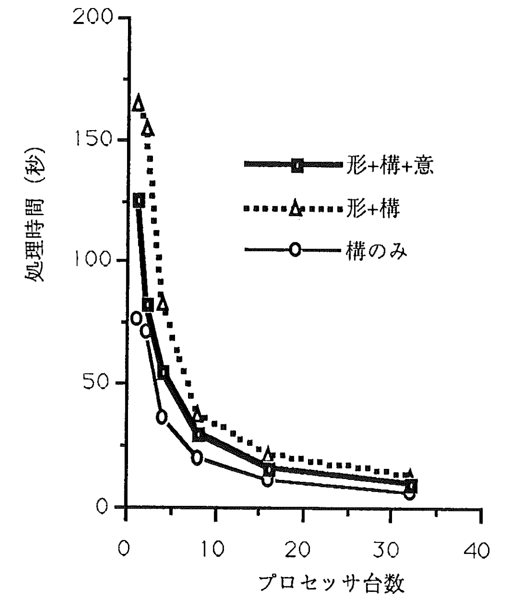


図 3.7.3 プロセッサ台数と解析時間

D. 並列処理効率向上のための文法の改良

並列処理効率を向上させるために次に問題になったのは，文法であった．筆者らが利用したのは ICOT の佐野洋氏が開発した日本語の構文文法である．この文法は，非常にスマートに設計された拡張性の高いもので，今後の発展が期待できるものであるが，あくまで日本語の構文文法の網羅的記述に重点をおいた研究の成果であり，並列処理効率についてはほとんど考慮されていない．

文法規則としては同じ意味でも，規則の書き方によって通信の量や並列度が大きく変わってくる．最初は，佐野氏が作成した文法を機械的に変換して，並列処理文法を作成して実験を行なったが，どうしても並列処理性能が出ないので，やむなく人間が文法規則全体の意味をよく読み取って，それを並列処理に有利な形に変えた文法を再設計した．できるだけ佐野氏の考察や思想を忠実に反映した文法になるように努力したが，完全に同じものにできなかったのが残念である．

E. 大規模化

筆者らの研究は，並列自然言語処理技術を実用技術にすることを目指している．自然言語処理の問題点は大規模化のときに現われてくるので，小さなシステムでは実用技術の評価はできない．実用技術としての評価ができるぎりぎりの規模として，単語辞書約 1 万語，意味概念約 700 概念，文法規則約 600 規則の規模のシステムを作成した．大規模化による並列処理効率の低下を恐れたが，実際にはそれほどひどいものにはならなかった．

F. 性能評価結果

評価用例文 18 文について，構文解析だけ，形態素解析と構文解析の協調，形態素解析と構文解析と意味解析の協調の三つのケースについてプロセッサ台数，リダクション数，解析時間の関係を調べた．その結果，32 台のプロセッサを使用すると，最高で約 13 倍速くなることがわかった (図 3.7.2)．

a. プロセッサ台数と解析時間

図 3.7.3に示すグラフは，解析に使用するプロセッサ台数を 1 台から 32 台に変えていったときの解析時間の変化と台数効果の推移である．

筆者らの並列自然言語解析システムの形態素解析は，文字列から認定可能なあらゆる単語をすべて探し出している．つまり，最長一致法や字種の違いの利用や接続関係の検査などの解の数を削る処理は一切行なっていない．認定された単語の並びの適切さは構文解析や意味解析による制約だけで判断している．

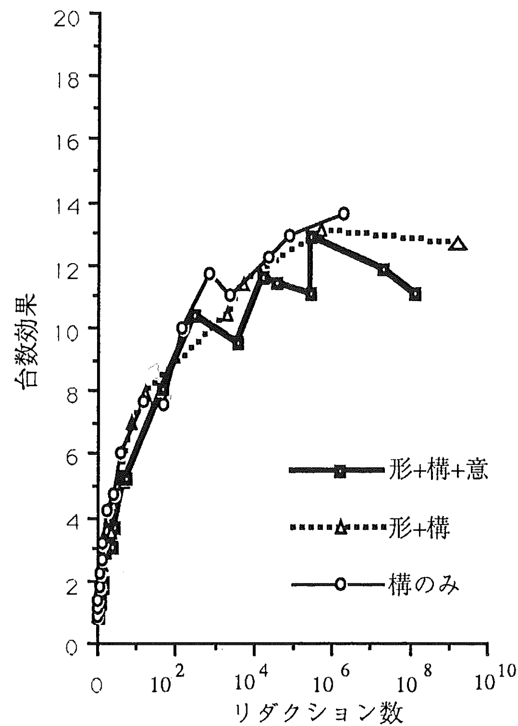


図 3.7.4 計算量と台数効果

このような処理方法は，逐次処理ではあまりにも無謀であった．実際，この実験結果でも 1 台のプロセッサで実行したときの処理時間は形態素解析を行なうと非常に遅くなっており，実用的とはいえない．しかし，並列処理を行なうと実質的に構文解析だけの処理時間とさほど違わない時間で処理が終わっているので，実用的価値がある処理方法になっている．

意味解析の並列協調は，構文解析の解の数を減らす効果があるだけでなく，全体の計算量を減らす効果もあることがこの実験によって確認できた．つまり，構文解析だけの処理の計算量よりも，構文解析と意味解析を並列協調させたときの計算量のほうが少ないことが確かめられた．

b. 計算量と台数効果

計算量に対する台数効果の推移はどのケースでもほぼ同じ振舞いを示した (図 3.7.4)．これは Laputa の処理系がいずれの処理においても同一の処理メカニズムで動作していることからの帰結として理解できる．この実験も 32 台構成の Multi-PSI を使用して実施している．したがって，台数効果は 1 台の処理速度と 32 台での処理速度の比率である．

いずれのケースでも計算量の増加とともに台数効果も向上しているが，13 倍のあたりで台数効果が飽和している．

3.7.8 むすび

筆者らが行なった研究は，結局，従来からある形態素解析，構文解析，意味解析を並列協調させただけであり，従来できなかったことができるように変わったわけではない．

しかし，自然言語処理にはもっと多くの協調させるべき要素が存在している．文脈や状況からくる情報，現実世界からのフィードバック情報や談話や知識を利用した類推系との協調など，ほかにも多くの情報チャンネルとその協調処理がある．こういう処理の協調によって，自然言語処理技術は本当に新しい段階に踏み出すことになるであろう．また，並列推論マシンの応用技術の観点からも，多数のプロセッサをさらに有効に利用するためには多くの協調要素をもつほど有利に働くであろう．

自然言語の意味は記述可能なものばかりではない．計算機を人間と人間が知識や情報を伝え合うためのメディアと見たときに，記号処理機械としての計算機の能力を有効に利用するためには，計算機内部での表現と人間が実際に目や耳にする画像や音声などを扱う処理の協調も並列処理の要素に入ってくるべきであろう．逆にいえば，現在の自然言語解析システムは，ヘレン・ケラー以上に閉ざされた境遇にあるともいえるだろう．こういう多様な処理を並列協調させてはじめて並列協調モデルの意味がでてくるものと筆者は信じている．

並列処理はまだ始まったばかりの技術であり，ほとんどが未踏の領域である．多くの研究者がこの未踏の領域の開拓に参加されることを期待する．

3.8 データベース

3.8.1 はじめに

第五世代コンピュータは既述のように，並列と推論の二つのキーワードでとらえられている．このなかで「並列データベース管理システム」はどのようなシステムで，どのような位置を占めるのだろうか，これが

本稿のテーマである．これに入る前に，この節ではまずデータベース自体について少し考えてみよう．

データベース (database) という言葉は，データとベースとの造語であり，「データの基地」を意図している．以前はデータ・ベース (data base) と書かれることが多かったが，最近ではデータベースと一語で書くのがふつうとなっている．データベースの略語はこの歴史を反映して DB である．

そのようなデータベースを管理するシステムがデータベース管理システム (database management system; DBMS) であるが，これを単にデータベースと呼ぶことも多い．したがって，文献情報や生物情報のデータベースという場合，それはデータ自体のことなのか，そのような情報をもったシステムのことなのか，文脈なしではわからないことが多い．

DBMS の教科書を見ると，それは以下のように 3 種類に大きく分類されていることがある．

1) 情報検索システム
2) (狭義の) データベース管理システム
3) トランザクションシステム

1) は，データベースから有効な情報をいかに早く取り出すかを中心に考えたシステムで，データベースの更新と検索は異なったタイミングで行なうのがふつうである．データベースの内容としては，当初は文献情報の二次情報が多かったが，最近ではさらに，文献の一次情報や化合物の構造情報などの複雑なデータも対象としている．2) が一般には最もよく知られた DBMS であり，教科書でもこれが中心となっている．この特徴は比較的単純なデータを対象にした処理で，事務処理がよく例として取り上げられている．3) は，銀行の預金システムのように，単純なデータを対象に，単純なトランザクションをいかに大量に効率よく処理するかを目指したシステムで，専用に近いシステムが多い．

しかし，データベースの応用分野が拡大するにつれて，このような分類は必ずしも現状に合わなくなってきている．たとえば設計支援システム (CAD) を考えれば，対象となるデータは構造をもった複雑なもので，設計文書の管理には情報検索技術も要求される．遺伝子情報処理のための分子生物学データベースの場合にも，配列の類似検索から構造データの取扱い，化学反応の推移閉包を求めるものまである．

IDS という初の DBMS が 1963 年に世の中に登場してから 30 年になるが，最近では上記の分類だけでなく，伝統的なさまざまなデータベースの枠組みが再考されるようになり始めている．データモデルという観点から考えると，ネットワーク型，階層型，関係型という三つの古典的なデータモデルのほかに，オブジェクト指向データベースや演繹データベースなどの数多くのデータモデルが提案されてきている．応用分野の拡大を考えれば，これらのなかの一つのデータモデルですべての応用をカバーするのではなく，複数のデータモデルを使い分ける必要があるとのコンセンサスが生まれてきている．

3.8.2 第五世代コンピュータとデータベース

第五世代コンピュータプロジェクトでは，このような背景のもとで，知識情報処理のための知識ベース管理システムを研究開発の一つとしてきた．そして現在もそれを継続している．知識情報処理として考えているのは，遺伝子情報処理，法的推論，自然言語処理，定理証明などのシステムである．遺伝子情報処理には，遺伝子やタンパク質の配列情報や構造情報，あるいは酵素反応などの機能情報といった大量のデータ/知識が必要とされる．法的推論のためには，法令だけでなく，過去の判例や学説などの大量のデータ/知識が必要とされる．ほかの応用についても同様である．

このような多種多様なデータと知識をいかにとらえるべきだろうか．たとえば，ある数式で記述された情報があるとしよう．一定水準の数学の素養のある人にとってはそれは知識の宝庫であるかもしれないが，そうでない人にとっては単なる文字 (記号) の列にしか見えないかもしれないし，さらに言葉を解さない人にとってはそれは意味不明な落書きでしかないかもしれない．知識情報処理の扱うデータと知識についても同じことがいえる．つまり同一の対象であっても，その理解能力によってそれはデータにも知識にもなりうるのである．扱われるさまざまな知識は，ディスクなどの永続記憶上では文字列あるいはビット列でしかない．データベースと知識ベースは永続記憶と応用の間に存在して，両者の橋渡しを行なう役割をもっている．これらシステムは，文字列をデータとして解釈し，さらにはデータを知識として解釈するのである．内包的に

定義されたデータに対しては，必要な推論を行なうことによってデータ/知識を取り出さなければならないのである．

データベースと知識ベースという言葉は立場によって用法が異なっている．たとえば，関係データベースを論理の枠組みで拡張し，ルールをもデータの一部として扱い，推論能力をもたせたデータモデルとして演繹データベースがある．これはデータベースの側からの用語である．この流れに沿って考えれば，さらに強力な知識表現言語を備え，推論能力を拡張していっても「○○データベース」という言葉が使われそうである．一方知識ベースという言葉は，エキスパートシステムなどの人工知能のなかで育ったが，内容的にはかなり曖昧なところがある．知識の量が増えれば，当然それらを効率的に永続記憶に格納し管理するためのデータベースエンジンが必要となるので，データベースと知識ベースの両者の統合が必要となるが，まだ明確な枠組みがないのが現状といってよいだろう．

第五世代コンピュータでの知識ベース管理システムは，データベース管理システムの拡張として考えたほうがよいとの立場に立っている [66]．そのための枠組みとして提案したのが，関係データベースを論理パラダイムで拡張した演繹データベースを，さらにオブジェクト指向パラダイムで拡張した「演繹オブジェクト指向データベース」[67] である．つまり用語として違和感を感じる方がいるかもしれないが，知識ベース管理システムとしての演繹オブジェクト指向データベース管理システムなのである．現在研究開発しているシステムは，上位層の知識表現言語あるいは知識ベース言語に相当する *Quixote* [68] と下位層のデータベースエンジンである Kappa-P[69] とから構成されている．この Kappa-P が本稿の主題である並列データベース管理システムである．*Quixote* については参考文献 [66] を参照していただきたい．

3.8.3 正規関係と非正規関係

Kappa プロジェクトは，自然言語処理システムのための辞書やシソーラス，証明検証システムのための数学知識を，逐次型推論マシン上でいかに効率よく管理するかという動機で，1985 年に始まった．たとえば，図 3.8.1の例を考えてほしい．これは，遺伝子の配列情報を集めた GenBank というデータベースの構造を示している．図中の属性名の後ろの "*" は繰返しであることを意味している．つまり属性が階層構造をしており，そのノードのいくつかで繰返しが許される構造となっている．このような関係を非正規関係と呼ぶ．一方，関係データベースが採用している平坦な構造を正規関係と呼んでいる．

利用者間でデータを共有するためには，それが同じプラットフォームに乗っている必要がある．現在のデータベース環境を考えれば，それは関係データベース管理システムしかない．これはプログラミング言語における COBOL や FORTRAN と同じような意味である．そこで GenBank の配布元では，図 3.8.1を いかに正規関係に展開するかを利用者マニュアルに記述している．それによれば，上の情報を 25 の実体に分解し，五つのグループに分け，その結果できる正規関係の数は何と 55 個である．実体とかグループというのは関係データベースの概念ではなく，ユーザが責任をもつべき概念なので，ここまで分解されると関係間のつながりを直観的につかむのが難しく，簡単な問合せをするのにも一工夫が必要となる．しかも関係を分割すると，結果を求めるために関係と関係を結合するというコストの高い演算が必要となり，処理効率も悪くなる．たとえば，某メーカの関係データベースのマニュアルには，正規化をしすぎないようにとの注意書きまである．

なぜこんな困難が生じたのかが重要なポイントである．現実のデータ構造と理論上の都合との乖離があまりに大きいのである．正規化は，データの削除時に関係のないデータまでも削除しないために必要な概念として提案されたものである．しかし，これが現実の応用を考えると大きな問題を生じるのは上の例から明らかである．そこで関係データベースの拡張が数多く提案されてきた (図 3.8.2)．

前項で述べたように，第五世代コンピュータの知識ベース管理システムとしては演繹オブジェクト指向データベースを採用しているが，その永続データを効率的に管理するためのデータベースエンジンとして非正規関係モデル [70, 71] を採用した．これは図 3.8.1 のデータに有効なことがすぐにわかる．正規関係に比べると関係の数が減少し，関係間の結合演算も減少するので，記憶域の効率的利用と処理効率の向上が期待できる．

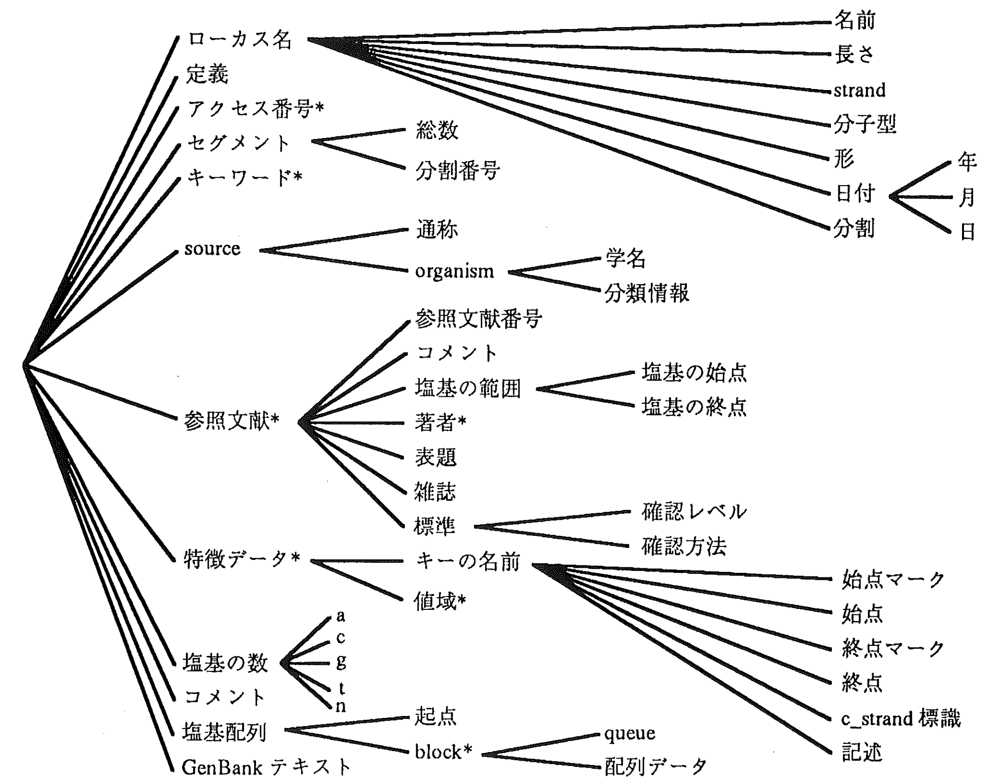


図 3.8.1 GenBank の構造

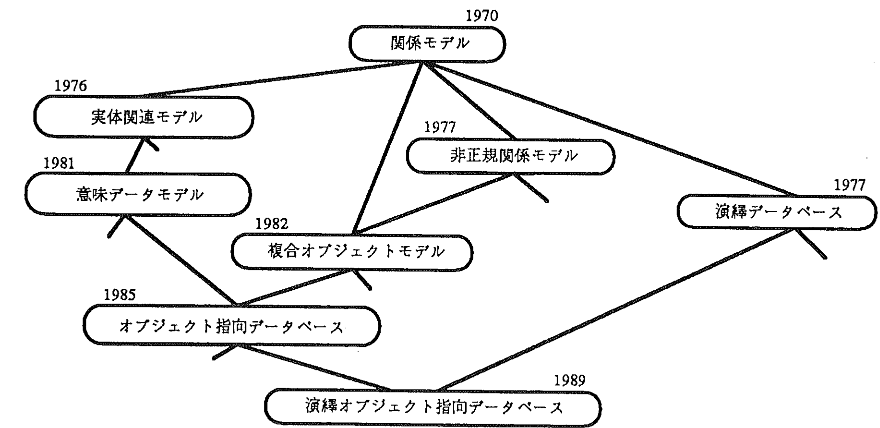


図 3.8.2 関係データベースの拡張

ただし，次のようないくつかの検討すべき問題点があった．

- 集合をもっているために，それにどんな意味論を与えるべきか．
- 関係モデルとどのような関係にすべきか．
- 構造体を効率的に処理するために，内部データ構造とアルゴリズムをいかにすべきか．
- ユーザデータを非正規関係に変換するための設計方針はあるか．

1985 年当時，非正規関係モデルに基づいたプロトタイプ・システムが三つあった．ドイツのダルムシュタット大学の DASDBS[72] とドイツ IBM の AIM-P[73],

そしてフランス INRIA の Verso[74] である．前の二つは同一人物 (H.-J. Schek) によって始められたもので，基本的には同じモデルと考えてよい．これらの特徴として，DASDBS と AIM-P は意味論は自然であるが処理は重い，Verso は処理は効率的であるが意味論は不自然である，と考えられた．そこで Kappa では，表現と処理のバランスを考え，第三の道を取った．つまり

1) 集合は構文上の略記と考える．
2) 属性の階層関係を除き，関係データベースと意味論上は同じ．
3) 独自の内部データ構造とアルゴリズムを考える．
4) 非正規関係の「正規形」設計論に従う．

という方針にした．開発の歴史は図 3.8.3のようになっている．ESP で開発された Kappa-I と Kappa-II [75] は逐次推論マシンで，KL1 で開発された Kappa-P は並列推論マシンで動作する．Kappa-II のなかで研究開発していた知識ベース管理システムは現在 QUIXOTE となっている．

3.8.4 非正規関係データベース管理システム

さて，いよいよ Kappa システムについて話を進めよう．並列 DBMS Kappa-P は，Kappa-II の並列版として設計され，多くの特徴を継承している．まず，両者に共通する事柄について述べよう．

A. データ型

Kappa は，知識情報処理環境の DBMS を目指しているので，新しいデータ型として項が加えられた．それは，多様なデータと知識はしばしば項の形式で表現されるからである．これらの操作のために，Kappa-II では単一化とマッチングがつけ加えられた．さらにまた，データ型によって，2 バイト文字 (JIS) と 1 バイト文字 (ASCII) を識別する．これは，遺伝子配列データのような膨大なデータを圧縮する助けとなる．

B. コマンドインタフェース

Kappa では，2 種類のコマンドインタフェースを用意している．下位レベルインタフェースの基本コマンドと，上位レベルインタフェースの拡張関係代数である．多くの応用では，(コストのかかる) 拡張関係代数のレベルは必ずしも必要ではない．このような応用では，ユーザは基本コマンドを使用することによって処理コストを減らすことができる．

DBMS とユーザプログラムの間の通信コストを減らすために，Kappa ではユーザ定義コマンドを用意している．それは，Kappa-II では Kappa のカーネルと同じプロセスで，Kappa-P ではそれぞれのローカル DBMS と同じノードで実行される．

ユーザ定義コマンド機能によって，ユーザはそれぞれの応用システムに適したコマンドインタフェースを設計したり，そのプログラムを効率的に走らせたりすることができる．

C. 実用性

すでに述べたように，Kappa は，QUIXOTE のデータベースエンジンとしてだけではなく，実用的な DBMS として QUIXOTE とは独立に稼働することも目指している．この目的を達成するために，いくつかの拡張機能がある．まず第一に，上記で言及したデータ型のほかに，その応用の環境を格納するために導入された，新しいデータ型がある．それはリスト，バッグ，そしてプールである．しかしながら，それらは意味論上の困難さによって，拡張関係代数においては十分にはサポートされていない．

Kappa は，このようなデータ型のために SIMPOS や PIMOS がもっているのと同一のインタフェースをサポートしている．

ウィンドウシステムから Kappa データベースを使うために，Kappa はスプレッドシート風のユーザ指向のインタフェースを用意している．それには，更新も含んだアドホックな問合せ機能，さまざまな出力フォーマットでのブラウジング機能，そしてカスタマイズ機能などがある．

D. 主記憶データベース

頻繁にアクセスするデータは，主記憶データベースとして主記憶にロードし保持しておくことができる．この主記憶データベース機能によって二次記憶データベースに余分の負荷を負わせたくないので，現在の実装では，一時関係の効率的な処理のためにのみ設計されている．したがって，遅延更新や同期などの機能はサポートしていない．Kappa-P では，主記憶データベースは二次記憶データベースよりも，少なくとも 3 倍は効率的である．二次記憶データベースと同期を取った主記憶データベースは，Kappa-P では複製データベースの一種として実装されている．

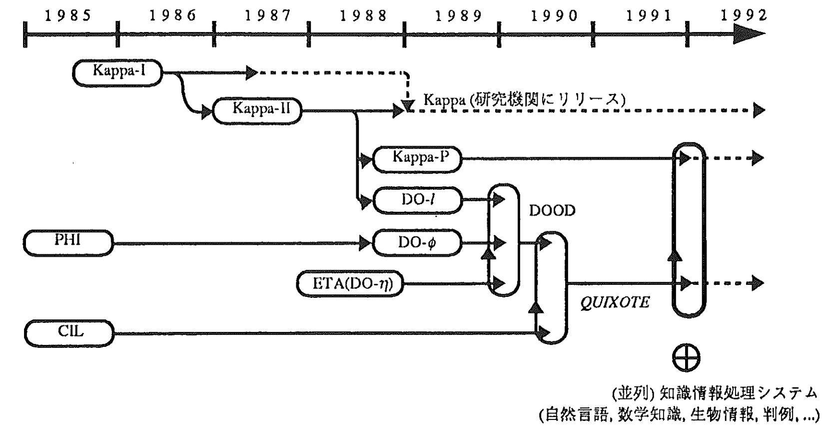


図 3.8.3 データベースと知識ベースのプロジェクトの簡単な歴史

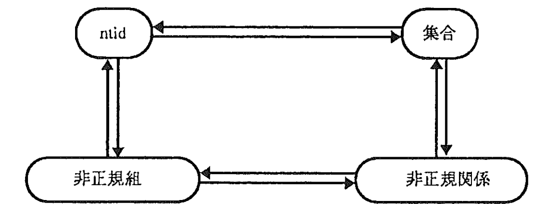


図 3.8.4 Kappa における“オブジェクト”とその基本操作

実装の面では，Kappa の効率的な処理にはいくつかの指摘すべきポイントがある．これらのうち，2 点について述べる．

E. ID 構造と集合演算

ネストした (非正規) 組は，関係のなかで一意に決められる組識別子 (ntid) をもつ．それは，明示的に処理されるべき一つの“オブジェクト”として扱われる．抽象的にいうと，非正規組，ntid，ntid の集合，そして非正規組の集合 (関係) の四つの種類の“オブジェクト”がある．基本的には図 3.8.4のように示される，変換のためのコマンドがサポートされている．ただし，Kappa-P では集合はストリームとして扱われている．大部分の演算は，ntid か集合の形式で処理される．

選択結果の処理のために，非正規組のそれぞれの部分組はまた，仮想的に sub-ntid をもつ．(アンネストとネスト演算を含む) 集合演算は，対応する組を読むことなく，主に (sub-)ntid もしくは集合の形式で処理される．

F. 格納構造

意味的にはアンネストされた組で構成される非正規組はまた，同時にアクセスされるアンネストされた組の集合としてもみなされる．それゆえ，非正規組は分解されずに，圧縮され，原則的には二次記憶の同じページに格納される．遺伝子配列のような膨大な組には，連続したページが使われる．効率的に組にアクセスするためには，二つのことを考えなければならない．まず，必要な組の位置を効率的に決定することと，組のなかから必要な属性を能率よく取り出すことである．組の位置決定のためには，図 3.8.5のように，ntid と論理ページ (lp) のアドレス変換テーブル，そして論理ページと物理ページ (pp) のアドレス変換テーブルを備えている．後者のテーブルは，下層のファイルシステムによって使用される．組から属性を取り出すために，非正規組のそれぞれのノードは，圧縮された組のなかに局所ポインタとカウンタをもっている．ただしこれは更新負荷を増大させるので，更新処理の効率とのバランスが重要となる．

インデクスのそれぞれのエントリはネスト構造を反映し，sub-ntid を含んでいることもある．そのエントリの値は，部分文字列や結合などの文字列演算の結果であってもよく，またユーザのプログラムによって抽出された結果であってもよい．

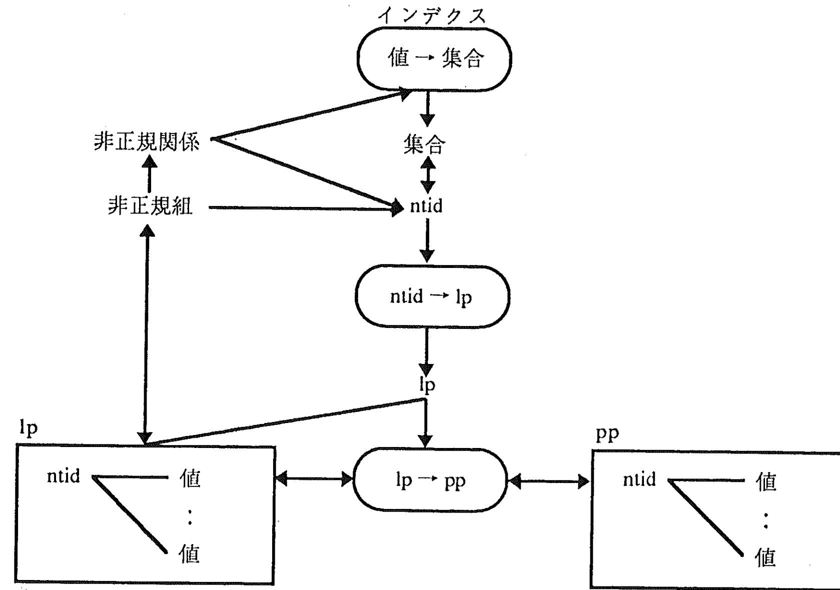


図 **3.8.5** 二次記憶 DBMS のアクセスネットワーク

3.8.5 並列データベース管理システム

次に，Kappa-P について並列化に絞って話を進めよう．

Kappa-P が動作する環境は，並列推論マシン PIM とそのオペレーティング システム PIMOS の環境である．PIM は MIMD 型の疎結合と密結合が混合されたハイブリッド並列マシンある．10 台程度の要素プロセッサ (PE) が，共有バス/共有メモリにより結合され一つのクラスタとなり，それがネットワークで結ばれている．共有メモリは，数百メガバイトで，ディスクは各クラスタに取りつけることができる．この環境では，ハードウェアとしては，クラスタを構成する PE，ネットワークでつながれたクラスタ，クラスタに取りつけられたディスクなどが並列に動作する．ソフトウェアとしては，並列 DBMS 自身とそのアプリケーションプログラムがそれぞれ並列を意識して書かれており，両方とも同じ並列マシン上で動作する．このような背景のもと，並列化にあたっては次の点を考慮した．

- 大量データに対する考慮
- ハードウェア資源の有効利用
  疎結合並列処理，密結合並列処理の両方を考慮
  大容量主記憶を生かした主記憶データベース機能
  ディスクに対する並列アクセス
- アプリケーション プログラムとの関係
  アプリケーション プログラムとの通信量の削減
  一つのアプリケーション プログラムから並列に要求が出せること

A. 全体構成

DBMS は大量のデータを扱うので，疎結合並列処理における通信量が大きな意味をもってくる．そのため，密接に関連するデータを同じクラスタに置くなどのデータの配置が重要で，さらに問合せ処理において通信量が少なくなるようなプランを立てる必要もある．これらを行なうためには，分散データベースの構成が向いている．

図 3.8.6に，Kappa-P の全体構成を示す．各クラスタにローカル DBMS(LDBMS) と呼ばれる DBMS を配置し，全体で一つのデータベースを管理する．この LDBMS は，それ自身で DBMS としての全機能をもち，複数 LDBMS が関与する問合せを処理するために，二相コミットプロトコルに基づく分散トランザクションの機能をもっている．また，複数の LDBMS により一つのデータベースを管理するため，関係名などの大域情報の管理が問題になるが，それを管理するサーバ DBMS (SDBMS) の複製を作ることにより，

アクセスの集中を回避する．クラスタに割り当てられた LDBMS は，その内部処理で密結合向きの並列処理を行なう．

問合せ処理は，インタフェース プロセスと呼ばれるプロセスを介して行なわれる．アプリケーションが並列言語で記述されるため，一つのアプリケーションが複数のインタフェース プロセスを利用できるようにしている．また，このインタフェース プロセスが，問合せ処理時に必要となる各種の変換を行ない，該当 LDBMS に対する部分問合せに変換する．

B. データ配置と並列処理

効率的な並列処理を行なうためには，各クラスタの負荷とクラスタ間の通信量をバランスさせてやる必要がある．DBMS の場合はデータが二次記憶上に保存されしかも大量であることから，データの配置と並列処理方式が密接に関係してくる．

- 分散配置
  関係の分散配置は，複数 PE の計算能力を利用する最も単純な場合である．各クラスタで行なわれる演算，データとも独立であり，MIMD 的な並列処理である．この場合，問合せ処理時の通信量を考慮し，関連の深い関係はなるべく同じクラスタに配置する必要がある．
- 水平分割
  関係の水平分割は，組を単位に一つの関係を複数に分割することである．分割された関係は，それぞれ異なる LDBMS に分散配置されるが，基本的に同じ演算が出され，SIMD 的な並列処理になる．一つの関係が一つのクラスタでは扱い切れないほどの量の場合や，特にデータ検索などの CPU 負荷が大きい演算を重視する場合などに有効である．
- 複製
  関係の複製によって，可用性を高めたりアクセスの集中を回避することができる．現在の実装では，大域情報管理でのみ実現されている．

クラスタに割り当てられる LDBMS 内でデータを主記憶に置くか二次記憶に置くかの選択もデータ配置の一種と考えることができる．

すでに述べたように，主記憶データベースとしては一時関係に絞って実装されているが，二次記憶データ

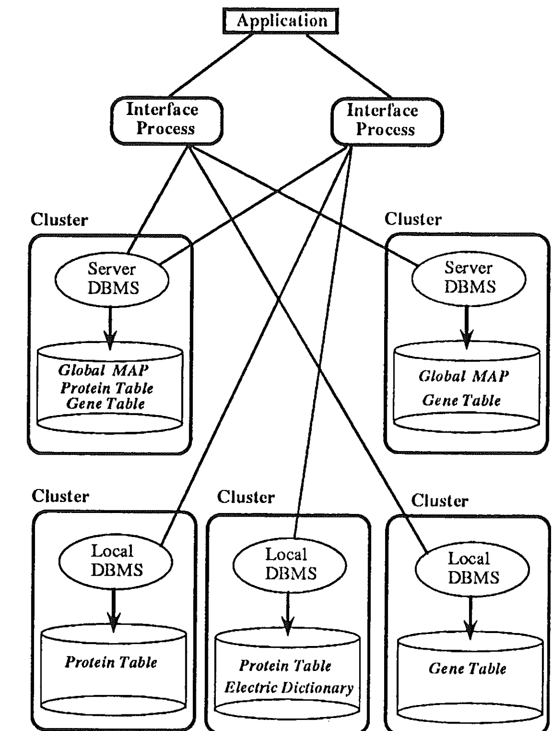


図 **3.8.6** Kappa-P の全体構成

ベースと同期を取った主記憶データベースを，Kappa-P では並列処理を前提とし，複製データベースの一種として実装している．これは，二次記憶関係の複製として主記憶上の一時関係を作り，読み系操作は一時関係に対し行ない，更新系操作は両方の関係に対して行なう．更新時は，両関係に対するアクセスになるが，これにはクラスタ内並列処理が有効に働く．

次に関係代数の並列処理を考えてみよう．問合せを関係代数として表現すると，代数演算子をノード，その間の依存関係をアークとしたデータフローグラフとみなすことができる．このグラフのうち入力が揃ったノードから順に評価していくことにより並列処理をすることができる．また，演算ノードをプロセス，アークを演算対象である組を流すストリームとすることで，パイプラインによる並列処理を行なうこともできる．Kappa-P はこの処理方式を採用しているが，これは実装言語である KL1 のプログラミング スタイルそのものといえるものである．ただし，ストリームを要求駆動にすることにより，処理や中間結果の片寄りをな

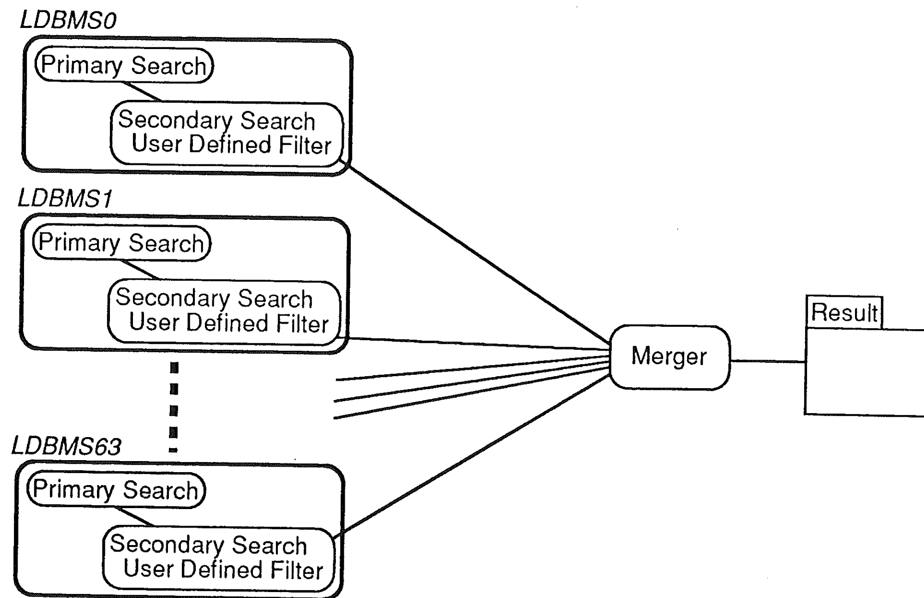


図 3.8.7 フィルタつき読出し機能

くすことを狙っている．

C. アプリケーション プログラムの関係

並列マシン上の DBMS としてアプリケーション プログラムと同じマシン上で動作するため，DBMS とアプリケーション プログラムの関係を次のように考えている．

- アプリケーション プログラムとの通信量の削減
- 一つのアプリケーション プログラムから並列に要求が出せること

アプリケーション プログラムもまた並列に動作するので，一つのアプリケーション プログラムが複数のインタフェース プロセスを作り，DBMS に対し並列に問合せを出すことを許す．問合せの結果はインタフェース プロセスが生成されたクラスタに集められ，アプリケーション プログラムに渡される．

アプリケーション プログラムと DBMS 間の通信量の削減のためには，アプリケーション プログラムを，対象データの存在する LDBMS と同じノードで実行してやればよい．しかし，問題は CPU の負荷にも関連するので，アプリケーションプログラムの一部分の処理で大きく通信量を減らせる処理のみに限るべきである．たとえば，応用に依存した類似検索などを行なう場合，DBMS に対しては一次検索を行ない，その結果に対しアプリケーション プログラム側で 2 次検索を行なうことがあるだろう．水平分割されていない関係に対しては，関係の存在するノードで二次検索を実行してやればよい．対象が水平分割関係の場合，実体は複数の関係で複数ノードに分かれて存在するので，二次検索はその複数のノードで実行すべきである．しかし，水平分割関係はユーザには一つの関係として見せているので，水平分割関係として扱うと，結果は DBMS の内部処理で一つにまとめられてしまう．もちろん，水平分割関係としてではなく，それを構成する関係をその対象とすれば可能ではあるが，それでは構成関係の個数が増えた場合などプログラムを書き直さなければならない．これを解決するために，二次検索処理を組を選別するフィルタとして与えられるようにしたものが，フィルタつき読出し機能である．図 3.8.7のように，二次検索は組を選別するフィルタとして結果が一つにまとめられる前に行なわれる．

3.8.6 むすび

Kappa-P は現在，初版が PIM 上で動作している．その上に分子生物学データベースの一つであるタンパク質データベースを格納し，類似検索の一種であるモチーフ検索などで並列処理の効果を確認している．また，逐次版 Kappa-II 互換のインタフェースを実装し，Kappa-II 上で開発されたアプリケーションプログラムの簡易的な移植を可能にした．今後，計測結果に基づく評価，システムの改良と安定化などを行ない，ICOT 公開ソフトウェアの一つとして公開されている．

今後の課題として以下のようなものがあげられる．上位層の知識ベース言語 $\mathcal{Q}$UIXOTE と統合した一つの知識ベース管理システムにするとき，オブジェクト識別子，入れ子トランザクションなど Kappa-P 側で提供したほうが，全体の効率が上がるものが明らかになった．また，Kappa-P は最小限のユーティリティが実装されているのみで，広く使われるためには，サービス・ユーティリティの充実が不可欠である．さらに広く使われるためには，異種マシンからの利用や異種 DBMS とのリンクなどが必要になるであろう．

第4章

むすび

第III編は，本書のなかでも最も力を入れて編集した部分となった．それは，本書の最大の主題を扱っているということも一つの理由であるが，もう一つはプロジェクトの締めくくりの時期に一番活躍した並列応用プログラムの開発担当者たちの，仕事の総決算とでもいうべき編集企画となったからである．けれどもそれは仕事の終わりを意味するのではなく，これまでの成果のまとめを示したのであって，むしろこれからさらに発展し続けるエネルギーを感じとっていただけたのではないだろうか．

それと合わせて，パワフルなKL1言語と，それが気持ちよく走る並列計算機システムの上で，これまで作られなかったような数々の並列応用と，新しい並列プログラミング技術がふつふつと湧き起こり，新世代の計算機文化が醸成されつつあるという編者の思い入れについても，いく分かのご理解をいただけたのではないかと思う．

KL1プログラムのプログラム方法論とプロセス構造の設計から始まって，負荷バランスや通信オーバヘッドなど，並列プログラミングに欠かせない諸概念，そして動的負荷分散や並列オブジェクトモデル上での分散アルゴリズムなどの具体例を経て，7種類の本格的並列応用プログラムの実現技術と評価結果，そして研究・開発奮戦記を十分堪能していただけただろうか．

お褒めの言葉や励ましをくださる方もあれば，まだこの程度のことしかできないのかとお叱りをくださる方もあることだろう．けれども，第五世代コンピュータの並列プログラミングが，計算機の世界のニューカルチャーを作ろうとしているのだと思うならば，長い道のりの出発点にあたって，好調な滑り出しができたという感慨は，執筆者たちの共通の思いであるに違いない．

本編が，並列プログラミングと並列処理に関心をおもちの方々にとって，何らかのお役に立てば幸いであり，さらに本編を読まれて，自分も並列プログラミングに挑戦しよう，計算機世界のニューカルチャーの開拓者になってやろうと思われる方があるならば，これは編者をはじめとして執筆者陣にとってこの上ない幸せである．

最後に，本編で紹介した並列応用プログラムは，ICOTのオープンソフトウェアとして公開されていることをお知らせしておこう．`icot.or.jp` に anonymous FTPをかけて，ディレクトリ `ifs` の下から取ってくることができる．KL1言語をUNIXマシン環境で試したい方には，プロセッサ1台で擬似並列実行を行なうPDSSという名の処理系が，同じくフリーソフトウェアに登録されている．この処理系は並列推論マシン上の環境とは異なるために，紹介した並列応用プログラムをそのまま実行するわけにはいかないが，KL1プログラミングの入門用には適しており，プログラミングのテキストも同様に公開されている．またICOTでは，KL1がUNIX環境で本当に並列に走る高性能な処理系を現在開発中であり，こちらも近いうちに公開の予定である．どうか利用されたい．

第III編 参考文献

[1] 沖 廣明, 瀧 和男, 清 慎一, 古市 昌一：マルチPSIにおける並列詰め碁プログラムの実現と評価, 並列処理シンポジウム JSPP'89 論文集, pp. 351–357 (1989)

[2] 古市 昌一, 瀧 和男, 市吉 伸行：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, 情報処理学会 計算機アーキテクチャ研究会, 89-79 (1989-11)

[3] Dijkstra, E. W.：A Note on Two Problems in Connexion with Graphs, *Numerische Mathematik 1,* pp. 269–271 (1959)

[4] 和田 久美子, 市吉 伸行：マルチPSI上の最短経路問題の実現と評価, 情報処理学会 計算機アーキテクチャ研究会, 89-79, pp. 83–91 (1989-11)

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

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

[7] 瀧 和男, 市吉 伸行：マルチPSIにおける並列処理とその評価— 小粒度高並列オブジェクトモデルに基づくパラダイムについて —, 電子情報通信学会論文誌, Vol.J75-D-I, No.8, pp.723–739 (1992-8)

[8] Shapiro, E.：The Family of Concurrent Logic Programming Languages, *ACM Computing Surveys,* Vol.21, No.3, pp.413–510 (1989)

[9] Rokusawa, K., Ichiyoshi, N., Chikayama, T. and Nakashima, H.：An Efficient Termination Detection and Abortion Algorithm for Distributed Processing Systems, *Proc. ICPP'88 : International Conference on Parallel Processing,* Vol.I, pp.18–22 (1988)

[10] Chikayama, T.：Operating System PIMOS and Kernel Language KL1, *Proc. FGCS'92 : International Conference on Fifth Generation Computer Systems,* (June 1992)

[11] Ichiyoshi, N.：Parallel logic programming on the Multi-PSI, ICOT Technical Report TR-487, ICOT (1989), also presented at the Italian-Swedish-Japanese Workshop'90 (1990)

[12] Kimura, K. and Ichiyoshi, N.：Probabilistic analysis of the optimal efficiency of the multi-level dynamic load balancing scheme, *Proc. Sixth Distributed Memory Computing Conference,* pp.145–152 (1991)

[13] 日経データプロ編集：x86マイクロプロセサの全貌—8086からMicro2000まで, 日経BP社 (1990)

[14] 伊達 博, 大嶽 能久, 瀧 和男：並列オブジェクトモデルに基づくLSI配線プログラム, 情報処理学会論文誌, Vol.33, No.3, pp.378–386 (1992)

[15] 北沢 仁志：高配線率線分探索の一手法, 情報処理, Vol.26, No.11, pp.1366–1375 (1985)

[16] Date, H. and Taki, K.：A Parallel Lookahead Line Search Router with Automatic Ripup-and-reroute, *Proc. EDAC-EUROASIC93,* Paris (Feb.1993)

[17] Briner, J. V. *et al.*：Parallel Mixed-level Simulation using Virtual Time, *CAD accelerators,* pp.273–285, North-Holland (1991)

[18] Tham, K. *et al.*：Functional Design Verification by Multi-Level Simulation, *21st ACM/IEEE Design Automation Conference,* pp.473–478 (1984)

[19] Bryant, R. E.：A Switch-level Model and Simulator fro MOS Digital Systems, *IEEE Transactions on Computers,* Vol.C-33, No.2, pp.160–177 (1984)

[20] 可児 賢二 他：超LSI CADの基礎, オーム社 (1983)

[21] 樹下 行三 他：VLSIの設計I, II, 岩波書店 (1985)

[22] Misra, J.：Distributed Discrete-Event Simulation, *ACM Computing Surveys,* Vol.18, No.1, pp.39–64 (1986)

[23] Jefferson, D. R.：Virtual Time, *ACM Transactions on Programming Languages and Systems,* Vol.7, No.3, pp.404–425 (1985)

[24] 工藤 知宏 他：共有メモリを想定した並列論理シミュレータ，電子情報通信学会研究会報告，CPSY91-23, pp.151–158 (1991)

[25] 下郡 慎太郎, 鹿毛 哲郎：メッセージドリブンによる並列論理シミュレーション, 電子情報通信学会研究会報告, CAS88-110, pp.23–30 (1989)

[26] Burdorf, V. and Marti, J.：Non-Preemptive Time Warp Scheduling Algorithm, *ACM Operating System Review,* Vol.24, No.2, pp.7–18 (1990)

[27] 福井 眞吾：バーチャルタイムアルゴリズムの改良, 情報処理学会論文誌, Vol.30,No.12, pp.1547–1554 (1989)

[28] 松本 幸則, 瀧 和男：バーチャルタイムによる並列論理シミュレーション, 情報処理学会論文誌, Vol.33, No.3, pp.387–396 (1992)

[29] Soulé, L. and Gupta, A. : Analysis of Parallelism and Deadlock in Distributed-Time Logic Simulation, Stanford University Technical Report, CSL-TR-89-378 (1989)

[30] Dayhoff, H. and Hurst-Calderone : Composition of Proteins, *Atlas of Protein Sequence and Structure 5:3,* Nat. Biomed. Res. Found., Washington, D.C., pp.363–373 (1978)

[31] Needleman, S. B. and Wunsch, C. D. : A General Method Applicable to the Search for Similarities in the Amino Acid Sequences of Two Proteins, *Journal of Molecular Biology,* 48, pp.443–453 (1970)

[32] Murata, M. : Simultaneous Comparison of Three Protein Sequences, *Proc. Natl. Acad. Sci. USA,* Vol.82, pp.3073–3077 (1985)

[33] Carrillo, H. and Lipman, D. : The Multiple Sequence Alignment Problem in Biology, *J. Appl. Math.,* 48, pp.1073–1082 (1988)

[34] 広沢 誠, 星田 昌紀, 石川 幹人, 戸谷 智之：MASCOT— 3 次元ダイナミックプログラミングに基づいた蛋白質のアライメントシステム, 情報学シンポジウム講演論文集, 日本学術会議 (1992)

[35] 石川 幹人, 星田 昌紀, 広沢 誠, 戸谷 智之, 鬼塚 健太郎, 新田 克己, 金久 實：並列推論マシンを用いたタンパク質の配列解析, 情報処理学会 情報処理基礎研究会報告, 23-2 (1991)

[36] 戸谷 智之, 星田 昌紀, 石川 幹人, 新田 克己：並列 3 次元ダイナミックプログラミング法によるタンパクの配列解析, 電子情報通信学会研究会報告, COMP91-72, Vol.91, No.344 (1991)

[37] Kirkpatrick, S., Gelatt, C. D. and Vecci, M. P. : Optimization by Simulated Annealing, *Science,* Vol.220, No.4598, pp.671–681 (1983)

[38] 金久 實：シミュレーテッドアニーリングを用いたマルチプルアライメント法, 分子生物学会年会 (1989)

[39] 荻原 淳, 金久 實：パターン認識を取り入れたマルチプルアライメント法, 日本生物物理学会年会 (1990)

[40] 木村 宏一, 瀧 和男：時間的一様な並列アニーリングアルゴリズム, 電子情報通信学会研究会報告, NC90-1 (1990)

[41] Branting, K : Representing and Reusing Explanations of Legal Precedents, *Proc. Int. Conf. on AI and Law* (1989)

[42] Fujita, M. *et al.* : A Model Generation Theorem Prover Using Ramified-Stack Algorithm, ICOT TR-606 (1991)

[43] Inoue, K. *et al.* : Embedding negation as failure into a model generation theorem prover, ICOT TR-722 (1991)

[44] Stickel, M. E. : A Prolog Technology Theorem Prover: Implementation by an Extended Prolog Compiler, *Journal of Automated Reasoning,* 4:353–380 (1988)

[45] Manthey, R. and Bry, F. : SATCHMO: a theorem prover implemented in Prolog, *Proc. 8th Conference on Automated Deduction,* Springer Verlag (1987)

[46] 渕 一博：KL1 プログラミング雑感 — prover の並列化の体験より —, *Proc. KL1 Programming Workshop'90,* pp.131–139, ICOT (1990)

[47] Hasegawa, R., Fujita, H. and Fujita, M. : A Parallel Theorem Prover in KL1 and Its Application to Program Synthesis, *Proc. Italy-Japan-Sweden Workshop '90,* ICOT (1990), also ICOT TR-588 (1990)

[48] Fujita, H. and Hasegawa, R. : A Model Generation Theorem Prover in KL1 Using Ramified-Stack Algorithm, *Proc. ICLP'91 : 8th International Conference on Logic Programming,* pp.535–548 (1991), also ICOT TR-606 (1990)

[49] 藤田 正幸, Slaney, J., 長谷川 隆三：並列推論マシン上の並列定理証明系による有限代数の新事実 — 概要 —, ICOT Technical Memo TM-1221,1222, ICOT (1992)

[50] Fujita, M., Slaney, J. and Bennett, F. : Automatic Generation of Some Results in Finite Algebra, Proc. IJCAI-93 (1993) 掲載予定

[51] 藤田 正幸, 粂野 文洋：Mgtp による有限代数の新事実の発見, 並列処理シンポジウム JSPP'93 論文集 (1993)

[52] 長谷川 隆三, 越村 三幸：モデル生成型証明器 MGTP の and 並列化方式, ICOT Technical Report 申請中

[53] Hasegawa, R. and Fujita, M. : Parallel Theorem Provers and Their Applications, *Proc. FGCS'92 : International Conference on Fifth Generation Computer Systems,* pp.132–154 (1992)

[54] Hasegawa, R., Koshimura, M. and Fujita, H. : Lazy Model Generation for Improving the Efficiency of Forward Reasoning Theorem Provers, *Proc. International Workshop on Automated Reasoning,* pp.191–202 (1992), also ICOT TR-751

[55] Inoue, K., Koshimura, M. and Hasegawa, R. : Embedding Negation as Failure into a Model Generation Theorem Prover, *Proc. 11th International Conference on Automated Deduction,* pp.400–415, Springer Verlag (1992), also LNAI 607

[56] 越村 三幸, 長谷川 隆三：モデル生成型証明器上の様相命題タブロ, *Proc. Logic Programming Conference'91,* pp.43–52, ICOT (1991)

[57] Inoue, K., Ohta, Y., Hasegawa, H. and Nakashima, M. : Bottom-Up Abduction by Model Generation, ICOT TR-816, ICOT (1992), also in Proc. IJCAI-93 掲載予定

[58] McCune, W. and Wos, L. : Experiments in Automated Deduction with Condensed Detachment, *Proc. 11th International Conference on Automated Deduction,* pp.209–223, Springer Verlag (1992), also LNAI 607

[59] McCune, W. W. : *OTTER 2.0 Users Guide,* Argonne National Laboratory (1990)

[60] Per Martin-Löf : Intuitionistic Type Theory, *Studies in Proof Theory,* Lecture Notes (1984)

[61] Aït-Kaci, H. and Nasr, R. : LOGIN: A Logic Programming Language with Built-in Inheritance, *The Journal of LOGIC PROGRAMMING,* Vol.3, No.3 (Oct.1986)

[62] Schmidt-Schauß, M. : Computational Aspects of an Order-Sorted Logic with Term Declarations, *Lecture Notes in Artificial Intelligence,* Springer-Verlag (1989)

[63] 橋田 浩一, 竹沢 寿幸：自然言語処理における統合の諸相, コンピュータソフトウェア, Vol.8, No.6 (1991-11)

[64] Matsumoto, Y. : A Parallel Parsing System for Natural Language Analysis, *Proc. ICLP'86 : 3rd International Conference on Logic Programming,* London (1986)

[65] 佐藤 裕幸：並列自然言語構文解析システム PAX の改良, KL1 Programming Workshop '90, ICOT, Tokyo (1990)

[66] Yokota, K. and Yasukawa, H. : Towards an Integrated Knowledge-Base Management System — Overview of R&D on Databases and Knowledge-Bases in the FGCS Project, *Proc. FGCS'92 : International Conference on Fifth Generation Computer Systems,* Tokyo (June 1992)

[67] 横田 一正：演繹オブジェクト指向データベースについて, コンピュータ・ソフトウェア, Vol.5, No.4 (1992)

[68] Yasukawa, H., Tsuda, H. and Yokota, K. : Objects, Properties, and Modules in *QUIXOTE*, *Proc. FGCS'92 : International Conference on Fifth Generation Computer Systems,* Tokyo (June 1992)

[69] Kawamura, M., Sato, H., Naganuma, K. and Yokota, K. : Parallel Database Management System: Kappa-P, *Proc. FGCS'92 : International Conference on Fifth Generation Computer Systems,* Tokyo (June 1992)

[70] Makinouchi, A. : A Consideration on Normal Form of Not-Necessarily-Normalized Relation in the Relational Data Model, *VLDB* (1977)

[71] 三浦 孝夫：Theory of Non First Normal Form Relational Databases — A Survey, アドバンスト・データベースシステム・シンポジウム (1987)

[72] Schek, H.-J. and Weikum, G. : DASDBS: Concepts and Architecture of a Database System for Advanced Applications, Tech. Univ. of Darmstadt Technical Report, DVSI-1986-T1 (1986)

[73] Dadam, P. *et al.* : A DBMS Prototype to Support Extended NF^2 Relations : An Integrated View on Flat Tables and Hierarchies, *ACM SIGMOD* (1986)

[74] Verso, J. : VERSO: A Data Base Machine Based on Non 1NF Relation, INRIA Techinical Report 523 (1986)

[75] Yokota, K., Kawamura, M. and Kanaegami, A. : Overview of the Knowledge Base Management System (KAPPA), *Proc. FGCS'88 : International Conference on Fifth Generation Computer Systems,* Tokyo, Japan (1988)

編者あとがき

本書の企画は1991年はじめ頃に，bit編集部の小山透氏との間でもち上がった．当初は第五世代コンピュータ全般を扱う案もあったが，範囲が広すぎるので，編者がホームグラウンドとしていた並列処理に限定して，総集編になるような企画を立てることにした．

論文を手直しして簡単にまとめる程度のつもりで安請け合いした編者だったが，いつもの悪い癖で，詳細化するにつれて頑張った企画になってしまった．当時，FGCS'92に向けて並列処理のよい成果が続々と出始めており，それをほうっておけないというのが一つの理由だった．

FGCS'92が無事終わって一息ついた頃，秋までに原稿を完成させてもらうように執筆担当者にお願いした．このころ，編者の離任の日程が決まり，9月1日付けで神戸大学に移った．当然編集作業はこれからという段階だったのだが，新しい環境ではなかなかペースが掴めない．おまけに後期の講義ももつことになり，結局後期の授業が終わる2月まで編者の担当作業がほとんど進まず，執筆担当の方々には迷惑をおかけした．

一時は没かと思われていた企画がこのころから再開されたのだが，ネックになり続けたのは編者の執筆担当分だった．とくに発行日程もスケジュールされたあとの歴史編の追込みでは，bit編集部の石井徹也，ICOT OBの稲村雄，ICOTの戸谷智之，六沢一昭の各氏には，編集作業で多大のお世話になるとともに面倒をおかけした．この場を借りて心からお礼を言いたい．

週刊誌のような原稿取立てスケジュールだとも評されたこの時期であったが，ページ割も済んだあとに編者がページ数を超過したために，歴史編の各所にわたって大幅な切り詰めを余儀なくされた．このとき，本来なら引用すべき歴史上のイベントやそこで活躍した方々の名前を十分な検討の時間もなく割愛せざるを得なかった．私の話がなぜないのだろうと不思議に思われた諸兄には，そのあたりの事情をお察しのうえお許しいただければ幸いである．

ともあれ，プロジェクト成果の貴重な報告の機会を没にすることなく期限ぎりぎりで出版にこぎつけられたのは，ひとえに関係諸兄のお力添えの賜と深く感謝しているしだいである．

編者はICOT研究所設立の日から，まる10年と3か月ICOTに在籍したわけだが，これはちょうど30歳台を通してFGCSプロジェクトとともにすごしたことになる．

本書を編集するにあたり，またプロジェクトの閉幕にあたり，胸に去来するものは数多い．

編者はかつてプロジェクト中期のはじめ頃，マルチPSI計画が動き出したあたりに，こんな不尊なことを考えた覚えがある．「このプロジェクトはもしかして，自分のためのプロジェクトではなかろうか．」そのあたりから，マルチPSI/V1，マルチPSI/V2，PIM/m，そして並列応用に至る研究開発の一連の流れを，ほとんど自分の思うとおりに計画し歩んできた，あるいは引っ張ってきたような気がしている．そしてPIMで大規模な並列応用が動き評価が出たとき，これで自分のプロジェクトの1ラウンドは終わったというような思いがあった．

これは仕事を終えたある種の充実感であり，また技術に対する自信だったかもしれない．このような技術者冥利に尽きる仕事の機会を与えてくれたICOTと，当時の上司の方々に心から感謝の意を表するものである．

あのときの不尊な考えをいま冷静に振り返ってみると，もしかしたらあれは，渕所長(当時)の「それぞれの第五」のプロジェクトマネージメントに，うまくのせられた，いやのせてもらっていたのかもしれないということに思い至る．

PIMで大規模並列応用を動かしたときの，あの技術に対する自信は，本書を編集する大きなエネルギー

となっている．その自信とはすなわち，本書の主題であった「第五世代コンピュータの並列処理技術により，従来は難しかった広い対象領域での効率のよい並列処理が実現可能になる」ことである．その対象領域とは，知識処理に代表されるような，動的な計算と不均質なデータを扱う必要のある大規模問題であった．

これを可能にしている技術の最も重要なものが，KL1言語であり，KL1のプログラミング技術であり，さらにKL1言語処理系，PIMOS，PIMであるという話は，本書の第II，第III編で扱ったとおりである．けれどもここで，研究開発の当事者だからこそいうべきは，これらの技術はまだ決して理想の，あるいは究極の技術ではありえないことである．従来の並列処理を越える一通りの技術レベルにきてはいるけれども，まだまだ研究の余地は十分にあることを心すべきである．

編者の最初のバックグラウンドであったハードウェアの話をするならば，紀元2000年にはLSI 1チップ上のトランジスタ数は1億個に達する．それを用いて作られるプロセッサ1,000台を内蔵する並列マシンは机の脇に置くことができる．そして，その性能はいまのワークステーションのおよそ1万倍である．LSIの進歩から予想されるこの値は誇張でも何でもない，きわめて実現性の高い値なのである．

問題なのは，そのようなスーパーなハードウェアを動かすための，並列ソフトウェアの技術がまだよく見えていないことである．いや，第五世代コンピュータの並列処理技術によって，ソフトウェア技術の進むべき一つの方向性はずいぶん明らかにされたのではないかと思う．けれどもそれはまだ，方向性であって実際に進んで確かめ，改良を続けなければならないものである．それを続けることによってはじめて，広い応用領域で将来のスーパーなハードウェアを使いこなすことのできる，真の「汎用並列処理への道」が開かれるに違いないのである．

最後に本書の締めくくりにあたって，「第五世代コンピュータの並列処理は，汎用並列処理への近道である」というもう一つの仮説を読者諸兄にお贈りしたい．

本書をお読みになったあなたは，もしかしたらこれから私たちとともに，この仮説の真偽を検証し，真の「汎用並列処理への道」を切り開いてゆくパイオニアになるべき人かもしれない．

1993年6月

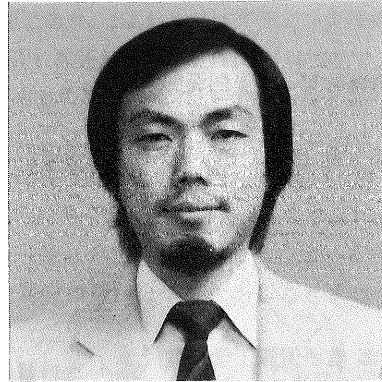
編　者

執筆者一覧

稲村 雄	日本電気技術情報システム開発 (株) ソフトウェア事業部 第一開発部
河村 元夫	ICOT研究所 第1研究部 主任研究員
越村 三幸	ICOT研究所 第1研究部 主任研究員
近藤 誠一	三菱電機 (株) 情報電子研究所 ビジネスコンピュータ開発部
瀧 和男	神戸大学工学部 情報知能工学科助教授
伊達 博	(株) 日立製作所 日立研究所 システム第1部 高度CAEシステム研究室
近山 隆	ICOT研究所 第1研究部 部長
戸谷 智之	ICOT研究所 第2研究部 研究員
新田 克己	ICOT研究所 第2研究部 部長
長谷川 隆三	ICOT研究所 研究担当次長
渕 一博	前ICOT研究所所長．現東京大学工学部 電子情報工学科教授
星田 昌紀	松下電器産業 (株) 東京情報システム研究所
松本 幸則	三洋電機 (株) 東京情報通信研究所 新情報処理技術研究室
屋代 寛	(株) 日立製作所 中央研究所 知能システム部
山崎 重一郎	(株) 富士通研究所 知識処理研究部 第1研究室
横田 一正	ICOT研究所 第2研究部 主席研究員
六沢 一昭	ICOT研究所 第1研究部 主任研究員
和田 久美子	沖電気工業 (株) 研究開発本部 総合システム研究所 知識情報処理研究部

(五十音順)

編 者 略 歴



瀧 和男 (たき かずお)

神戸大学 工学部助教授・工学博士

1952年 神戸生まれ
1979年 神戸大学 大学院工学研究科 修士課程修了
同年 ㈱日立製作所入社，制御用計算機 HIDIC の開発に従事
1982年 ㈶新世代コンピュータ技術開発機構(ICOT)に出向．逐次型および並列型推論マシンと並列応用プログラムの研究開発に従事
1986年 同機構主任研究員
1990年 同機構第一研究室室長
1992年 現職着任（工学部 情報知能工学科），現在に至る

並列マシンのアーキテクチャ，並列プログラミング，LSI CADなどに興味をもつ．高並列オブジェクトモデルに基づく並列処理システムと，LSI CAD分野への応用について研究中．

1993年7月号別冊

bit 別冊

第五世代コンピュータの並列処理

——汎用並列処理への道，言語・OS・プログラミング——

平成5年7月5日発行
編集人 小 山 透
発行人 南 條 正 男
印刷所 大 日 本 印 刷

発行所 **共立出版株式会社**
〒112 東京都文京区小日向4-6-19
Tel. 03-3947-2511
Fax. 03-3944-8182
振替 東京 1-57035

© 1993 Kyoritsu Shuppan Co., Ltd.

「bit」別冊

コンピュータサイエンスをいかに学ぶか

——情報分野への道案内——

有澤 誠・筧 捷彦・土居範久・廣瀬 健・深澤良彰・村岡洋一・安村通晃著

本別冊は，『bit』に連載(1991年5月号～1992年8月号)した「コンピュータサイエンスをいかに学ぶか」に加筆・訂正したもので，コンピュータサイエンスに興味をもつ大学新入生を意識しているが，広くこれからコンピュータサイエンスを学ぼうとしている人々を対象にした道案内的入門書。

“コンピュータサイエンスとは何か”から始まり，“どのようなものを，どのように学んでいけばよいか”など，その秘訣を斯界屈指の執筆陣が，ときにはユーモアを交えながら具体的に興味深く解説している。

目　次

1. コンピュータサイエンスとは
2. コンピュータサイエンスの学び方
3. コンピュータリテラシ-1 ——道具としてのコンピュータ
4. コンピュータリテラシ-2 ——エディタと清書システム
5. プログラミング 1
6. プログラミング 2
7. コンピュータアーキテクチャ
8. アルゴリズム
9. データ構造
10. オペレーティングシステム
11. コンピュータサイエンスの数学
12. コンピュータサイエンスの実験
13. 日本語技術文書構成法
14. コンピュータサイエンスの先端分野
15. 近未来に向けてのコンピュータサイエンス

B5判・144頁・定価2,400円(税込)／好評発売中

共立出版

認知科学ハンドブック

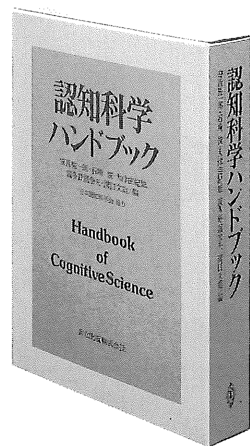
Handbook of Cognitive Science

認知科学は，人間の心についての探究の最新の体系的な試みであり，まだ生まれたばかりである。

本ハンドブックは，その研究分野全体として，また研究者個人として考えたときに便利で重要な知識の集大成であるとともに，大学院生や関連分野の研究者など，新規参入者の多少とも体系的な学習を促進し得るものとして，認知科学の主要な内容分野の現状を総合的でバランスよくまとめた。

編集委員

安西　祐一郎
石　崎　　俊
大津　由紀雄
波多野誼余夫
溝　口　文　雄
（日本認知科学会協力）



B５判・768頁・上製函入
定価23,690円（税込）

好評発売中
●内容見本送呈●

主な目次

- 第Ｉ編　相互作用　チーム航行のテクノロジー／状況に埋め込まれた認知と，学習の文化他
- 第II編　思　　考　メンタル・モデル／知識獲得におけるメカニズムとしてのアナロジ推論：発達的視点から他
- 第III編　意　　識　意識研究における問題と方向性／モジュラリティを超えて：生得的制約と発達的変化他
- 第IV編　感　　情　感情と行動のアージ理論
- 第Ｖ編　記　　憶　談話の理解とメモリ・モデル／辞書記憶：視覚と言語の接続他
- 第VI編　知　　識　知的マシーンの設計における参照問題のフレーム／常識とコンピュータ・システム他
- 第VII編　言　　語　言語の本質と使用と獲得について／音韻バリエーション他
- 第VIII編　読み書き　読みの両側性協同モデル／基本的な読みの過程：日本語と英語では異なるか他
- 第IX編　談　　話　メンタル・スペース理論：自然言語理解への新しいアプローチ／発話解釈と認知：関連性理論について他
- 第Ｘ編　視　　覚　３次元世界の再構成／視覚系の神経回路モデル他
- 第XI編　遂　　行　表象とパフォーマンス／「わざ」の形成他
- 第XII編　学　　習　言語獲得における立上げ問題／語の獲得他

共立出版

人工知能ハンドブック

全４巻

E.A.Feigenbaum 他編／田中幸吉・淵　一博 監訳

スタンフォード大学を中心とし，他の大学および人工知能研究センターから参加した研究者・大学院学生らによって，人工知能関係の全文献をほとんど網羅し，かつ広範囲にわたる分野を要領よく首尾一貫して体系づけを行なった世界初のハンドブック。

第Ｉ巻　Ａ５判・532頁・定価9,270円

●主な内容　**序論**（人工知能／人工知能ハンドブック／人工知能関係の文献）**探索**（概要／問題表現／探索法／探索プログラムの例）**知識表現**（概要／知識表現手法の概観／知識表現方式）**自然言語理解**（概要／機械翻訳／文法／解析／文章生成／自然言語処理システム）**話し言葉の理解**（概要／システム・アーキテクチャ／ARPA音声理解(SUR)プロジェクト）

第II巻　Ａ５判・584頁・定価9,270円

●主な内容　**人工知能研究用プログラミング言語**（概要／LISP／人工知能用言語の特徴／依存関係と仮説）**応用指向の人工知能研究：科学**（概要／TEIRESIAS／化学への応用／他の科学分野への応用）**応用指向の人工知能研究：医療**（概要／医療システム）**応用指向の人工知能研究：教育**（概要／ICAIシステムの設計／知能CAIシステム／教育への他の応用）**自動プログラミング**（概要／プログラムの仕様記述方法／基本的なアプローチ／自動プログラミング・システム）

第III巻　Ａ５判・848頁・定価13,390円

●主な内容　**認知のモデル**（概要／一般問題解決プログラム(GPS)／臨機応変的問題解決／EPAM／記憶の意味ネットワーク・モデル／信念システム）**自動証明**（概要／導出原理／自然演繹法／BoyerとMooreの証明システム／非単調論理／論理プログラミング）**コンピュータ・ビジョン**（概要／積木の世界の理解／視覚データの初期処理／シーンの性質の表現／コンピュータ・ビジョンのアルゴリズム／装置とシステム）**学習と帰納的推論**（概要／暗記学習／助言に基づく学習／例題からの学習／複数概念の学習／複数ステップ知識習得のための学習）**計画と問題解決**（概要／STRIPSとABSTRIPS／非階層的計画法／階層的計画法／素計画の精密化）

第IV巻　Ａ５判・768頁・定価15,450円　★最新刊！

●主な内容　**黒板システム**（概要/問題解決の黒板モデル/黒板アーキテクチャの発展他）**協調分散問題解決**（概要／CDPSの例／CDPSの重要なアプローチと経験的洞察/結論）**エキスパート・システムの基礎**（概要／基本原理／技術の現状他）**自然言語理解**（概要/ユニフィケーション文法／意味解釈他）**知識ベースを使ったソフトウェア・エンジニアリング**（概要／仕様獲得／プログラム合成他）**定性物理**（概要／定性演算則／定性演算則を用いた挙動の推論他）**知識に基づくシミュレーション**（進化・応用・システムの設計・定性的性質・実世界におけるアプリケーション・開発と使用における問題他）**コンピュータ・ビジョンの最近の話題**（低レベルの視覚処理/コンピュータ・ビジョンの進歩他）

共立出版

（定価は消費税込みです）

〈編集委員〉廣瀬　健・野崎昭弘・鈴木則久・小林孝次郎

情報数学講座 全15巻

計算機科学に役立つ数学を体系的にまとめることは，使用分野の広範さ，奥深さ，計算機科学の流動性などにより困難であるが，その発展のためには，数学的基礎をしっかり持つことが不可欠である。本講座は，計算機科学のための数学を大胆に整理・統合化することを目的として，定理・証明の羅列は控え，話題の展開や表現の仕方を工夫し，例題を豊富に用いて抽象的概念をわかりやすく解説している。

■第1回配本／第3巻

離散構造

根上生也著　234頁・定価2,884円

“離散構造”を有限集合が複合して創り出す構造ととらえ，その最も基本的な構造である“グラフ”を題材に組合せ論的な考え方や方法論を解説。

【目次】 グラフの基礎概念／グラフの連結性／グラフの内部構造／グラフの平面性／有向グラフ／代数的取り扱い／数え上げの手法／他

■第2回配本／第4巻

計算の理論

笠井琢美・戸田誠之助著　220頁・定価2,884円

チューリングマシン，帰納的関数，NP完全性，チャーチのテーゼなどを主題として，コンピュータと数学基礎論との橋渡しをする分野を紹介。

【目次】 計算可能性／実際的計算可能性／並列計算／確率アルゴリズム／近似アルゴリズム／歴史と今後の展望／他

■第3回配本／第14巻

最適化の手法

茨木俊秀・福島雅夫著　'93．7月刊行予定

情報科学の基礎分野である「最適化」の代表的な手法を解説したもので，特に最近研究が進められている新しい手法を重点的に取り上げた。

【目次】 線形計画：シンプレックス法／ネットワーク最適化／多面体アプローチ／非線形最適化／ニューラル・ネットワークと最適化／他

全巻の構成

1 応用論理　山田眞市著……続刊
2 情報代数　小野寛晰著……続刊
3 離散構造　根上生也著……定価2,884円
4 計算の理論　笠井琢美・戸田誠之助著……定価2,884円
5 言語と構文解析　徳田雄洋著……続刊
6 データ構造論　小林光夫著……続刊
7 プログラム意味論　横内寛文著……'93．11月刊行予定
8 プログラム検証論　林　晋著……続刊
9 プログラム言語の基礎理論　武市正人他著……続刊
10 知識と推論　森下真一著……'93．10月刊行予定
11 符号と暗号の数理　藤原　良・神保雅一著……'93．9月刊行予定
12 計算幾何学　今井　浩・今井桂子著……続刊
13 グラフィックスの数理　杉原厚吉著……続刊
14 最適化の手法　茨木俊秀・福島雅夫著……'93．7月刊行予定
15 性能評価の基礎と応用　亀田壽夫他著……続刊

■各巻＝A5判・平均230頁・上製本

◆続刊の書名，執筆者は変更される場合があります。

共立出版

（定価は消費税込みです）

IF-Solutions for Prolog
and Knowledge Engineering

IF/Prolog
進歩の楽しさ

楽しい仕事はいい仕事です。ソフトウェア開発はまさに手作業なので，楽しいほど生産性が高くなります。

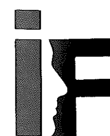
IF/Prologの設計はその楽しさを応援します：既存のソフトウェア資産との接続。ソフトウェア部品としての利用。常に変わっていくハードウェアに素早く対応。業界標準への積極的なサポート。
IF/Prologの品質はいい仕事の開発基盤です：開発者は効率よく経済的に目的を達成できます。最新のカーネルはもちろん，周辺ソフトウェアとの高度な接続機能を提供していますので応用の専門分野に集中できます。

インタフェイスコンピュータは技術と経済の調和を追求しています。
先端のソフトウェア工学，高品質，将来性，そして楽しさ。

IF: Perfection in Prolog

IF/Prologの主な機能：
- インタプリタ＋コンパイラ
- 全画面デバッガ
- 例外＋割り込み処理
- インタフェース
  - C言語(双方向)
  - (バックトラックを含む)
  - OS(双方向)
  - GKSグラフィックス
  - GNU-Emacsエディタ
  - SQL-データベース
  - X Window(オプション)
  - OSF/Motif(オプション)

IF/Prologの対応機種：
UNIX, AIX, OSF/1, VMS
MS-DOS, MS-Windows (386, 486)
SCO UNIX, UNIX SYSVR4 (386, 486)
(70社600機種以上対応)



インタフェイスコンピュータジャパン株式会社
〒113東京都文京区本郷3-21-10 浅沼第2ビル
Tel：(03)3818-5826　Fax：(03)3818-5829

InterFace Computer worldwide

Munich	Tel: +49-89-51086-0
Dresden	Tel: +49-351-336-1186
Tokyo	Tel: +81-3-3818-5826
Austin	Tel: +1-512-327-5344
Hong Kong	Tel: +852-544-8171