プライオリティ制御機構を有するOR並列Prologにおける負荷分散方式とその評価

内垣 雄一郎，林 彰吾
松田 秀雄，金田 悠紀夫
神戸大学工学部

本稿では，OR並列方式に基づいてPrologのゴールの実行を複数のプロセスに分配して並列に処理する処理系をLANによりワークステーションを結合した分散環境上に構築する場合の負荷分散方式に関して述べ，その性能評価を行っている．分散環境ではプロセッサ間の通信コストが高いため，負荷情報を集中制御し，負荷の発生によって分散の駆動を行う方式を提案しており，台数効果と，従来の逐次処理系との速度比較により性能評価を行っている．

Dynamic Load Balancing in OR-Parallel Prolog with Priority Control and Its Performance Evaluation

Yuichiro UCHIGAKI, Shogo HAYASHI
Hideo MATSUDA and Yukio KANEDA,
Faculty of Engineering, Kobe University

In this paper we discuss a metod for performing dynamic load balancing in or-parallel Prolog implemented on a loosely-coupled multicomputer system. Although loosely-coupled multicomputer systems take highly inter-proccessor communication cost, it has the advantage of scalability to any number of processors. We propose a method for dynamic load balancing with centralized control of load information on each processor. The effectiveness of our method is demonstrated by measuring performance compared with a sequential Prolog system.

1 はじめに

論理型言語であるPrologは，知識情報処理分野において有効なプログラミング言語である．知識情報処理は近年盛んに研究されている分野であり，これらが実用段階にはいれば扱う問題は飛躍的に大きくなることが予想される．従って，論理型言語にはよりいっそうの速度の向上が望まれている．速度の向上には，CPUの処理性能を上げることによって高速化を図るような，ハードウェア的なアプローチと，処理の並列化によって高速化を図るような，ソフトウェア的なアプローチがある．

以前，著者らは共有メモリを持つ密結合型の計算機上にプライオリティ制御機構を有するOR並列Prolog処理系を実装した[1]．この処理系では，積極的には負荷分散は行わず，複数のプロセッサが共有メモリ上におかれたキューにアクセスする形を取っていた．

本研究では，疎結合型の計算機システムにおいてOR並列Prolog処理系を実装した．疎結合型の計算機システムは，分散システムとも呼ばれ，密結合型の計算機システムに比べて台数拡張性に富むという利点があるが，計算機内部の処理速度に比べてプロセッサ間の通信にかかる時間が大きいという問題点があるため，通信コストを考慮した負荷分散を行う必要がある．また，本処理系では，探索を効率良く行うためにプライオリティによる実行順序制御機構を持っており，プライオリティ制御を考慮して負荷分散を行う必要がある．

2 OR並列Prologのプライオリティ制御

2.1 プライオリティ制御

OR並列は，OR関係にある節を並列に実行するもので，バックトラックの先取りと言える．

本処理系では，総ての節を並列に実行するわけではない．プログラム中でユーザにより並列実行するよう宣言された節についてのみ並列実行する．しかしながら，再帰を用いて書かれたプログラムの場合，生成されたプロセスが更にプロセスを生み，プロセスの数が爆発的に増えることになる．この際，探索空間の一部のみが深く探索されることになり，探索木全体の見通しも悪くなる．また，プロセスが不必要に増加することによってメモリを圧迫するようになる．

こうしたことを回避するため，並列実行を制御する種々の方式が提案されている[1, 2, 3, 4]．これらの中で，著者らは文献[1]で，プライオリティによる実行順序制御を提案している．プライオリティは整数値で，正負それぞれシステムで定められた範囲内で任意の値を取る．プライオリティはtask(3.3節参照)に一つずつ与えられている．処理系は，プライオリティが高いtaskから実行する．ユーザは組み込み述語によって実行中のtaskのプライオリティ値を変更することができる．従って，プライオリティ値を変更することによって，taskの実行順序を制御することができる．

プライオリティの設定は，次の3つの組み込み述語によって行う．

setPriority(X)	taskのプライオリティをXに設定する．
upPriority(X)	taskのプライオリティをX上げる．
downPriority(X)	taskのプライオリティをX下げる．

設定するのはこの述語を実行したtaskのプライオリティである．他のtaskのプライオリティはアクセスしない．

2.2 並列実行宣言

OR並列Prologでは，OR関係にある節を並列に実行するが，これらを総て並列に実行したのでは一つ一つの節の実行の粒度が小さくなり過ぎる．本処理系では，並列化するのはプログラムにおいて特別に宣言をした節に限定することでこの問題を解決している．

この宣言は，次のような形をしている．

:- para *predicate name/arity.*

ソースプログラムの形は図1のようになっている．

先頭の宣言により，述語p, qは並列実行する．従って，pが呼び出されると，q, r, sは各々別のtaskとして生成される．qについても同様である．

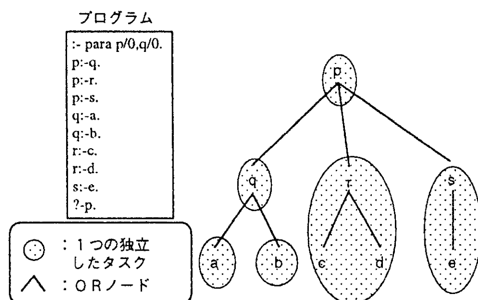

図 1: プログラムと並列実行木

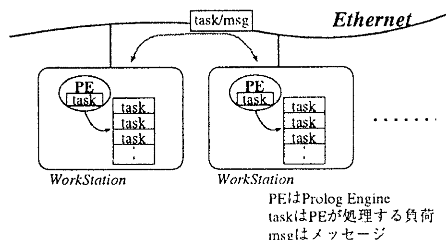

図 2: 実行環境

3 OR 並列 Prolog における負荷分散

3.1 実行環境

今回処理系を実現するに当たって想定した環境は，ワークステーションを LAN などの通信コストの高いネットワークによって結合した環境である．各ワークステーションのメモリは各々独立しており，共有メモリは持たない．プログラムは各ワークステーション上で動作し，メッセージによる通信によって負荷分散を行う．

処理系を実装したのは，TOSHIBA AS4025 をイーサネットで結合した環境上である．計算機内部での演算速度に対して，ネットワークを用いた通信では多くの時間がかかる．従って，実装は，通信にかかる時間の削減に注意を払いながら行った．

ワークステーション間の負荷の移動では，多くのデータを通信する必要がある．従って，負荷の移動は慎重に行わなければならない．そのため，負荷に関する情報はできるだけきめ細かくやりとりすることが望ましいが，そうすると通信量が大きくなってしまう．通信コストが高いので，1 回あたりの通信量を小さく抑えるとともに，通信回数も低く抑える必要がある．

3.2 負荷分散方式

OR 並列実行では，OR 関係にある節を各々独立に実行する．従って，これらを実行の単位とする．

これまでに提案された負荷情報の管理方式としては，集中管理方式と分散管理方式がある．前者は負荷情報を集中して管理する方式で，後者は複数の場所で管理する方式である．分散管理方式の例としては文献[5, 6, 7]がある．また，負荷分散を行うきっかけとなる方式に，要求駆動方式と発生駆動方式がある．前者は他のプロセスから負荷分散の要求を受けてそこに負荷を配分する．後者は負荷が発生した時点で負荷分散を行う．文献[5, 6]は要求駆動方式である．文献[7]は，拡散型の負荷分散方式を提案しているが，これは，自プロセッサが近傍の中で比較的忙しいと判断した場合に負荷の分散を行うものであり，発生駆動方式に近い．この方式は，大規模なマルチプロセッサ向きの方式であり，せいぜい十数台の並列処理環境には向かない．

要求駆動方式の場合，暇なプロセッサが task を受けとるまでに時間的な遅れが生じる可能性がある．また，本研究では，通信コストが高いことを前提としているため，task の要求などのメッセージ通信が実行時間に大きな影響を与えると考えるため，発生駆動方式を採用した．発生駆動であれば，task の要求等のメッセージ通信なしに負荷分散の指示を出せる．

本処理系では，集中制御による負荷分散を行う．集中管理方式のもっとも簡単な実現方法は，総ての task を 1 箇所で管理し，分配する方法であるが，この方法では，生成した task を一度負荷分散の管理を行うプロセス(以下，負荷分散プロセスと呼ぶ)に送り，再配分することになり 2 度の通信を必要とする．本処理系は通信コストが大きなシステムに実現しており，通信量の削減が必要不可欠である．そこで，負荷分散プロセスは，task 本体を持つのではなく，各プロセッサが保持する task 数，プライオリティ値などの情報を集中管理する．それに伴って，負荷

分散プロセスが負荷の移動を指示する 発生駆動方式 を採用する．また，プロセッサ台数はそれほど大規模ではなく，イーサネットにバス結合された環境に実装したため，拡散型の負荷分散は行わず，総てのプロセッサ間で負荷の移動を行うこととした．

3.3 Prolog Engine と task

Prolog プログラムを並列実行するプロセッサは，Prolog Engine (PE)と呼ばれる．PE は，ソフトウェアによる仮想マシンであり，実体としては，UNIX のプロセスである．PE は，Prolog の逐次処理系と同様に縦型の探索を行う[1].

PE が実行するのは task である．task は逐次的なゴールの実行である．プログラムの実行が並列実行宣言されている述語を実行すると，並列実行される節をそれぞれ一つの task として，新たに生成する[1].

task はそれぞれ独立して実行可能である．Prolog プログラムが実行につれて複数の task に分割され，これらが複数台起動された PE によって同時に実行されることによって，並列実行が行われる．

3.4 task の生成

プログラムの実行が，並列実行宣言されている節にさしかかると，PE は新たに task を生成する．task は，task control block (TCB)と呼ばれるブロックによって管理されている．TCB には，task の状態を表すのに必要な情報が格納されている．

生成した task は，一旦待ち行列に繋ぐ．この待ち行列をタスクキューと呼ぶ．タスクキューはプライオリティの値によってソートされている．PE はタスクキューからタスクを一つとって実行に移すのであるが，待ち行列の先頭から task を取るため，プライオリティの高い task が優先的に実行されることになる．この機構により，実行の順序制御を行っている．

3.5 プライオリティ制御と負荷分散

本処理系は共有メモリを持たない分散環境上に実現されている．PE は Prolog プログラムの実行に先だって各々の計算機上に UNIX の rsh によって起動される．起動された PE は次節に述べる負荷分散処理に従い task を送受信しながらプログラムの実行を進める．タスクキューは，PE が各々持っており，プライオリティによる制御は PE 単位で行う．従って，システム全体としてのプライオリティ制御は行っていないので，各 PE 間でプライオリティ値に極端な差がある場合は，負荷分散の際に補正する必要がある．

4 負荷分散処理機構の実現

4.1 負荷分散プロセス

負荷分散プロセスは，次の 3 つの処理を行う．

(1) 各 PE の task に関する情報の管理
(2) 各 PE への task 移動の指示
(3) 解の出力選択

各 PE は，タスクキューに繋がっている task の数，実行中の task のプライオリティ値，キューの先頭の task のプライオリティ値を負荷分散プロセスに送信する．負荷分散プロセスは，これらの情報を表の形で持ち，これを参照して task の移動を決定する．

負荷分散プロセスは，task 移動の指示を出すだけで，task の移動は当該 PE 間で行われる(図 3).

4.2 task 分配の決定方法

負荷分散プロセスは，前節で述べたように表を参照して各 PE に task 移動の指示を出す．移動先，移動元の決定条件は次の通りである．

task 移動元の決定条件

(1) 処理系全体で現在実行中の task の中で最も低いプライオリティ値よりも，プライオリティの高い task をタスクキューに持つ PE．
(2) タスクキューにタスクの数が最も多い PE．複数の PE がこれに該当する場合は，直前に移動を指示した PE 以外にする．

但し，タスクキューに繋がっている task の数が一定数よりも多くなければならない．

task 移動先の決定条件

(1) task のない PE．

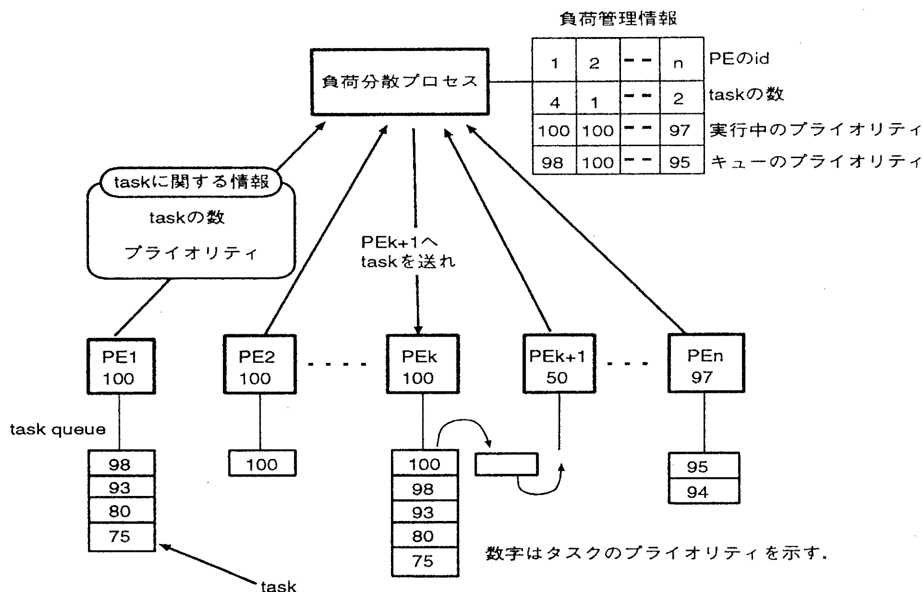

図 3: task に関する情報の集中管理と移動の指示

(2) 処理系全体で現在タスクキューにある task の中で最も高いプライオリティ値よりも，プライオリティの低い task を実行中の PE．
(3) タスクキューに繋がっている task の数が一定数よりも少ない PE．

移動する task は，移動元 PE のタスクキューの先頭に繋がれている task である．V.Kumar らは探索の深さ毎に 2 分割する方法等を示している[8]が，本処理系はプライオリティによって実行順序制御を行っているので，移動する task もこの制御を効果的に働かせるように選択している．

4.3 task の移動タイミング

task の移動は，一定時間間隔で行う他に，実行すべき task がなくなった PE が生じた時に行う．

本処理系は通信コストの高い環境に実現しているため，頻繁に task の移動を行うことは望ましくない．従って移動は一定の時間間隔をおいて行う．この間隔は，コンパイル時にユーザが指定できる．また，実行の途中で task がなくなる PE が生じる可能性もあるので，この様な場合，時間間隔による移動のタイミングを待たずに task の移動を行う．この際，1 つの task を送っただけではすぐに task が尽きてしまうことも考えられるので，可能であれば複数個の task を送る指示を出す．

4.4 通信量の削減

負荷分散プロセスには各 PE から task の数やプライオリティの情報が送られる．この量が多くなり過ぎると，負荷分散プロセスの処理能力を越えてしまう．また，通信時間が増大し，実行速度を落してしまう．従って，通信量はできる限り削減する必要がある．

プライオリティ制御を行うために，プライオリティ値は変更される度に通信する必要がある．しかし，タスクキューの task の数は，詳細に通信する必要がない．従って，前回の通信内容を記憶しておき，task 数の変化を知らせるのは数回に一度の割合で通信するようにする．但し，task 数が 0 になった時は，新たな task を獲得するため直ちに通信を行う．

評価プログラム 10–Queen で実験したところ，task 数が変化する度に通信を行った場合，全体

の 90%の情報が読み捨てられていた．これは，負荷の管理情報を参照する頻度に比べて情報の通信回数が多過ぎるためである．これを，変化 10 回に一度の通信にすることにより，読み捨てられる情報を 7%程度に抑えることができた．

4.5 解の選択機構

プライオリティ制御を用いることで，特別なアルゴリズムを用いずとも，全解探索せずに最適解を求めることができる．しかし，本処理系は通信量を減らすため，プライオリティ値に厳密に従った task の移動を行っているわけではないので，処理系全体で厳密にプライオリティ制御を行っているとはいえない．そこで，得られた解をプライオリティの順に出力する機構を設ける．各 PE 内では厳密にプライオリティ制御されているので，得られた解はその PE が探索した中で最適である．そこで，解が得られてもすぐに出力せずに，他の PE の状況を調べてプライオリティ順に出力する．この機能は現在のところ負荷分散プロセスに持たせている．

5 実行例による性能評価

5.1 Queen 配置問題

組合せ問題の例として，Queen 配置問題を解いた．実験では，10–Queen，負荷分散プロセスが task の移動を指示する間隔を 500[msec]，各 PE が task に関する情報を負荷分散プロセスに通信するのを task 数の変化 10 回に 1 回の割合と設定した．

実行時間と並列台数効果

先ず，並列台数効果について考える．実行時間，台数効果を図 4に示す．並列化しないプログラムでの実行時間は 43380[msec]であった．並列化したプログラムで PE1 台での実行時間の方が速いが，これは，並列化して task に分割することによってバックトラックが殆どなくなり，具体化された変数の未定義化などの処理が不要となったためと考えられる．

実験の結果から，台数効果は PE の台数が増えるにつれて線形に増加していることが分かる．PE9 台の時，約 6.1 倍の台数効果が得られている．全解探索問題の代表的な例である Queen 配置問題に対して，本処理系の負荷分散方式が有効であることが分かった．

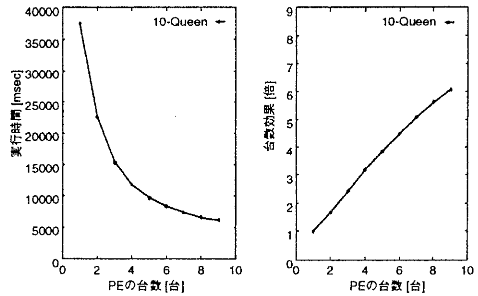

図 4: 実行時間と台数効果:10–Queen 問題

負荷の分散状況

次に，負荷の分散状況について検証する．この問題での総 task 数は，1353 個であった．

各々の台数での並列実行の際に，各 PE の中で最も多く task を実行した PE の実行 task 数を最大 task 数とし，総 task 数を PE 台数で割ったものを理論値として合わせて図 5に示す．

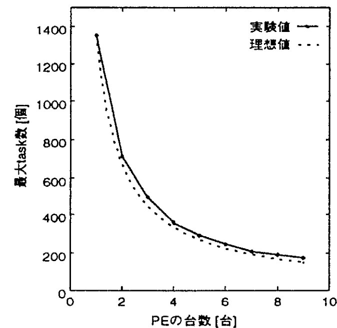

図 5: 最大 task 数:10–Queen 問題

図 5より，task はほぼ均等に各 PE に分散されていることが分かる．

PE9 台で実行した時の各 PE の保持する task 数の遷移を見てみると，実行開始時は task は 1 つのみでこれは PE1 が保持する．実行が進むと並列分岐して task 数が増える．そしてこれら

の task は各 PE に分散される．データから，実行開始時以外では各 PE の保持する task 数はほぼ一定となっており，task の分散がうまく行われていることが分かる．

5.2 巡回セールスマン問題

次に，組合せ最適化問題の代表例である巡回セールスマン問題をとりあげる．代表的な探索アルゴリズムに A*アルゴリズム[9]がある．A*アルゴリズムは最良優先探索の一種であり，問題固有の知識を使って探索効率を向上させるのが特徴である[10]．例えば，巡回セールスマン問題であれば，現在まで辿ってきた経路の重みによって，最適解となる可能性を推し量る．本処理系では，プライオリティ制御により，この重みをプライオリティとして設定することで，特別なアルゴリズムを用いることなく，容易に A*アルゴリズムを OR 並列実行環境で書くことができる[11]．

実行したプログラムは，task の粒度制御のためにプライオリティ制御と A*アルゴリズムを組み合わせたアルゴリズムによって記述した．都市数は 7，実行時の設定は Queen 配置問題と同じである．この問題の可能解は 720 個，最適解は 1 個である．また，解の選択機構を利用して，最適解だけを求めるようにした．PE の台数を 1 台から 9 台まで変化させた結果，総ての場合において正しい最適解が得られた．

実行時間と並列台数効果

実行時間と台数効果を図 6 に示す．並列化しないプログラムでの最適解探索時間は 23759[msec] であった．並列化したプログラムでの PE1 台の実行の方が速いのは，先の Queen 配置問題と同様の理由の他に，並列化して複数の task に分け，プライオリティ制御することによって最適解に近いと思われる task が優先的に実行された結果である．

実験の結果から，PE9 台で約 5.8 倍の台数効果が得られていることが分かる．更に，PE2 台の場合で約 3.2 倍の台数効果が得られている．これは，探索空間が複数の task に分割され，プライオリティ制御機構に基づく負荷分散の結果，見込みのある部分を複数の PE で効果的に探索できるからである．このことから，プライオリティ制御を行うことによって，最適解の探索効率が向上することが分かる．

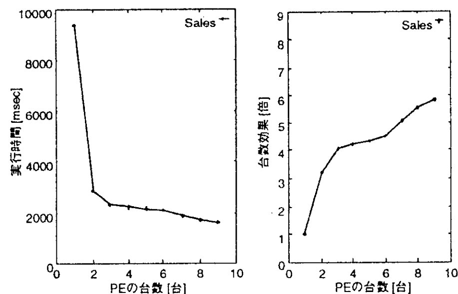

図 6: 実行時間と台数効果:巡回セールスマン問題(7 都市)

また，PE の台数を 4 台以上にしてもこれほどには台数効果が伸びていないが，これは問題の規模がそれほど大きくなかったためであると考えられる．現在，本処理系はメモリ容量等の制約上，大きな問題を実行することができない．これらの改善は今後の課題である．

5.3 他の処理系との比較

既存の処理系との比較として，SICStus Prolog ver.2.1[12]と処理速度の比較を行った．

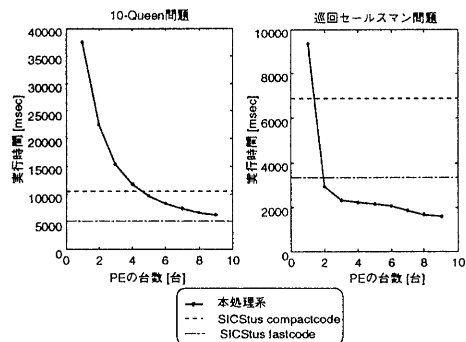

図 7: 本処理系と SICStus Prolog との処理速度比較

本処理系において，PE1 台で実行した場合は，SICStus Prolog に比較して遅い．これは，SIC-

Stus Prolog がコンパイル時に様々な最適化を行って効率の良いコードを生成するからである.

Queen 配置問題では, PE 4 台以上で compact code[1]に比して速くなる. これは並列台数効果のためである. 巡回セールスマン問題では, PE 2 台以上で fast code[2]よりも速くなる. これは, 並列台数効果とともに, プライオリティ制御による探索効率の向上によるところが大きいと考えられる.

6 結論

論理型言語は, 他のプログラミング言語にはない特徴を持っている. それ故に, 人工知能等の知識情報処理分野での期待も大きい. 並列処理は, 処理速度の向上に有効な手段である.

本研究では, 分散環境下での並列実行に着目した. 分散環境下での並列実行は, 通信コストが高いという欠点を持つが, 台数拡張性に優れており, 特別なハードウェアを必要としないので, 柔軟に並列処理環境を構築することができる. この様な環境下では, 負荷分散のアルゴリズムが重要である. 本処理系では, 負荷分散専用のプロセスを用いることによって負荷分散を効率良く行うことができた. そして, 処理系全体に渡ってプライオリティが生かされるように, 解をプライオリティの順に出力する機構を設けた. これにより, 特別なアルゴリズムを用いることなく最適解を効率良く求めることができる.

評価プログラムによる性能評価を行ったところ, Queen 配置問題では, PE 9 台で, 約 6.1 倍の台数効果を得た. また, 負荷分散の状況を調べたところ, 理想に近い形で負荷分散が行われていることが分かった. 巡回セールスマン問題では, PE 9 台で約 5.8 倍という台数効果を得, PE 2 台の時には約 3.2 倍と, 線形以上の効果が現れた. これは, プライオリティ制御により, 最適解の探索が効率良く行われたためであると考えられる.

参考文献

1) 松田秀雄, 鈴鹿重雄, 金田悠紀夫: OR 並列 Prolog におけるプライオリティ制御機構とその応用, 情報処理学会論文誌, Vol.34, No.4, pp.773–781(1991).

2) Reynolds, T.J. and Kefalas, P.: OR–Parallel Prolog and Search Probrems in AI Applications, Logic Programming, *Proc. 7th Int'l Conf.*, MIT Press, pp.340–354(1990).

3) Szeredi, P.: Exploiting Or–Parallelism in Optimisation Probrems, *Proc. Joint Int'l Conf. and Symp. on Logic Programming*, MIT Press, pp.703–716(1992).

4) 長塚雅明, 今野和浩, 小林直樹, 松岡聡, 米澤明憲: 超並列計算機上の並列制約論理型言語 PARCS の柔軟な探索枝制御と高効率な枝刈り法, 並列処理シンポジウム JSPP'94, pp.327–334(1994).

5) 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, 情報処理学会研究報告, ARC79–10(1989).

6) 古市昌一, 中島克人, 中島浩, 市吉伸行: スタック分割動的負荷分散方式とマルチ PSY 上での評価, 電子情報通信学会技術研究報告, CPSY91–8(1991).

7) 佐藤令子, 佐藤裕幸, 中島克人, 田中千代治: 疎結合型マルチプロセッサ上の拡散型動的負荷分散方式, 情報処理学会論文誌, Vol.35, No.4, pp.571–580(1994).

8) Rao,V.N. and Kumar,V: Parallel Depth First Search. Part I. Implementation, *International Jornal of Parallel Programing*, Vol.16, No.6, pp.479–499(1987).

9) Hart,P.E., Nilson,N.J., and Raphael,B.: A Formal Basis for the Heuristic Determination of Minimun Cost Paths, *IEEE Trans. Syst. Sci. Sybern.*, Vol.SSC–4, No.2, pp.100-107(1968).

10) 松田秀雄, 鈴鹿重雄, 金田悠紀夫: プライオリティ制御機構を有する OR 並列 Prolog の探索問題への応用, 並列処理シンポジウム JSPP'93 , pp.55–62(1993).

11) 松田秀雄, 金田悠紀夫: プライオリティ制御機構を有する OR 並列 Prolog の分子系統樹作成への応用, 情報処理学会論文誌, Vol.35, No.4, pp.658–665(1991).

12) *SICStus Prolog User's Manual*, Swedish Institute Computer Science(1993).

[1] 中間コードに変換されるコンパイルモード.

[2] 機械語まで変換されるコンパイルモード. Sun–3,Sun–4,NeXT でのみ使用可.