ICOT Technical Memorandom: TM-1058

TM-1058

KL1プログラミングワークショップ '91予稿集

市吉　伸行、松本　幸則
和田　正寛

June, 1991

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

Institute for New Generation Computer Technology

序

第五世代コンピュータ・プロジェクトの初期において，核言語に関する活発な研究が続けられ，その中で生まれたのが，GHC であり，KL1 であるが，今日のプロジェクトの基礎が，その時に固められたと言っても過言ではない．其の後，言語と並んで，Multi-PSI が完成し，KL1 のユーザが増えてくるにしたがって，プログラミング技術の開発も急速に進んだ．昨年の第一回ワークショップは，丁度その本格化がスタートした時点に開催された．Prolog の文化が成熟期を迎えたことと対比して言えば，KL1 の文化は成長期にあると言えよう．

プログラミング言語はソフトウエア文化の中核であり，言語の普及を通して，始めてその文化が大きく開花をすると言うことは，良く知られている．これまでの歴史において，我が国で開発されたプログラミング言語が世界的に認知された例は残念ながらまだないが，今我々が手にしている GHC/KL1 は，その有力な候補となりつつある．現に，GHC は，世界の研究者から，理論，実装，応用の各側面で，注目を浴びている．筆者は，今年の二月に，Miami で開催された Artificial Intelligence Application のコンファレンスにおいて，KL1 による応用プログラムを中心とした招待講演を行ったが，そこで，人工知能の専門家からも好意的な反応を得た．それは嘗てなかったことであり，我々の研究成果が，始めて目に見える形で本格化してきたことの現れであると言えよう．

この時期に，第 2 回の KL1 ワークショップを開催する意義は，大変大きい．本ワークショップが，KL1 プログラミングの更なる発展の契機となることを期待したい．

KL1 プログラミングワークショップ '91
プログラム委員長　　古川 康一

プログラム委員会

プログラム委員長
古川 康一

幹事
市吉 伸行

委員

相場 亮	稲村 雄	上田 和紀
河村 元夫	越村 三幸	瀧 和男
谷口 尚	近山 隆	西ヶ谷 茂
新田 克己	萩原 馨	星田 昌紀
松本 幸則	六沢 一昭	和田 正寛

KL1 Programming Workshop '91

プログラム

日時: 平成 3 年 5 月 28 日 (火) 9:50 ～ 20:00
5 月 29 日 (水) 9:30 ～ 18:00

会場: (財) 新世代コンピュータ技術開発機構 アネックス会議室
〒 108 東京都港区三田 1 丁目 4 番 28 号 三田国際ビル
電話: 03-3456-3193 (ICOT 21F), 03-3769-1298 (ICOT アネックス)

主催: (財) 新世代コンピュータ技術開発機構

5 月 28 日 (火)

1　9:50~10:00　『開会』　古川康一

1.1　『開会の辞』　古川康一

2　10:00~12:00　『セッション (1): チュートリアル』

2.1　『KL1 処理系実装方式について』 1
稲村雄 (ICOT)

2.2　『涙なしの KL1 プログラミングのために』 8
近山隆 (ICOT)

2.3　『スケーラブルな並列プログラムのすすめ』 15
市吉伸行 (ICOT)

12:00~13:30　昼食休憩

3　13:30~15:30　『セッション (2): LSI-CAD プログラム』　座長: 瀧和男

3.1　『協調型論理設計エキスパートシステム co-LODEX の試作』 21
澤田秀穂、箕田依子、滝沢ユカ、丸山文宏 (富士通)

3.2　『バーチャルタイムによる並列論理シミュレーション』 29
松本幸則、瀧和男 (ICOT)

3.3　『電子回路レイアウトシステム co-HLEX の開発状況』 37
渡辺俊典、小松啓子 (日立製作所)

3.4　『並列オブジェクトモデルに基づく LSI 配線プログラム』 43
伊達博、大嶽能久、瀧和男 (ICOT)

15:30~15:50　休憩 (coffee break)

4　15:50~17:30　『セッション (3): 負荷分散』　座長: 六沢一昭

4.1　『スタック分割動的負荷分散方式とマルチ PSI 上での評価』 51
古市昌一、中島克人、中島浩 (三菱電機)、市吉伸行 (ICOT)

4.2　『囲碁対局システム・並列版『碁世代』の試作』 59
清慎一 (ICOT)、沖広明 (未来技術研究所)、瀧和男 (ICOT)

4.3　『Iterative-Deepening A* の並列化とその評価』 68
和田正寛、市吉伸行 (ICOT)

4.4 『疎結合マシン上での確率的アルゴリズムの並列化』 75
岩山登、佐藤健 (ICOT)

17:30~18:15 休憩 (食事と懇談)

5 18:15~20:00 『セッション (4): KL1 なぜなに教室』

『KL1/PIMOS のここをこうして欲しい』

司会: 和田 正寛

要望側: KL1/PIMOS ユーザ代表

回答: 上田 和紀 、稲村 雄、 PIMOS グループ代表

5 月 29 日 (水)

6 9:30~12:00 『セッション (5): 遺伝子情報処理 / 知識処理』 座長: 新田克己

6.1 『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』 83
戸谷智之、星田昌紀、石川幹人、新田克己 (ICOT)

6.2 『並列処理マシン上でのシミュレーティッド・アニーリング』 93
荒木均、舘野峰夫、加藤等、間藤隆一 (松下電器産業)

10:30~10:40 休憩

6.3 『制御用エキスパートシステムのマルチ PSI 上での実現』 101
鈴木淳三、小沼千穂、岩政幹人、市川哲彦、末田直道 (東芝)

6.4 『適応型電子装置診断システムにおける並列処理』 109
太田禅、大石貢、田中淳 (日本電気技術情報システム開発)、
田中みどり、中茎洋一郎、古関義幸 (日本電気)

12:00~13:20 昼食休憩

7 13:20~14:40 『セッション (6): 言語 / 基本ソフトウェア / ユーティリティ』
座長: 堀内謙二

7.1 『KL1 上の並列プロセス指向言語 *AYA*』 117
寿崎かすみ、近山隆 (ICOT)

7.2 『並列データベース管理システムの問い合わせ処理』 126
河村元夫 (ICOT), 佐藤裕幸 (三菱電機)

7.3 『メタプログラミングのための KL1 ユーティリティ』 133
越村三幸 (東芝情報システム)、藤田博 (三菱電機)、長谷川隆三 (ICOT)

14:40~15:00 休憩 (coffee break)

8 15:00~16:00 『セッション (7): 自然言語処理』 座長: 田中裕一

8.1 『並列自然言語解析における並列協調の効果について』 141
山崎重一郎 (富士通)

8.2 『並列一般化 LR パーザの性能評価』 147
沼崎浩明 (東京工業大学)、池田朋男 (東芝)、田中穂積 (東京工業大学)

16:00~16:10 休憩

9 16:10~17:50 『セッション (8): 問題解決器』 座長: 相場亮

9.1 "The Concurrent Constraint Language GDCC and Its Parallel Constraint Solver" . 155
David J. Hawley(ICOT)

9.2 『KL1 による 1 階述語論理プルーバ – その後の MGTP–』 166
長谷川隆三、藤田正幸 (ICOT)、藤田博 (三菱電機)、越村三幸 (東芝情報システム)

9.3 『並列 ATMS』 169
中島誠 (JIPDEC)、太田好彦、井上克己 (ICOT)

10 17:50~18:00 『閉会の辞』 渕一博

KL1処理系実装方式について

稲村 雄
ＩＣＯＴ第一研究室

概要

マルチPSIにおける現KL1処理系実装方式の特徴について述べる。

1 プロセッサ内処理概要

KL1の実行は基本的にはゴールのリダクションサイクルとして表現できる。各ゴールは優先度に応じた**ゴールスタック**(ゴールレコードをポインタで繋いだものとして表現)に保持され、各ゴールスタックは**優先度別ゴールスタックエントリテーブル**によって管理される。

また最高優先度ゴールスタックは**GSP(ゴールスタックポインタ)**と呼ばれるレジスタにキャッシュされ、通常時のゴールエンキュー/デキュー処理にメモリアクセスが発生することを避けている(図1)。

各リダクションサイクルは以下のように行われる。

デキュー: 最高優先度ゴールスタック(GSPによって保持)トップからゴールを取り出す処理。

— ゴールの引数は引数レジスタに展開され、以降の引数に関する処理はこのレジスタに対して行われる。

ガード部実行: 当該述語中の各クローズのコミット可不可を逐次にチェック。各クローズ毎に

1. 入力引数の具体化/値を述語の引数並びに従って逐次にテスト。

   ⇒ **未具体化変数発見:** 当該変数を中断要因候補変数として登録後、次のクローズのチェックを実行(中断)

   ⇒ **値のマッチに失敗:** 次のクローズのチェックを実行(失敗)

   注1: 未具体化変数同士の同一性はチェックしていない。そのため、

   ```
   ?- p(X,X).
   p(X,X) :- true | ・・・
   ```

   という呼び出しは成功せず中断する。

   注2: チェックの逐次性のため、失敗が検出されずに中断することがある。

   ```
   ?- p(X,1).
   p(1,2) :- true | ・・・
   ```

   という呼び出しは失敗せず中断する。

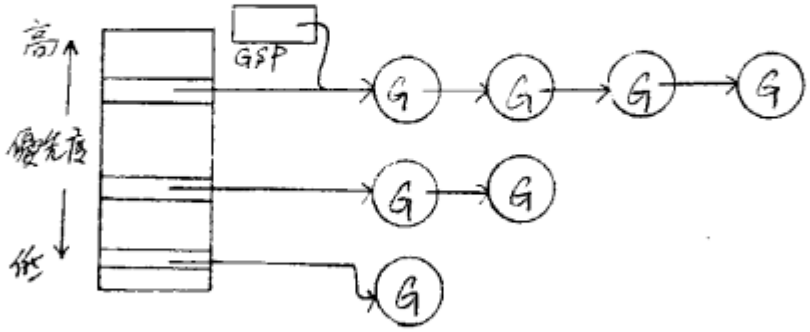


図1: ゴールスタックエントリテーブル

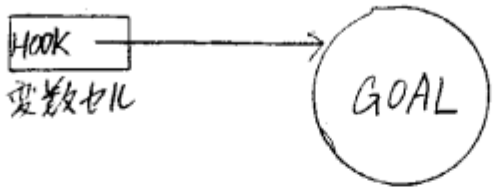


図 2: 変数へのゴールのフック

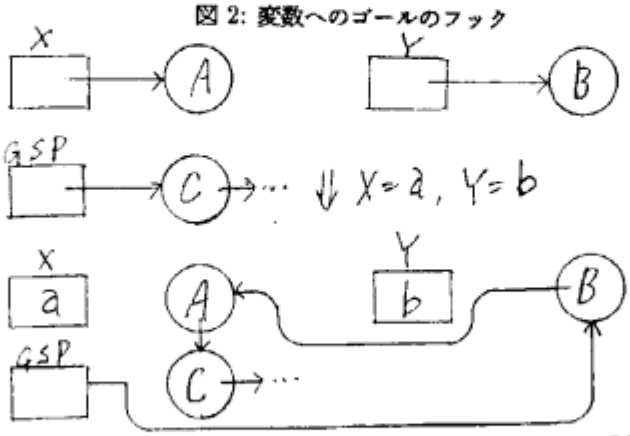


図 3: ユニフィケーションの実行順序とリジュームされたゴール順序の関係

2. **あるクローズの全てのガード部実行が終了** ⇒ そのクローズがコミットされる。(ボディ部実行へ)
3. **ある述語の全てのクローズが中断もしくは失敗した場合**

   ⇒ **中断要因候補変数が存在する:** 当該変数にゴールをフック。具体化を待たせる。(図 2)

   ⇒ **中断要因候補変数が存在しない:** 当該述語呼び出しの失敗。例外事象として処理系に報告する。

ボディ部実行: コミットされたボディ部を実行

1. ボディユニフィケーション: 基本的にはクローズに書かれた順序で、ボディ部の最初に実行される。未具体化変数を具体化する場合のような単純な処理は直ちに実行され、当該変数にフックしたゴールは、その場でゴールスタックにプッシュされる。(**リジューム処理**)

   注: ゴールがスタック形式で保持されているため、1 クローズ中の複数のボディユニフィケーションで複数個のゴールがリジュームされる場合、各ゴールの優先度が同じなら、ユニフィケーションの順番とは逆順に各ゴールが実行されることになる (図 3)。

   また、リスト同士のユニフィケーションのように複雑な処理となる場合、当該ユニフィケーション処理は

   ```
   unify_list([X1|X2],[Y1|Y2]) :- true | X1 = Y1, X2 = Y2.
   ```

   のようなシステムで用意された特別なコードを実行するようなゴールに変換され、その実行が遅らされる場合がある。

   注: あるユニフィケーションが複雑か否か、はシステムによって任意に判断される事項である。また、前注にあったリジューム後の実行順序についても現マルチ PSI 処理系独自のものと認識するべき事柄であり、実行の成否をこれらに頼ったプログラミングは避けるべきであろう。

2. ボディ組込述語: ボディ組込述語はユニフィケーションとユーザ定義述語呼び出しとの間で、基本的にはクローズに書いた順序で実行される。

   ただし、入力引数が具体化されていなかった場合には、組込述語の実行はシステムで用意した当該述語のみを実行する特殊なコードを実行するゴールの生成によって代用され、そのゴールを当該未具体化変数にフックすることで、入力変数の具体化を待つことになる。

3. ユーザ定義述語呼び出し: ユーザ定義述語はクローズに書かれた逆順にゴールスタックへプッシュされる。(**ゴールのエンキュー処理**) ゴールスタックの管理が LIFO であるため、結果的にはクローズに書かれた順番に実行が行われることになる。(4)

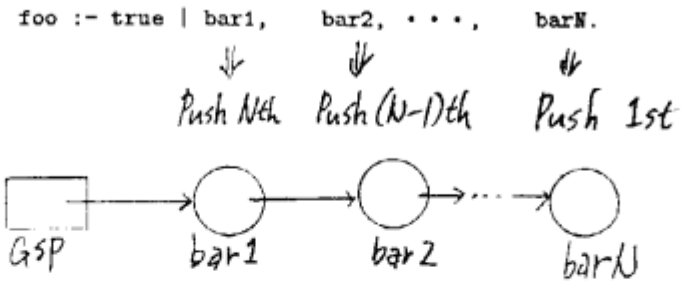


図 4: ユーザ定義述語呼び出し順序

4. テイル・リカージョン・オプティミゼーション: 最適化のため、クローズ中の最初に書かれたユーザ定義述語に関しては、ゴールスタックへのプッシュを行わず、引数を引数レジスタに載せたままで次の述語呼び出しが行われる。これを **TRO(テイル・リカージョン・オプティミゼーション)** と呼び、これによって 1 ゴールのエンキュー / デキュー処理の手間を省略することができている。

以上のようなリダクションサイクルは割り込み (他プロセッサからのメッセージ到着等) が検出されるまで続行される。コンテキスト・スイッチの容易さから、そのような割り込み検出はリダクションの切れ目にのみ行われるようになっており (**スリットチェック**)、割り込み検出時には、その要因のハンドラによって割り込みを解消した後に、デキュー処理から実行が再開されることになる。

注: 以上の説明のように、各述語のクローズをチェックする順序、各クローズの引数チェックの順序、およびボディ部の実行順序は使用するコンパイラ & 処理系によって一意に定まっている。

このことは **1 プロセッサのみを使用する場合、プログラムの実行は完全に決定的である**ということを意味している。しかし、このような決定性はコンパイラもしくは処理系によって任意に変更され得るものであり、**従ってこのような決定性を計算の成功に必須なものとして使用するプログラミングは止めるべきである**と言えよう。

2 データ構造

マルチ PSI 上の KL1 処理系において使用される各種データ型について概説する。

2.1 変数

具体値に束縛される前の変数はメモリ上の 1 ワードセルとして表現される。これらの変数は任意のデータによって具体化し得る。ゴールレコード、引数レジスタには当該セルへのポインタが格納される。

2.2 アトミックデータ

本処理系中で 1 ワード[1]で表されるデータの総称。ゴールレコード、引数レジスタなどに直接載り得るデータである。

- アトム
- 整数
- 浮動小数点数[2]

が一般的に使用されるアトミックデータである。

なお、{}(要素サイズ 0 のベクタ) は 1 ワードで表現できるため、ユニフィケーション等の処理ではアトミックデータと同様に処理されるようになっている。

[1] タグ (1 byte) + データ (4 byte) で表される 40 bit データ

[2] マルチ PSI の場合。PIM では倍精度浮動小数点数のみをサポートするため、浮動小数点数は 2 ワードデータとして表される。

2.4 ベクタ

一次元配列であり、メモリ上の連続した要素サイズ分のワードとして表現される。

なお、マルチ PSI の場合、要素サイズ7以上のベクタはミュータブルアレイとして表され、MRB が黒い場合でも一定コストでベクタの更新処理が行えるような仕組みとなっている。

2.5 ストリング

いわゆる文字列型データであり、各要素は各種サイズ (現状では 1 bit、8 bit、16 bit、および 32 bit の 4 種類をサポート) の整数に限定される。

2.6 コード / モジュール

プログラムコードをコンパイルした結果であるモジュールも、また、処理系中では単なるデータとして取り扱われる。すなわち、ゴールレコードやその他のデータから参照され、実体が他 PE にある場合には必要に応じて転送されることになる (PE 間処理参照)。

3 処理系実装における留意点

マルチ PSI 上に KL1 処理系を実装する際に特に留意した点は、手続型言語に比較してオーダーの悪くならない処理の実現であった。

そのために導入されたのが MRB、ストリームマージャなどの仕組みである。ここではこれらについての概説を行う。

3.1 MRB

論理型言語の枠組の中で構造体の破壊的要素更新を可能にするために考案された機構である。

基本的には、参照パスが唯一である構造体の要素を更新する場合には、更新処理完了後、オリジナルの構造体への参照は消滅するため破壊的要素更新を行ってもプログラムの論理性を損なわずに済む、ということであり、データの単一参照性を実行時に判定するための MRB(Multiple Reference Bit) と呼ばれる 1 bit データの導入により、この破壊的要素更新が可能となった。

MRB は、全ての参照ポインタに付加される 1 bit の情報であり、あるポインタの MRB がオフならば、そのポインタによって指されるデータは単一参照と保証されるように処理が行われる。

注: 未具体化変数に関しては最大 2 個までの MRB オフ参照が許される。これは、未具体化変数には具体化パスと読み出しパスの二つの参照があるのが一般的だからである。この辺りの詳細に関しては参考文献 [1] を参照のこと

3.1.1 MRB に関する処理

MRB に関する処理はコンパイラによって静的に決定される。

参照数増加時: ヘッド中の変数がボディに 2 回以上現れる場合、もしくはボディに初出変数が 3 回以上現れる場合 ⇒ MRB をオンにする命令を発行。

```
foo(X) :- true | bar1(X), bar2(X).
foo :- true | bar1(X), bar2(X), bar3(X).
```

参照数減少時: ヘッド中の変数がボディで使用されない場合 ⇒ 当該データを回収する命令を発行。

```
foo(X) :- true | true.
```

注: このことから MRB をオンにしないために、若干トリッキーな組込み述語が幾つか導入されている。例えば、ボディの string_element/4 述語がそれであり、

```
string_element(String,Position,Element,NewString)
```

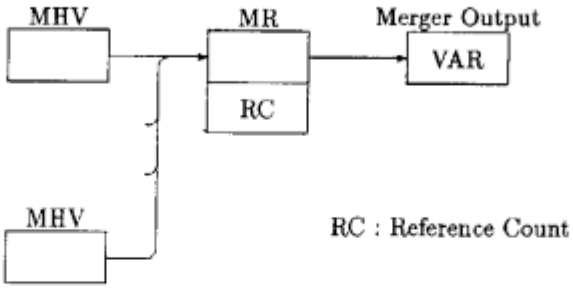


図 5: 組込みストリームマージャの構造

において、NewString という引数には入力の String がコピーされるだけであるから、本来不要なものであり、3 引数述語で用が足りるものなのであるが、3 引数述語を使った

```
foo(S,N) :- true | string_element(S,N,Elm),
                   bar(Elm), foo(S,~(N-1)).
```

のような用法では、引数 S に対して MRB をオンにする命令が発行されてしまうため、このような不都合を回避するために第 4 の NewString という引数の追加が必要となったのである。

3.2 一定コストストリームマージャ

論理型言語で一つのリソースを複数のプロセスでシェアするための機構として考案されたのが **ストリームマージャ**である。通常、ストリームマージャは

```
merge([], In2, Out) :- true | Out = In2.
merge(In1, [], Out) :- true | Out = In1.
merge([X|In1], In2, Out) :- true |
                Out = [X|Out2], merger(In1,In2,Out2).
merge(In1, [X|In2], Out) :- true |
                Out = [X|Out2], merger(In1,In2,Out2).
```

のように言語自体で定義されて実現されるが、この方法には

- 一要素入力毎の merge ゴールのリジューム / 中断に必要なコストが大きい
- マージインするストリーム本数を動的に増減させるためには、ネストした merge プロセスが必要となるため、処理のオーダーが悪くなる

といった欠点がある。そこで、以上の欠点を避けるために本処理系では merge/2 組込み述語と、その述語によって作られるマージャプロセスのための特殊な構造を用意した (図 5)。
この構造により、

- 入力にユニファイされるリストデータによって、即時に出力が具体化される。
- 入力にユニファイされるベクタデータによって、マージインするストリーム数を動的に増加できる。

という効果が得られ、上に書いた欠点を解消することができた。[2]

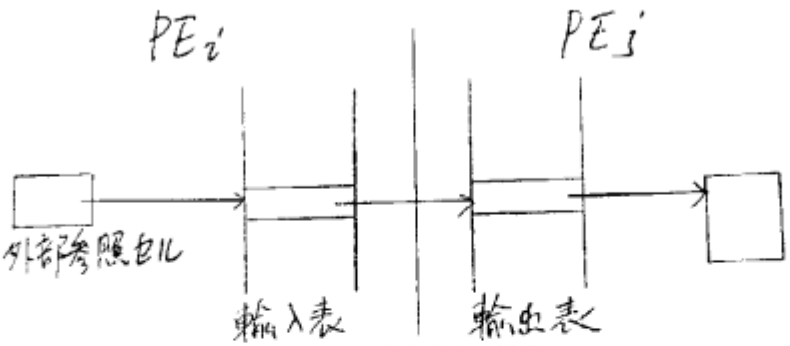


図 6: 外部参照の構造

```
foo :- true | bar(X,{1,2,3},a)@node(1).
```

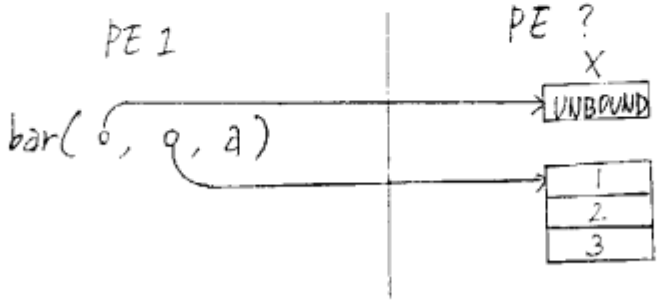


図 7: 外部参照の生成

4 プロセッサ間処理概要

4.1 プロセッサ間データ転送に関するポリシー

プロセッサ間処理方式に関して重要なのはデータ実体をプロセッサ間で移動させるタイミングである。

マルチ PSI において採用されたのはレイジーな移動方式であり、この方式ではデータ実体はガードユニフィケーションなどで実際に必要になった時に初めて転送される。

イーガーにデータ実体を送り付ける方式と比較して、データ転送のために処理遅延が生じる可能性はあっても、不必要なデータ転送が避けられるメリットの方が大きいであろう、という予測からこの方式を採用したのである。

4.2 外部参照

他プロセッサに実体のあるデータへの参照を外部参照と呼ぶ。外部参照は輸入表、輸出表と呼ばれる 2 種類のデータ構造を経由して間接的に参照されるが (図 6)、これは局所 GC(1 プロセッサ内でデータのアドレスが勝手に変更される処理) を可能とするためである。

また、これらの輸出表 / 輸入表という構造は同一データに対する外部参照を一つにまとめるためにも使用されている。

4.3 外部参照の生成

基本的には外部参照は @node プラグマによってゴールが他プロセッサに送出されるのに伴って生成される。

具体的には、投げられるゴールの引数が未具体化変数もしくは非アトミックデータである場合に、当該データが外部参照としてゴール送出先のプロセッサから参照されることになる。(図 7)

注: アトミックデータの場合にはデータ転送に要するコストが非常に小さいので、常にゴールの引数として実体が送られることになる。

注: 図 7のように、変数セル (X) やベクタ ({1,2,3}) はそのゴールのみからしか参照されていなくても、ゴール送出側プロセッサに作られるということに注意が必要。

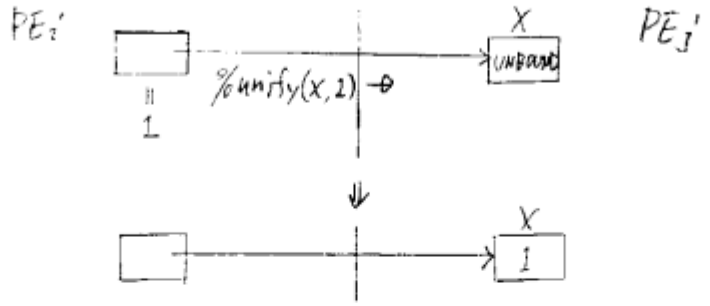


図 8: %unify メッセージの送出

4.4 データの転送

4.4.1 外部参照データの読み出し

他プロセッサからゴールを受け取った後の実行中、いずれかの時点で、ガードユニフィケーションなどにより外部参照データの実体が必要となった場合、データの読み出しが行われる。データの読み出しは、

- 外部参照先に向けて %read メッセージを送出。
  - — 外部参照はチェーンを形成し得るので、その場合、%read メッセージはチェーンに沿って複数のプロセッサを転送される。
- %read メッセージを受け取ったプロセッサでは %answer_value メッセージによってデータ実体のコピーを %read 送出元プロセッサへ送る。この際も、やはりレイジーな転送ポリシーに従って、1 レベル転送という方式でコピーが送られることになる。

  1 レベル転送: 送られるデータがベクタなどの非アトミックデータである場合、**そのトップレベルのみを実体として送り、ベクタの各要素に関しては外部参照に変換する**というデータ転送方式。
  ただし、ゴール送出時同様、ベクタなどのある要素がアトミックデータである場合には、転送コストが小さいため、その要素まで実体として送られることになる。

4.4.2 外部参照へのデータの書き込み

ボディユニフィケーションで外部参照と具体値とのユニフィケーションが発生した場合、その具体値のコピーが %unify メッセージによって外部参照先へと送られる。

注: ボディユニフィケーションを行ったプロセッサではデータのコピーを参照先へ送出するだけであり、自プロセッサ上の外部参照セルを当該データに具体化したりはしない。(図 8)

%unify 送出時にも %answer_value における処理同様、データの転送は 1 レベル転送方式によって行われる。

5 おわりに

効率的な KL1 プログラミングを行うために必要になると思われるマルチ PSI 上の処理系実装上の特徴などについて記した。

最後に改めて注意しておいてもらいたいのは**ここで説明した特徴はあくまでもマルチ PSI 上の現 KL1 処理系のみに当てはまることである**ということである。

これらの特徴は処理系、コンパイラの変更により任意に変更され得るものであり、性能向上のために利用するのは結構なことではあるが、**実行の成否をこれらの特徴に頼ったプログラミングはしてはならない。将来の処理系変更によって動かなくなるプログラミングはしないことが望まれる。**

参考文献

[1] T. Chikayama and Y. Kimura. Multiple Reference Management in Flat GHC. In *Proceedings of the Fourth International Conference on Logic Programming*, Vol. 2, pp.276–293, 1987.

[2] Y. Inamura, N. Ichiyoshi, K. Rokusawa and K. Nakajima. Optimization Techniques Using the MRB and Their Evaluation on the Multi-PSI/V2. In *Proceedings of the North American Conference on Logic Programming 1989*, pp.907–921, October, 1989.

涙なしの KL1 プログラミングのために

近山 隆
ICOT 第2研究室

概要

KL1 は非常に簡素な言語で，言語自体としては特定のプログラミング・メソドロジに沿った機能を取り込んでしまわず，自由なスタイルで書けるようにしている．この自由度のため逆に，プログラミングの方針が不適切だと非常にバグが取りにくいプログラムを書いてしまう可能性がある．一方，KL1 のプログラム開発環境としてどのようなものがふさわしいかはまだ研究途上であり，当面の環境である PIMOS は種々の実験的な道具をそろえてはいるものの，まだ開発環境として成熟したものとはいえない状況である．

本稿は，KL1 プログラム開発者のために，プログラム作成にあたってどのようなスタイルをとるとバグが出にくい，ないし取りやすいプログラムになるかのヒントを与えることを目的とする．また，バグが入ってしまった場合，その存在個所をつきとめるために，当面使える開発環境 (PIMOS) ではどのような手段があるかについても述べる．

1 予防医学

バグは後から取るべきものではない．最初から入れないに限る．デバッグの苦労を思えば，その前のプログラミングにかなりの労力をかけてでも，バグの入りにくいプログラミングをする価値がある．

1.1 バグを入れないプログラム構造

まずプログラミンの構造自体，バグが入りにくいようなものを選ぶべきである．

1.1.1 スタイルを決める

KL1 は非常に簡素な仕様の汎用並列プログラム言語である．たとえば，普通の手続き言語ベースでの並列プログラミングと比べてみると，ごく少数の機構で数多くの機能を実現していることがわかる．

- 手続き言語の条件実行 (if-then-else)，多岐分岐 (switch)，並列プロセスの同期，逐次実行などは，すべて単一の「ガード」という機構で実現される．
- 手続き言語の繰り返し，サブルーチン，コルーチン，並列実行などは，すべて単一の「ボディゴール呼出し」という機構で実現される．

こうした簡素な仕様は言語処理系の実現や言語自体の研究には有利で，また言語仕様自体の習得を容易にしている．言語仕様自体はあらかじめ特定のプログラミング・メソドロジを想定していないので，新たなメソドロジを自由に取り込んでいける奥行きの深さを持っているのも利点である．

与えられた問題をどう解くか，さらには，たとえアルゴリズムまで与えられていても，それをどうプログラムに表すかについて，プログラマに大きな自由度が残っている．たとえばプログラムをどのように構造化していくかについても，さまざまな方法が可能である．

このことは逆に言うと，KL1 の言語仕様は何も決めてくれない，ということになる．プログラムの質がプログラマの技量に大きく左右されやすくなっている．常にある程度以上の品質のプログラムを得ようと思えば，言語仕様自身が提供する機構を生のまま使うのではなく，あえて一定のプログラミング・スタイルを決め，その方針に沿ってプログラミングしていくのが有利になる場合が多い．スタイル (複数でも良い) を決めて書いてあると，他人の書いたプログラムでも読みやすくなる利点もある．

以下に，KL1 のプログラム構造として代表的なもののいくつかと，そうした構造化の方針を取

る場合にバグの混入を未然に防ぐための注意事項を述べる．

1.1.2 木構造

問題を部分問題に分割して子供ゴールに解かせ，その結果を統合して親に返す，という方式をプログラムの構成原理にする方針である．この方針に従うと，典型的な述語は以下のような形になる．

```
p(I, O) :-
  decompose(I, I1, ..., In),
  p1(I1, O1),
  ...,
  pn(In, On),
  compose(O1, ..., O).
```

ここで述語 decompose, compose の内容は，以下のようなものである．

decompose: 問題記述 I から，部分問題記述 I1, ..., In を作る．具体的には，たとえば以下のようなものである．

- Unification により構造体を部分構造体に分割する
- 問題を場合分けする（数値範囲なら部分範囲に分ける）

compose: 部分問題の解 O1, ..., On から，全体の解 O を作る．具体的には，たとえば以下のようなものである．

- 部分解を要素に持つような全体の解を作る（たとえば，部分解を要素とする構造体を作る）
- 部分解のうちの最良のものを選ぶ
- 数値的な部分解の和などをとる

この方針を取ると，プログラム全体はゴールの木構造になる．多くの探索問題は簡単にこの構造にできるだろう．

木構造を取る場合の留意点としては，以下のものがある．

入出力を分ける: どの引数が問題記述（入力）で，どれがそれに対する解（出力）であるかをはっきり分け，それを意識しながらプログラミングする．Prolog について喧伝されたような双方向性を持つプログラムは KL1 では記述できない．[1]

部分木は直接通信させない: 部分問題を解く計算を通して得られたデータを，別の部分問題を解くところでも活用したくなる場合は少なくない．このような場合も，部分木どうしが直接通信するようにプログラムすると，プログラムの全体の木構造が崩れて，構造が把握しにくくなりやすい．

プログラム全体は木構造のままで，部分木間でデータを共有したい場合は，共有データのためのサーバプロセスを置き，ここを経由して通信するようにすると，構造を把握しやすくなる．具体的には，以下のような構造になるだろう．[2]

```
p(I, O) :-
  decompose(I, I1, ..., In),
  merge({S1, ..., Sn}, S),
  shared_data_server(S),
  p1(I1, S1, O1),
  ...,
  pn(In, Sn, On),
  compose(O1, ..., On, O).
```

決してとるべきでない通信形態は「部分木のうち最初に解を見つけたものが他の部分木の計算をやめさせる」というような方法である．このような形態をとると複数の部分木がほぼ同時に解を見つけた場合に，いつでも正しく動作させるようにするのが難しくなる．解が見つかった，という事実は共有データである．解を見つけた部分木は共有データのサーバに連絡し，このサーバが他の部分木を終了させるようにするのが良い．一般に，競争条件は原則としてストリームのマージ機構に一元化すべきである．

[1] Prolog でも実際に双方向性が役立つことは少ない．

[2] これは後述するネットワーク構造に一歩足を踏み入れている．

1.1.3 ネットワーク構造

ゴールのネットワーク構造で問題を解く方針である．各子供ゴールは入力データになんらかの処理を施して，出力データを作る．この出力は他の子供ゴールの入力として与えられる場合もあれば，最終的な出力になることもある．プログラム全体としてはゴールをノード，共有変数を方向付きのアークとするネットワーク構造になる．

データを構造体にして，トップダウンに徐々に具体化するようにすれば，出力が完成する前に次段階が動き始めることができる．データをリスト構造にしたのが，Shapiro-竹内の「オブジェクト指向プログラミング」で，リストの car だけ決めて cdr が決まる前に次段階を動けるようにするのが「ストリーム通信」である．

ネットワークの各ノードの役割は，以下のように分類できる．

フィルタ: 入力データを加工し，結果を次段階の加工を行なうゴールに渡す．述語の仕様は概略，

$$p(\text{入力}, \text{出力})$$

のようになる．

ディストリビュータ: 単一の入力を噛み砕いて複数の出力を作る．これらの出力は複数のゴールに与えられることになる．述語の仕様は概略，

$$p(\text{入力}, \text{出力}_1, ..., \text{出力}_n)$$

のようになる．

マージャ: 複数の入力を統合して，単一の出力を作る．述語の仕様は概略，

$$p(\text{入力}_1, ..., \text{入力}_n, \text{出力})$$

のようになる．組込みのマージャはこの一種である．加減乗除などの算術演算組込述語もマージャの一種と考えられる．

このような子供ゴールを組み合わせて，ゴールのネットワークを作り，全体の問題を解く．典型的な構造には以下のようなものがある．

パイプライニング: 複数のフィルタを通すことによって，問題を段階的に解に変換していく構造．

```
p(X0, Xn) :-
  p1(X0, X1),
  ...,
  pn(Xn-1, Xn).
```

このような述語全体をブラックボックスとして考えると，一入力一出力のフィルタであるから，パイプライン構造はネストして一本の長いパイプラインになる．

ディスパッチング: 問題をディストリビュータで分割し複数のフィルタに入力し，その出力をマージャで統合して全体の出力を作る構造．

木構造はグラフの一種であるから，木構造プログラミングもプロセス指向プログラミングの一種である．木の各ノードはディスパッチング構造をしている．

```
p(In, Out) :-
  decompose(I, I1, ..., In),
  p1(I1, O1),
  ...,
  pn(In, On),
  compose(O1, ..., On, O).
```

この述語 decompose はディストリビュータ，compose はマージャで，部分問題を解くゴールはフィルタである．ディスパッチング構造がネストしたものが木構造である．

ネットワーク構造のプログラミングをする場合の留意点としては，以下のようなものがある．

入出力を分ける: これをはっきりさせないと，各ノードの役割やその結合形態を云々すること自体難しくなる．

各ノードの役割は単純に: KL1 では多入力・多出力のマージャとディストリビュータを兼ねるゴールを作ることも簡単にできる．しかし，このようなゴールを多用するとプログラムの全体構造が見えにくくなる．できる限りマージャ，フィルタ，ディストリビュータといった，単純な構成要素から考えた方が良い．

ネットワーク構造が大きくなってくると，当然モジュール化が必要になる．サブネットワー

クをモジュールとして考え，ブラックボックスとしてひとつのネットワークノードと考えるのが便利である．このときも各サブネットワークが上述のみっつの典型のどれかの役割を果たすようにすると，プログラム構造がわかりやすくなる．

実際には完全にこれだけで済ませることできないことは多い．だが，その場合でもプログラムの全体構造を決める主たる入出力は単一にし，それ以外は副次的なものと考えられる構造にしておくと整理が付けやすい．前述した木構造の場合の共有データサーバを使う方法は，この考え方に基づくものである．

ループは避ける: ストリーム通信を使えばループ構造の正しいプログラムも書けるが，ループ構造はデッドロックの温床になるので乱用すべきではない．ループがなければデッドロックはあり得ない．

直接的なループ構造を避けるためには，やはりサーバ方式が有効である．パイプライン構造の下流から結果を直接上流にフィードバックするのではなく，データベースであるサーバプロセスに送ることとし，上流からはこれにアクセスして情報を得るようにすると，基本的なプロセス構造がすっきりして把握しやすくなることが少なくない．

1.1.4 サーバ・クライアント構造

特定の機能を持ったプロセスを作り，その機能が必要な時にはストリーム通信でアクセスするようなプログラム構造である．これはプログラム全体の構造というより，上述した例にもあるように，プログラム構造が複雑になり過ぎるのを防ぐための補助手段として有効である．また，プログラムの部品化のためにも有効なモジュール化手法である．

サーバプロセスの典型的な述語構造は以下のようなものになる．

```
p([message(I, O)|S], State) :-
    compute(I, State, O, New),
    p(S, NewS).
```

1.2 バグを入れないプログラム記述

具体的なプログラム記述にあたっても，バグが入りにくいような記述をするように留意すべきである．

KL1 プログラムに多く見られるバグには，以下のようなものがある．

- 変数のつづり間違いによる永久中断．変数のつづり違いのために，渡された引数をそのまま子供ゴールに渡すべきところを新しい変数を渡してしまうと，子供ゴールがこれを具体化してもそれが元々の変数の具体化にならず，その具体化を待つゴールの実行が中断したまま再開されない．
- ストリームの閉じ忘れによる永久中断．プロセス終了時にそのプロセスが送信していたストリームを閉じる操作を書き忘れ，そのためストリームからのメッセージまたはストリームの閉鎖を待つプロセスの実行が中断したまま再開されない．特に，両者の間にストリームマージャが入っていると，マージインするストリームの内どれを閉じ忘れているのかを知るのは難しくなる．これはプログラムが正しい結果を計算し終えたのに実行が終らない，という現象になって現れる．

こうしたバグはプログラムの改訂時にも混入しやすい．プロセスの状態変数(再帰呼出しの引数)を増やす時にはすべての節の引数を増やさなければならない．これには手間がかかるだけでなく，つづり間違いやストリーム閉じ忘れを招きやすい．

こうしたバグが生じるのは KL1 が使い方を限定しないニュートラルな言語であるため，引数の特定の使い方パターンに対しての自動的な処理や誤り検出機構を持ちにくいからである．[3] ということは逆に，引数の使い方を明示してやれば，こうした誤りを未然に防ぐことができる．

KL1 の暗黙引数マクロ (PIMOS 2.5 版マニュアル「2.1.6 マクロ記法」のうち 25–30 ページ) はこの目的のためのものである．暗黙引数マクロを用いれば，

[3] 大域的なデータフロー解析などをすれば検出できる誤りをかなり増やせるはずではあるが，まだそうしたユーティリティはない．

- 親ゴールからすべての子ゴールに渡すだけの引数や，パイプライン構造プログラムでフィルタをつなぐだけのための引数は，それぞれ shared 型，oldnew 型の暗黙引数として宣言しておけば，いちいち記述せずに済みつづりを誤る可能性もない．
- ストリーム引数については，stream 型暗黙引数と宣言しておけばプロセス終了時（ボディゴールのない節を選んだ時）に自動的に閉じられるので，閉じ忘れの可能性はない．

となることから，上述の典型的なバグは最初から入れずに済む．また，

- 状態変数の追加時には，実際にその状態変数を参照する部分と暗黙引数宣言部だけを訂正すれば良く，単に受け渡しているだけの中間部分には手を入れなくて良い．

ので，改訂時にバグを混入する恐れも小さい．

暗黙引数マクロは最初に記述する時にはやや面倒ではあろう．しかし，次第に仕様を拡張し複雑になっていく運命にあるプログラムの記述にあたっては，できるだけ早い時期から利用するのが有利である．

なお，現在開発中の言語 AYA も同様の目的の言語で，仕様上の類似点も多く，暗黙引数マクロで記述しておけば AYA への移行も簡単になる．

1.3 バグを取りやすいプログラミング技法

どんなに注意してみても，プログラムにバグを入れないことは難しい．もっともとりにくいバグはアルゴリズムのバグで，プログラミングのレベルでどう工夫してみてもこれを防ぐことはできない．

入れてしまったバグはなんとかして取らなくてはならない． PIMOS のリスナで追いかけるなどの方法もあるが，リスナのような汎用のデバッグ支援機能には KL1 プログラムレベルでの知識しかないので，どこが勘所かがわかっておらず，ある程度以上の規模のプログラムになると，本当に調べたい部分を調べられるようにするまでの操作に結構手間がかかる．

いっそ，自分の書くプログラムにはバグは必ず入る，それを取ることになるのだと観念し，最初からプログラム中にバグを取るための機構を仕掛けておくのが，往々にしてたいへん役に立つ．そのような機構を最初から全部は入れないまでも，入れようと思えばいつでも入れられる枠組を考えておくのは重要である．

数多くのプロセスが協調動作するようなプログラムの場合，どのプロセスがどんな状態にあるのかを知るのはデバッグの大きな助けになる．リスナでこれを知ろうと思うと，すべてのプロセスをトレースしておき，その最後のリダクション状況をすべて調べる必要があり，操作は繁雑で時間もかかる．

このような機能を提供するためには，全プロセスから状態報告を集めてくるような仕掛けを最初からプログラム中に入れておけば良い．プログラマは処理系よりも高いレベルでプログラム構造を把握できるので，必要になる情報が何かを取捨選択できる．その有力な手段として，あらかじめ全プロセスを串刺しにするようなストリームを張っておく方法がある．

各プロセスを記述する述語にはふたつの引数を追加し，たとえば以下のような節を追加する．

```
p(..., [state(Ans)|Q0], Q) :-
  Ans = [状態|AnsTail],
  Q = [state(AnsTail)|Q1],
  p(..., Q0, Q1).
```

この引数は子供ゴールを生成する時にはパイプライン的に渡し，プロセスが終了する時には両引数をユニファイする．

```
p(..., Q0, Q) :- 終了条件 |
  Q0 = Q.
p(..., Q0, Q) :- 分岐条件 |
  p1(..., Q0, Q1),
  ...,
  pn(..., Qn-1, Qn),
  p(..., Qn, Q).
```

このようにストリームを実現するための引数をいつも記述するのは繁雑なので，oldnew 型の暗黙

引数にしておくのが良いだろう.

```
:- implicit
   他の暗黙引数, ...,
   status_query:oldnew.
```

こうすれば上述の子供ゴールの生成時の受渡し, プロセス終了時のユニファイは自動的に行なわれる.

このような準備をしておけば, トップレベルの述語の対応する引数に state/1 メッセージを流すことによって, 全プロセスの状態を把握することができる. 具体的には, 追加した引数の最初のものを, [state(L)|Q0] のようなタームに, もう一方を [state(□)|Q1] に具体化してやれば, L に各プロセスの状態のリストが得られる. 次に状態を知りたい時には, これらのタームの中の Q0, Q1 を用いれば良い.

この手法はデバッグ以外にも, 全プロセスの完了を知るためなどにも有効である. 対応する引数の最初の方を □ に具体化して, もうひとつの引数が □ になるのを待てば良い.

2 治療医学

どう予防してもバグは入る. 入ってしまったバグをどうやって取り除くかとなると, 残念ながらあまりシステマティックな方法はない.[4] そこで, 現在 PIMOS が提供している機能の中でバグを取るのに役立ちそうなものをいくつか紹介する.

2.1 デッドロック

デッドロックは一番やっかいなバグである. このバグを入れないためには, 前述の通りプログラム構造をできるだけ簡潔に理解しやすいものにしておくのが第一である. 不幸にしてデッドロックが生じてしまった場合には, 以下の方法で問題を解析する.

2.1.1 変数名つづり誤りのチェック

変数名のつづり誤りがあると, 具体化すべき変数を具体化しそこねることになり, その具体化を待っているプロセスが永久に先に進めない状態になる. これは本来のデッドロック (複数のプロセスが互いに相手が進むのを待っている) ではないが, プログラムが全然進まなくなるという現象面では似ている.

つづり誤りを見つけるには, PIMOS の提供するユーティリティ (PIMOS 2.5 版マニュアル「12.9 変数チェッカ」193–195 ページ) を使うのが便利である.

2.1.2 ガーベジコレクションによる検出

Multi-PSI などの KL1 処理系は永久待ち合わせ (デッドロックも含む) を検出する機能を提供している. これはガーベジコレクション時 (実行中のインクリメンタルなガーベジコレクションも含む) にどこからも起動されないゴール (普通複数ある) を発見し, その依存関係を解析して, 根本原因になっているゴールを報告するもので, デッドロック検出の強力な武器になりうる.

リスナからの実行中にデッドロックが疑われる状態に入ったら, control-C によって実行を中断し, ガーベジコレクタを起動すれば, 永久中断ゴールを探させることができる (PIMOS 2.5 版マニュアル「13.5 永久中断ゴールの検出」213–219 ページ).

この際, リスナのトップレベルで出現した変数の値を覚えておかないモードにしておく方が良い (forget コマンド; PIMOS 2.5 版マニュアル「13.6.5 変数ブール操作コマンド」229 ページ). 検出機構は KL1 処理系のものであるから, リスナも応用プログラムも同等に扱う. リスナが覚えている変数は後でリスナから具体化するのかも知れないので, それを (あるいは, その要素になっている変数を) 待っているゴールは永久中断ゴールとは判定できないのである.

なお, この検出機能は万能ではなく, プロセサ間に渡る参照構造には対処していない (現在のところプロセサ単位のガーベジコレクションしか提供していないため).

2.2 無限ループ

外見上デッドロックと良く似ていて同じようにたちの悪いバグのが, 無限ループによる暴走である. デッドロックとの区別は, ときどき消費資源量を見ると増えている (たとえばリスナで

[4] KL1 プログラムの論理としての側面に注目した方式の提案はあるのだが, 耐えられるオーバヘッドの範囲で実用的なツールを作ることは現在のところ難しい.

Control-C で実行を中断して，r コマンドで調べる; PIMOS 2.5 版マニュアル「13.2.7 実行の中止」の 203–204 ページ) ことからわかるが，ループが何も外部に出力しないと，どこでループしているのかはまったくわからなくなる．プログラムが小さいうちはすべてをトレースすることもできるが，ある程度大きいシステムになると実際上不可能である．

このような場合には ParaGraph を用いると便利である． ParaGraph は本来はデバッグのためのツールではなく，負荷分散戦略設計などの性能チューニングを支援するプロファイリング・ツールなのだが，暴走の解析にも非常に有効なのである．

具体的には以下のようにする．

1. 暴走するプログラムをプロファイルを取りながら実行する (profile コマンド; PIMOS 2.5 版マニュアル「13.6.9 ParaGraph 関連コマンド」240–242 ページ).
2. 暴走が疑われる状態になったら，Control-C で実行を中断して a (abort) コマンドで実行を放棄してしまう (「13.2.7 実行の中止」の 206 ページ). こうすると実行を放棄した時点までの実行プロファイルが得られる．
3. このプロファイル情報を paragraph コマンド (リスナでの操作は 242 ページ; 詳しくは「第 15 章 ParaGraph」304–318 ページを参照) を用いて表示する．どの述語がどのプロセサでどのぐらい走ったのかは，what × where 表示でわかる．

十分長い時間暴走させれば暴走部分の実行回数が突出するので，暴走した述語が何か，どのプロセサで走っていたのかわかり，暴走原因を見つける重要なヒントになることが多い．

3 おわりに

冒頭に書いたことの繰り返しになるが，バグは最初から入れないのが一番である．すでにバグが入りやすい書き方をしてしまったプログラムは，バグが入りにくいように書き直すに限る．とりあえず動いているからといって使い続けると，必ず仕様拡張などの手直しが必要になり，そのたびに新たなバグを入れてデバッグに苦労することになる．

プログラミングの労力の大部分は，タイプインすることでもなければ，低レベルの表現をどうするか考えることでもない．高いレベルでのアルゴリズム設計やプログラム構造の設計にある．そうでないような労力配分で作ったプログラムは，たいがいは性能が低くバグが入りやすいプログラムか，さもなければ最初から書かなくても済んだようなプログラムだろう． 高いレベルの設計ができてしまっていれば (あるいは，一回作ることによってより良い設計のアイディアが固まっていれば)，プログラムを全面的に書き直すのには最初に書くときに比べてずっと小さい手間で済む．全面的に見直して思い切って書き直す，これが涙なしのプログラミングの秘訣だろうと私は考える．

最後に，本稿の基礎になったアイディアを提供してくれた，あるいは本稿で紹介した機能を設計実装してくれた諸氏に感謝したい．プログラムの設計あるいは再設計・書き直しの際に本稿で紹介した指針が少しでも役に立つことを願う．

スケーラブルな並列プログラムのすすめ

市吉 伸行
ICOT 第7研究室

1 「スケーラブル」って何?

「スケーラブル(scalable)」という言葉を聞いてことがあるだろうか? これは「スケールアップできる」、すなわち「大規模化できる」、という意味で並列処理の世界でよく使われる用語である。[1]

並列計算機アーキテクチャ屋が「スケーラブルなアーキテクチャ」と言うと、「そのままのアーキテクチャでプロセッサ台数を幾らでも増やせる」ということである。例えば、Symmetry のような共有メモリ型アーキテクチャは共有バスがボトルネックになるからプロセッサ台数をやたらに増やす訳には行かないのでスケーラブルでない。一方、マルチ PSI や PIM のような分散メモリ型のアーキテクチャはそのようなアクセスボトルネックがないのでスケーラブルである。

せっかく、マルチ PSI や PIM のようなスケーラブルな並列マシンがあるのだから、『スケーラブルな』並列プログラムを走らせたい。『スケーラブルな並列プログラムのすすめ』である。

2 スケーラブルな並列プログラム

2.1 第1の定義

まず「スケーラブルな並列プログラム」といっても曖昧なので定義を決めよう。まず反例から考えてみよう。例えば、「append プログラムはスケーラブルな並列プログラムか」と聞かれたら誰もがノーと答えるであろう。プロセッサが何台あっても append プログラムは速くならない。では、

```
X = (A * B) + (C * D)
```

という代入文の右辺の2つの掛け算を並列に行なうプログラム(??)はどうであろう? これもスケーラブルとは言いたくないであろう。何故ならプロセッサが3台以上あっても速くならないからである。つまり、スケーラブルでないプログラムとは、プロセッサ台数を増やして行ってもある台数から先は速度向上しないようなプログラムのことと言えそうである(言うことにしよう)。そこで、これを裏返して、第1の定義:

> **定義1** スケーラブルな並列プログラムとは、プロセッサ台数を増やして行くと速度がだんだん速くなるようなプログラム(マシンがスケールアップするにつれて、性能もスケールアップするようなプログラム)である。いやしくも、プロセッサ 256 台構成の PIM/m や 512 台構成の PIM/p の上でプログラムを走らせるなら、性能も 100 倍、200 倍... とスケールアップさせたいのが人情で、スケーラブルなプログラムならそれが可能な訳である。

2.2 Amdahl の壁

ではどのようなプログラムが第1の定義にいう処のスケーラブルなプログラムであろうか。一般にどのようなプログラムも逐次に処理をしなければならない部分を含んでいる。あるプログラムにおいて、そのような逐次部分が s で並列化できる部分が p だったとしよう、すると後者を幾ら高速化しても全体の実行時間は s だけかかる。プロセッ

[1] AI では、狭い範囲の例題だけをきれいにこなすプログラムを「そのままのメカニズムで本格システムに拡張することができない」という意味で「スケーラブルでない」と批判することがある。このようなプログラムは『トイプログラム』と呼ばれる。

サ1台での実行時間を T_1、プロセッサ p 台での実行時間を T_p とすると、速度向上率 S は

$$S = \frac{T_1}{T_p} < \frac{s+p}{s}$$

となる。これは、どんなプログラムも、並列化による速度向上がある一定値を超えないことを意味している。例えば、s が全体の 10% あると、速度向上率の限界は 10 倍である。これが有名な Amdahl の法則である。

したがって、スケーラブルな並列プログラムは存在しない!?

2.3 第2の定義

スケーラブルなプログラムは存在しないと言われても、それは抽象的な話で、現実には、例えばマルチ PSI なら最高 50 倍程度に速度向上するプログラムは存在する訳で、PIM に 256 台や 512 台のプロセッサがあれば、それに見合った速度向上するプログラムだってありそうだ、という疑問が生じるであろう。そこで、もう一度、考え直してみよう。

ある問題を逐次実行で解くと W 時間かかるとする。これを最小粒度 u で並列実行できたとすると、速度向上の上限は W/u である。粒度 u はある値(例えば、一つの浮動小数点演算の実行時間)を下回れないので速度向上には限界がある。粒度を小さくすることに限界があるのに対し[2]、全体の仕事量 W は問題を変えれば大きくできる。すなわち、問題を固定すると速度向上には天井があるが、問題を大きくすれば速度向上の天井も高くなる。そこで、スケーラブルなプログラムの第2の定義:

> **定義2** スケーラブルな並列プログラムとは、プロセッサ数を増やし、問題サイズも大きくしていくと、幾らでも大きな速度向上の得られるようなプログラムである。

速度向上のために問題を大きくするのは本末転倒ではないかという批判があり得よう。「私の対象とする問題はこれこれのサイズが普通で、それより大きな問題は考えるのは現実的でない。」これはもっともな批判で、構築可能なサイズの並列マシンのプロセッサ数に見合った並列度を含むような問題が実際的サイズかどうかが、あるアプリケーションが大規模並列マシンを有効活用できるかどうかの正に分かれ目と言えるであろう。効率よく実行できる最小粒度をできるだけ小さくしてあげるのが、アーキテクチャ屋と処理系屋の仕事であると同時に、より並列度の高い並列プログラムを書くようにプログラマは努力すべきである。

3 スケーラブルな並列性を引き出さないには

では、スケーラブルな並列性を引き出すにはどのようなことに注意すればいいであろうか。ここでは、反語的にスケーラブルな並列性が出ないようにするにはどうすればよいか、書いてみよう。

スケーラブルな並列性を引き出さないには

- 大きな処理を逐次的に行なう (Amdahl の原理)

  例えば、処理の 20 % を占める部分を逐次実行すれば、速度向上が 5 倍以上になる恐れはない。これは非常に強力な原理であり、幾つもの応用テクニックがある。

  1. 小規模問題

     前述のように、小規模な問題を扱っていれば、どんなプログラムを書いても大きな並列度が出る心配はない。

  2. 少数プロセス

     全体の処理を両手で数えられる数の並行プロセスに分割し、それぞれを逐次実行する。

     これはやや邪道なので、もう少し目立たない方法もある。

[2]少なくともプログラマにとっては。

3. 星型管理

全体処理を多数のプロセスに分割するが、それらが1つの共通資源(例えば、共有データベース)をアクセスするようにする。あるいは、多数のプロセスの出力データを単純にマージして1つのプロセスが処理するようにする。

しかし、プログラムの静的構造を調べられると木構造にしなかったことが判明してしまう。静的解析では見つけ難いもっと高等なテクニックもある。

4. 意外に大きな逐次部分

例えば、KL1 の典型的な qsort プログラムでは、親 qsort が 2 つの子 qsort を生み、子のそれぞれが 2 つずつの孫 qsort を生み、というように木状にプロセスが増えていくので、高い並列度が出るように見える。長さ n のリストのソートに全体で平均 $n \log n$ の比較が必要であるのに対し、最初の partition に n 回の比較が必要である。したがって、逐次部分が全体の $\log n$ 分の 1 である。これなら、例えば長さ 100 万のリストでも並列度は 20 程度に抑えることができる。[3]

- 見込み計算

後で無駄になるような処理を沢山行なうことで、プロセッサ稼働率は高いが、速度向上率を低く抑えることができることがある。例えば、枝刈りのできる木探索アルゴリズムの単純な並列化など。ただし、場合によっては、並列化によって線形以上の速度向上が得られてしまうので、注意が必要である。

- 通信オーバヘッド

あるプロセスがよくアクセスするデータが局所メモリになければ、ネットワーク経由でデータが運ばれ、通信遅延やメッセージ処理コストによって性能を抑えることができる。

- その他並列化オーバヘッド一般

頻繁なプロセスチェンジによってスループットを落とすことができる。KL1 の中断はプロセスチェンジを引き起こす。ガード部中断の他、ボディ組込み述語の中断もあり、活用の機会は多い。

負のテクニックばかり挙げたが、次にそれに違反する並列プログラムの例を示そう。

4 事例: 15 パズル盤面の数え上げ

15 パズル、牛パズル、ルービックキューブのようなパズルにおいて、ある与えられた配置からプリミティブな操作(コマを 1 マス移動させる、あるいは、可動部を 90 度回転させる)を何回か繰り返して到達できる配置の集合を求める問題を考える。そのような集合が決定できれば、初期配置から目標配置へ至る最短経路(最小回数の操作手順)を決定する問題は、単純な応用として解くことができる。初期配置から始まる状態空間が木状に近い場合はそのような探索は逐次深化探索(iterative deepening search)によって効率良く探索できるが、異なる枝からの合流性の強い場合、特に n が増えるに従って多項式オーダ程度でしか S_n が大きくならないような場合は、状態集合 S_n を数え挙げて重複を省くやり方(数え挙げ法)の方が一般に計算量が少なくて済む。このような問題を並列に解くことを考えてみる。

4.1 基本アルゴリズム

15 パズル、牛パズル、ルービックキューブのようなパズルを、以下のように定式化する。パズルの可能な配置の空間(状態空間) $\mathcal{S}$、配置を配置に変換する操作の集合 $\mathcal{F}$ が与えられているとする。操作 $f \in \mathcal{F}$ は全ての状態に適用できるとは限らないので、一般に、状態 $s \in \mathcal{S}$ と操作 $f \in \mathcal{F}$ に対し、s に f を適用できてその結果が t の時に $f(s) = \{t\}$、s に f を適用できない時に $f(s) = \{\}$ [4]と定義する。また2項関係 R を、$s, t \in \mathcal{S}$ について、$f(s) = \{t\}$ なる $f \in \mathcal{F}$ が存在するとき、またその時のみ、sRt (「s から t に1手で移れる」)と定義する。

[3] しかし、このテクニックとても、パフォーマンスメータや ParaGraph を使うと発見されてしまう可能性があるので、注意が必要である。

[4] 空集合を {} で表すものとする。

S の中の与えられた要素 g^* から「n 手以内で移れる要素の集合」G_n は次のように定義される。

$$G_{-1} = \{\},\ G_0 = \{g^*\},\ G_{n+1} = G_n \cup G_n R$$

(ただし、$XR = \{x \in S \mid \exists y \in X \text{ s.t. } yRx\}$.)

「x からちょうど n 手で移れる要素の集合」S_n は次のように定義される。

$$S_n = G_n \backslash G_{n-1}\ (n \geq 0)^5$$

上記、15パズル、牛パズル、ルービックキューブのようなパズルは、操作が可逆である。すなわち、sRt ならば tRs が成り立つ(関係 R が対称である)。このようなパズルを対称パズルと呼ぶことする。対称パズルは次の命題の性質を持つので、それまでに生成された全ての状態を保持しておかなくてはならない場合と比較して、状態数え上げの計算量が小さくて済むという特徴がある。

命題 1

$$S_{n+1} = S_n R \backslash S_n \backslash S_{n-1}\ (n \geq 0)$$

(証明) まず、

$$\begin{aligned}
&(\text{左辺}) = G_{n+1} \backslash G_n \\
&= (G_n \cup G_n R) \backslash G_n = G_n R \backslash G_n \\
&= (S_n \cup G_{n-1}) R \backslash G_n = (S_n R \backslash G_n) \cup (G_{n-1} R \backslash G_n) \\
&= S_n R \backslash G_n \cup \{\} = S_n R \backslash G_n
\end{aligned}$$

である。また、

$$(\text{右辺}) = S_n R \backslash (S_n \cup S_{n-1})$$

だから、(左辺) = (右辺) を証明するためには、$S_n R \cap G_n = S_n R \cap (S_n \cup S_{n-1})$ を示せばよい。$G_n = (S_n \sqcup S_{n-1} \sqcup G_{n-2}$ (排他和) だから、$S_n R \cap G_n = (S_n R \cap (S_n \cup S_{n-1})) \sqcup (S_n R \cap G_{n-2})$ である。したがって、$S_n R \cap G_{n-2} = \{\}$ を示せばよい。$x \in S_n R$ とする。定義から、ある $y \in S_n$ があって、yRx。R は対称だから、xRy。もし、$x \in G_{n-2}$ とすると、$y \in G_{n-2} R \subseteq G_{n-1}$ となり、これは $S_n \cup G_{n-1} = \{\}$ に矛盾。 ∎

この命題を用いると、S_n の系列を生成するには、最新の2つの集合から新たな集合を作ることを繰り返せばよい。手続き言語風に書いたアルゴリズムを図 1に示す。

4.2 並列化

この基本アルゴリズムにおける並列処理の候補としては、繰り返し文の並列化と個々の操作の並列化があり得よう。それらについて検討してみる。

- 繰り返し文の並列化

  (A) 3-9 行目の for 文
  異なる k について for 文の中を並列に実行する。

  (B) 5-8 行目の forall 文
  S(k) の要素それぞれについて for 文の中を並列に実行する。

  (C) 6-8 行目の forall 文
  xR の要素それぞれについて for 文の中を並列に実行する。

- 個々の操作の並列化

[5]中置記号 "\" は集合差を表わす。すなわち、$X \backslash Y = \{x | x \in X, x \notin Y\}$ である。"\" は "−" と同様、左結合的とする。すなわち、$X \backslash Y \backslash Z$ は $(X \backslash Y) \backslash Z$

```
1   S(-1) = {}
2   S(0) = {g*}
3   for k = 0 to n-1 do
4       S(k+1) = {}
5       forall x in S(k) do
6           forall y in {x} R do
7               if ( (not y in S(k)) and (not y in S(k-1)) ) then
8                   Add y to S(k+1)
9       (delete S(k-1))
10  return S(n)
```

図 1: S_n を生成する基本アルゴリズム

(D) S(k)、S(k-1) に関する要素チェック、S(k+1) への要素追加、それぞれの操作の並列処理。

スケーラビリティの観点からこれらの並列化を検討してみよう。

(A) は、S(k) 間に依存関係があり、S(k+1) へ要素を追加し始める前に、S(k-1)、S(k) が確定していなければならないので、ほとんど逐次的処理となる。

(B) は、S(k) のサイズ (要素数) に比例した並列度がある。S(k+1) への要素追加の順は任意なので並列実行可能。ただし、このままでは S(k+1) への追加は逐次化される。

(C) は $\{x\}R$ のサイズ (要素数) に比例した並列度があるが、これは状態空間によって決まる定数である。例えば、15 パズルでは高だか 4 である。n を増やすことで問題サイズを大きくしても、並列度が上がらないので、スケーラブルでない。

(D) の並列度は、集合をどのように表現して、要素チェックと要素追加をどのような逐次／並列アルゴリズムで行なうかに依存する。2 つの状態の同一性のみを判定できる場合は、要素チェックと要素追加には $O(N)$ の手間がかかる (N は集合の要素数)。大小が比較できれば、集合を 2 分木等で表現して、要素チェックと要素追加を $O(\log N)$ で行なえる。さらに、ハッシュ表を用いることができれば、逐次操作の平均処理時間オーダは $O(1)$ であり、(C) の場合と速度向上は一定数を越えない。

15 パズルなどでは状態を一定桁数の整数にエンコーディングでき、それに対してハッシュ関数を定義するのは難しくない。ここでもハッシュ表表現が可能だと仮定しよう。したがって、(D) による速度向上はスケーラブルでない。

まとめると、スケーラブルな速度向上が得られる可能性があるのは (B) のみということになる。

(B) において集合要素数に比例する並列度を実現するためには、集合からの要素取り出しと集合への要素挿入のそれぞれが、多くの要素について同時並列的に処理できる必要がある。

そのためには、集合に属する多数の要素を分散したプロセッサに付随するメモリに分散して配置して、各プロセッサが自分に割当てられた部分集合を処理することを可能にすればよい。

例えば、ハッシュ表サイズが N でプロセッサが p 台 (p は N を割り切るとする) の時、プロセッサ j ($j = 0, 1, \ldots, p-1$) が、ハッシュ表エントリ Nj/p から $N(j+1)/p - 1$ まで受け持つようにする。集合要素チェック、要素追加は、チェックまたは追加すべき状態のハッシュ値を計算し、ハッシュ値を受け持つプロセッサにその要素を送り出すようにすればよい。

4.3 実験

分散ハッシュ表による集合数え挙げアルゴリズムを KL1 でコーディングし、15 パズルを例題として、マルチ PSI 上で実行時間を実測してみた。結果の要素を 1 つのプロセッサに集めるとボトルネックになるので、プログラムは要素数のみを結果として返すような仕様になっている。

結果の一部を表 1 に示す。プロセッサ台数の 1/2 程度の台数効果が得られている。

表 1: 15 パズル実行時間 (単位秒)

プロセッサ数	1	4	16	64
12	29.3	10.4	3.1	2.7
13	46.4	17.9	5.2	3.3
14	81.2	33.2	9.6	4.5
15	152.5†	63.8	17.5	6.7
13−12	17.1 (1.0)	7.5 (2.3)	2.1 (8.1)	0.6 (28.5)
14−13	34.8 (1.0)	15.3 (2.3)	4.4 (7.9)	1.2 (29.0)
15−14	71.3 (1.0)	30.6 (2.3)	7.9 (9.0)	2.2 (32.4)

- かっこ内はプロセッサ 1 台と比べた速度向上
- †は推定値 (実行時間から推定 GC 時間を引いて補整)

表 2: S_n と G_n の要素数

n	$\sharp S_n$	$\sharp G_n$	n	$\sharp S_n$	$\sharp G_n$	n	$\sharp S_n$	$\sharp G_n$
0	1	1	10	1,948	3,754	20	1,637,383	3,418,020
1	2	3	11	3,938	7,692	21	3,098,270	6,516,290
2	4	7	12	7,808	15,500	22	5,802,411	12,318,701
3	10	17	13	15,544	31,044			
4	24	41	14	30,821	61,865			
5	54	95	15	60,842	122,707			
6	107	202	16	119,000	241,707			
7	212	414	17	231,844	473,551			
8	446	860	18	447,342	920,893			
9	946	1,806	19	859,744	1,780,637			

なお、15 パズルの S_n と G_n の要素数を表 2挙げておく。

協調型論理設計エキスパートシステムco-LODEXの試作

澤田秀穂，箕田依子，滝沢ユカ，丸山文宏
富士通株式会社

1．はじめに

我々は、ハードウェアの機能レベルの仕様から、回路規模と遅延時間に関する制約条件を満たす回路を生成する論理設計支援システムco-LODEX（co-operative logic design expert system）を開発している。第五世代プロジェクト中期には、与えられた制約条件に対して設計結果を評価し、違反が検出された場合、制約条件を満たすよう設計のやり直し（再設計）を行なう方式の研究において、論理式で表現する制約条件違反情報（NJ:Nogood Justification）を考案し、これを利用して再設計を実行するシステムを試作した［1］。後期では、この方式を並列協調方式を用いて、並列処理システムに拡張している［2］、［3］。

本稿では、まず、co-LODEXの概要と設計の流れを述べ、並列処理システム上に拡張するための並列協調方式について説明する。次に、Multi-PSI上にKL1を用いて部分試作した、試作システムについて述べ、試作システムを用いた負荷分散方式の検討のための実験とその結果について報告する。

2．co-LODEXの概要

co-LODEXにおける設計の流れを図１に示す。エージェントと呼ぶ自律的に動作する実行単位を設けて、部分回路を並列に、全体に対する制約条件を満足するように詳細化することが特徴である。

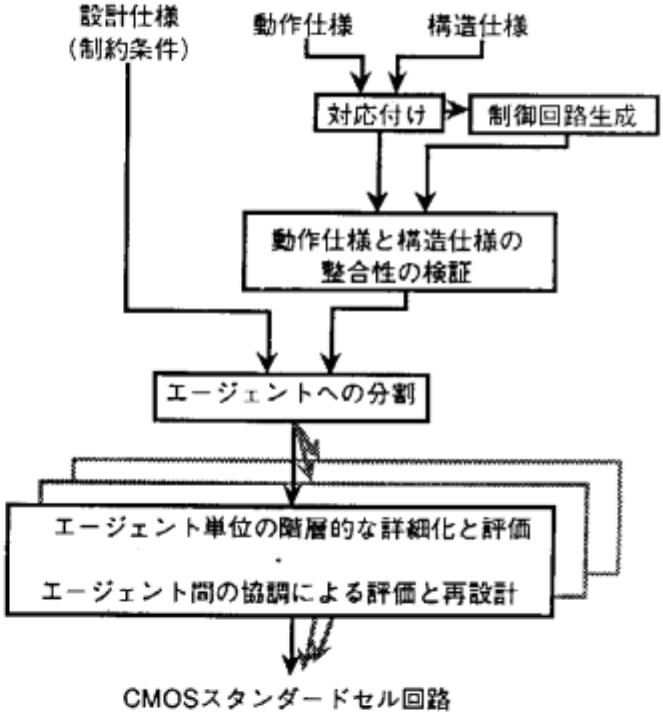


図１　設計の流れ

入力は、機能レベルの動作仕様と、レジスタ、演算器などの機能ブロックと機能ブロック間のデータの流れを示すブロック図、生成した回路が満たすべき制約条件（ゲート数と遅延時間の上限）の３種類である。動作仕様は、状態遷移とレジスタ転送などを表す仕様記述言語で記述したものであり、これだけでも仕様を入力することができるが、設計者が機能ブロックをイメージしている場合は、ブロック図を入力することにより、設計者の意図を反映させることができる。

動作仕様から、レジスタ転送とターミナル結合に関するオペレーション（データパスオペレーション）とステート遷移に関するオペレーション（制御オペレーション）を抽出し、これらのすべてがブロック図で表されたデータパス上で実行可能なことをチェックする。この過程において、機能ブロックとその接続関係、および制御回路の仕様を決定する。

制約条件は、デフォルトNJと呼ぶ、制約条件と等価な不等式（不等式が成立することが制約条件違反の必要十分条件になる）に書き換える。設計の実行中に生成されるNJは、デフォルトNJを起源とし、その成立が制約条件違反の十分条件になる論理式である。NJは、属性値（ゲート数または遅延時間）を示す変数と属性値、制約値を示す変数を用いて表す。

機能ブロック（制御回路も制御ブロックと呼ぶ機能ブロックのひとつ）とその接続情報に書き換えられた仕様を、部分仕様に分割する。分割は、並列協調アルゴリズムの特性を考慮して、「クリティカルパスの候補をなるべく少ない数のエージェントが担当するように分割する」という方針で行なう。回路分割に従って、デフォルトNJもエージェントの属性値を示す変数を使用したものに書き換える。

部分仕様と、デフォルトNJを各エージェントに通知する。エージェントは協調して、部分回路仕様の詳細化と評価を繰り返すことにより、制約条件を満たす回路を設計する。出力は、CMOSスタンダードセルの接続情報である。もし、制約条件を満足する回路を生成することができなかった場合は、違反した制約条件を提示し、ユーザが制約条件を変更することによって、再設計を開始することができる。制約条件の変更による再設計においては、蓄積されたNJを再利用する。

3．並列協調方式

集中管理部を持たない、本並列協調方式の特徴は、以下の2点である。

(1) 部分仕様を独立に（並列に）詳細化する。

(2) 各エージェントは、全体の制約条件を満たすために、自律的に軌道修正する。

3．1　並列協調アルゴリズム

図2に沿って説明する。並列協調は、部分仕様とデフォルトNJが通知された時点で開始する。以下の処理は、各エージェントが独立に（並列に）行なう。エージェント間の同期は、属性値の通知とNJの通知によって実現されている。例えば、【5】では、自分自身（エージェント）が成功したとき、他の（少なくても一つの）エージェントが失敗するか、あるいは、他のすべてのエージェントが成功するまで処理を待つ。

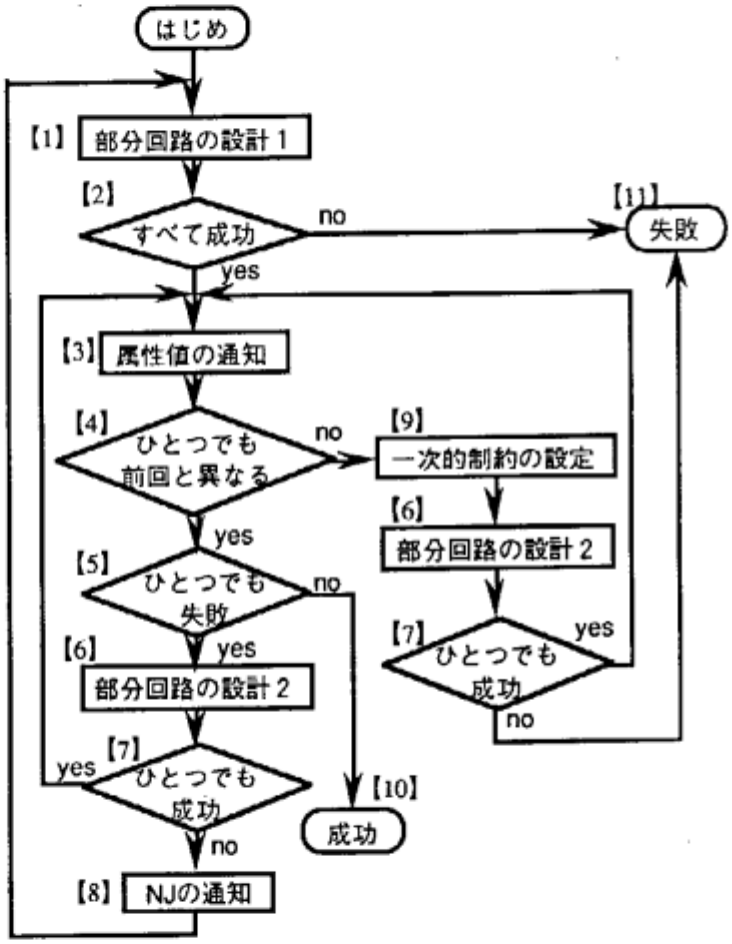


図2　並列協調アルゴリズム

【1】蓄積しているNJ（初めはデフォルトNJだけ）を成り立たせないように、部分回路を設計する。他のエージェントが担当している部分は0として扱う。

【2】少なくともひとつのエージェントが部分回路の設計に失敗すると、与えられた制約を満たす回路は存在しない。すべてのエージェントが成功したことを確認する。

【3】各エージェントの設計結果（部分回路の属性値）を通知し合う。

【4】ひとつでも前回と異なる結果があるかを調べる。

【5】通知された設計結果を代入しても、デフォルトNJが成立しなければ成功である。

【6】蓄積しているNJを成り立たせないように、部分回路を設計する。他のエージェントが担当している部分は【3】で通知された値を使用する。

【7】少なくともひとつのエージェントがNJを成立させない結果を得られたならば、成功した結果で置き換えて、【3】へ。この処理により、違反が生じている制約の数は減少する。失敗したエージェントには、エージェントが担当する部分回路に相当する変数を含まないNJが生成されている。

【8】各エージェントが生成したNJを関係するエージェントに通知する。通知されたNJは、「NJの組み合わせ」を行なうことにより、新しいNJとして蓄積される。

【9】違反している制約のうち、関係するエージェントが最少のものをひとつ選び、この制約を厳しくした

一時的制約を与える。

【10】部分回路を結合する。

【11】現在の制約条件では設計不可能である。制約条件を緩和することにより、蓄積されたNJのいくつかを無視できるようになり、再設計を開始できる。

3．2　NJの組み合わせ

NJの組み合わせは、他のエージェントの設計不可能範囲から、自分自身の設計範囲を制限するNJを生成する処理である。例を用いて説明する。3つのエージェントに関係する制約があり、すべてのエージェントで再設計不可能であるとする。このとき、次のNJが生成されている。A1、A2、A3はそれぞれエージェントの属性値に対応する変数である。

エージェント1：5＋A2＋A3＞C，　エージェント2：A1＋10＋A3＞C，　エージェント3：A1＋A2＋8＞C

これらを互いに通知し合う。エージェント1においては、A1＋18＞Cという新しいNJが生成される。このNJにより、A1を除く部分に最低でも18必要なことがわかり、エージェントの条件が厳しくなる。部分回路の設計1（図2）においては、他のエージェントの担当部分を0として扱っているが、この時でも他のエージェントの設計不可能範囲を考慮することができるようになる。

4．試作システム

本試作システムは、co-LODEXのエージェント間の並列協調方式と、エージェントが独立に（並列に）行なう設計―評価―再設計の過程を実現したものである。

4．1　試作システムの概要

図3は、co-LODEXの構成を示すブロック図であり、試作したシステムは、網掛け部分に相当する。

入力は、分割されるべき回路の全体仕様と、エージェントへの割り当て、制約条件と等価なデフォルトNJである。各部分仕様は、機能ブロックの端子名とそのビット幅などの仕様と、機能ブロック間の接続関係である。部分回路仕様とデフォルトNJは、詳細化部への入力である。詳細化部は回路の部分仕様を階層的に詳細化する。機能ブロックライブラリは、まとまった機能を持つ機能ブロックの仕様と、機能ブロックの詳細化のルール、テクノロジマッピングのルールが格納されているライブラリである。セルライブラリは、セルの特性の情報が格納されているライブラリである。設計結果はセルの接続情報として出力される。制約条件を満足する回路を設計できなかった場合は、違反情報（NJ）が報告される。

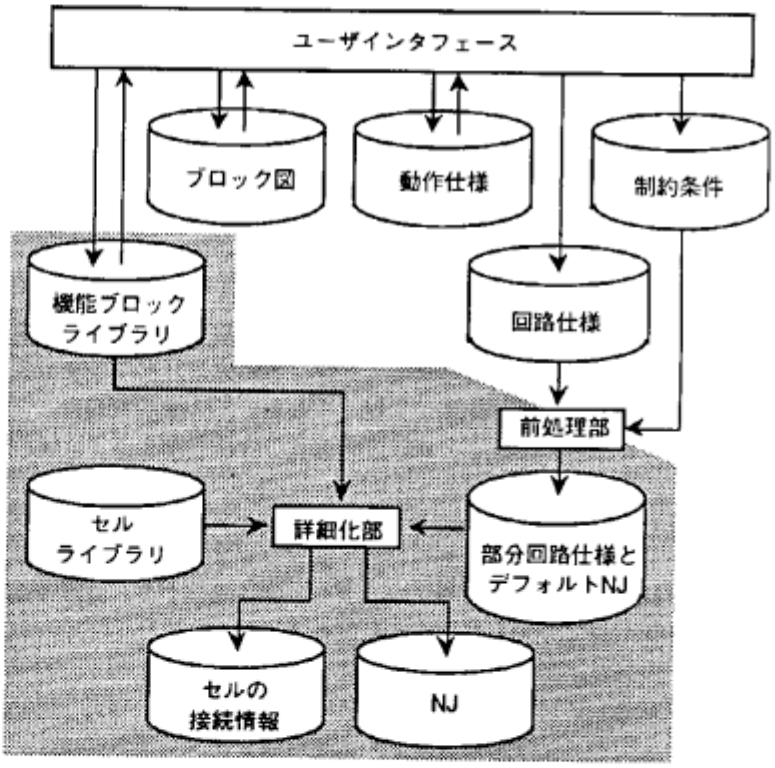


図3　co-LODEXの構成図

これらの構成要素のブロックのうち、前処理部は全体に共通のものであるが、詳細化部は各エージェントが個別に持っている。各エージェントは階層設計により、セルの接続情報や違反情報を生成する。こうした一連の処理のなかで、機能ブロックライブラリ、セルライブラリが共通に参照される。

4．2　インプリメンテーション

試作システムの各構成要素はプロセスであり、メッセージ通信をしながら処理を進める。

設計の詳細化部における、設計の階層を構成するプロセスの概念図を図4に示す。楕円はプロセスを表し、矢印は入力／出力ストリームを表す。図4の概念図は、設計が進行したある状況におけるプロセスによる構成を表すものであり、設計の初期状態では、入出力のプロセスだけが存在する。入出力以外のプロセスは、設計の進行に応じて生成する。

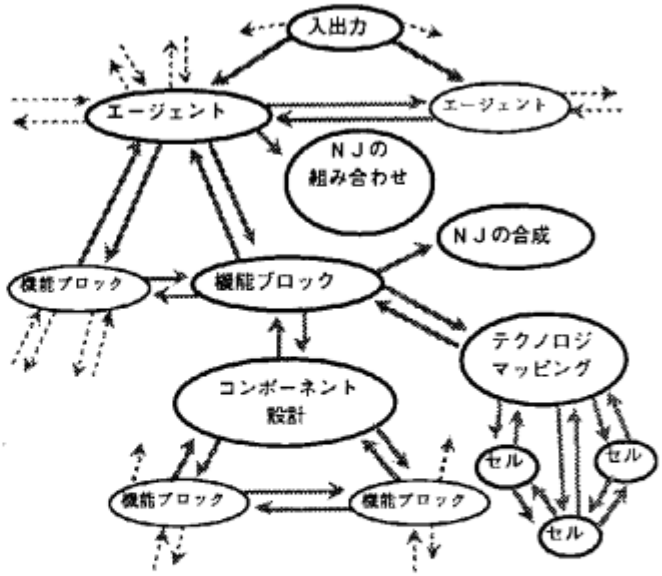


図4　プロセスの概念図

入出力のプロセスは入力された回路の全体仕様を分割し、分割した回路の部分仕様と関係するデフォルトNJを各エージェントに通知する。

エージェントのプロセスは並列協調方式に従う。部分仕様が通知されると、制約条件を考慮しない詳細化を開始する。詳細化のために、機能ブロックプロセスを生成する。デフォルトNJが通知されると、エージェントの属性に関する上限値を示す、エージェントの制約条件（設計を担当する機能ブロックに関する局所的な制約条件）を生成する。設計の進行に伴って、他のエージェントから属性値またはNJが通知されたときも、同様な制約条件を生成する。エージェントの制約条件をもとに、機能ブロックの属性に関する上限値を生成し、担当する機能ブロックへ一斉に通知する。上限値は機能ブロックの評価において何度か変更する。

NJの組み合わせプロセスは、他のエージェントの設計不可能範囲から、設計範囲を制限する処理を行なう。NJの組み合わせプロセスはエージェント毎に生成する。

機能ブロックのプロセスは、設計ルールを参照し、詳細化することにより設計を行なう。すなわち、一つ下位のオルタナティブ（コンポーネント設計／テクノロジマッピング）のプロセスを生成し、1レベル詳細化した機能ブロック／セルの仕様をオルタナティブプロセスに通知する。上限値が通知されたときは、下位のオルタナティブに上限値を越えない設計を依頼する。設計に成功したときは結果を上位のプロセスに返し、このオルタナティブをIN状態として記録する。失敗したオルタナティブもOUT状態として記録しておく。すべてのオルタナティブが上限値を越えたとき、NJの合成プロセスにより、これ以上厳しい条件のときの再設計を禁止する情報を生成し、設計可能な最小の属性値を返す。NJの合成プロセスは機能ブロック毎に生成する。

コンポーネント設計オルタナティブのプロセスは、機能ブロックを1レベル詳細化した構成法に対応している。設計時には、上位の機能ブロックプロセスから与えられた上限値を越えないような、下位の機能ブロックの上限値を生成し、機能ブロックへ一斉に通知する。設計の進行に伴って、生成する上限値は下位の機能ブロックの評価において何度か変更する。

テクノロジマッピングオルタナティブのプロセスは、セルへのマッピングの方法に対応している。構成要素のセルが通知されると、セルのプロセスを生成する。与えられた上限値を越えるか否かの結果と、設計結果（ゲート数と遅延時間）を上位の機能ブロックプロセスに返す。セルプロセスへのゲート数、遅延時間の計算の依頼は、各セルへ一斉に通知する。セルプロセスはセルに対応し、セルの特性値を持つ。

5．実験

実験に用いた例題は、2階微分方程式 $y'' + 5xy' + 3y = 0$ を解く回路［4］である。UHDL（Unified Hardware Description Language）［5］で記述した、回路の動作仕様を図5に示す。回路規模として、本例題のデータパス（図6）はマルチプレクサ、レジスタ、乗算器などの28個の機能ブロックで構成される。組み合わせ問題として捉えると、2兆7千億個の組み合わせがあり、階層設計上には350個の機能ブロックがある。階層設計

```
UHDL;
manage_view: example01;
    system: example01;
interface_view: interface_example01;
    purpose: demonstration;
    inputs: .xi(12), .yi(12), .dxi(12), .ui(12), .ai(12);
    outputs: .xo(12), .yo(12);
behavior_view: behavior_example01;
    purpose: demonstration;
    define: const5 = 5, const3 = 3;
    terminal: u1(12), u2(12), u3(12), u4(12), u5(12), u6(12), y1(12), FF;
    operator: 2stage_pipelined_multiplier(x, y, z) = ( len = 2 ),
            z <- x * y; end_op;
    function: main: clk;
        while (FF) do
            2a: '2stage_pipelined_multiplier'(u, dx, u1);
            3a: '2stage_pipelined_multiplier'(x, const5, u2);
            4a: '2stage_pipelined_multiplier'(const3, y, u3);
            5a: '2stage_pipelined_multiplier'(u2, u1, u4),
                x <- x + dx;
            6a: '2stage_pipelined_multiplier'(u, dx, y1),
                FF <- x < a;
            7a: '2stage_pipelined_multiplier'(u3, dx, u5),
           u6 <- u - u4;
            8a: y <- y1 + y;
            9a: u <- u6 - u5, xo := x, yo := y;
      enddo;
      1a: stop(x<a), x <- xi, y <- yi, dx <- dxi, u <- ui, a <- ai;
endUHDL.
```

図 5　2 階微分方程式 y" + 5xy' + 3y = 0 を解く回路の動作仕様

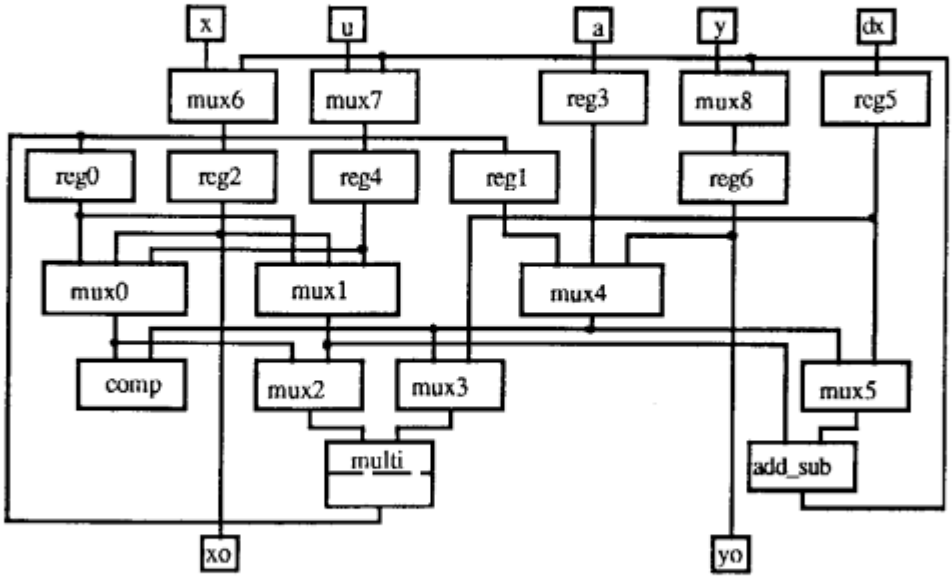


図 6　2 階微分方程式を解く回路のデータパス

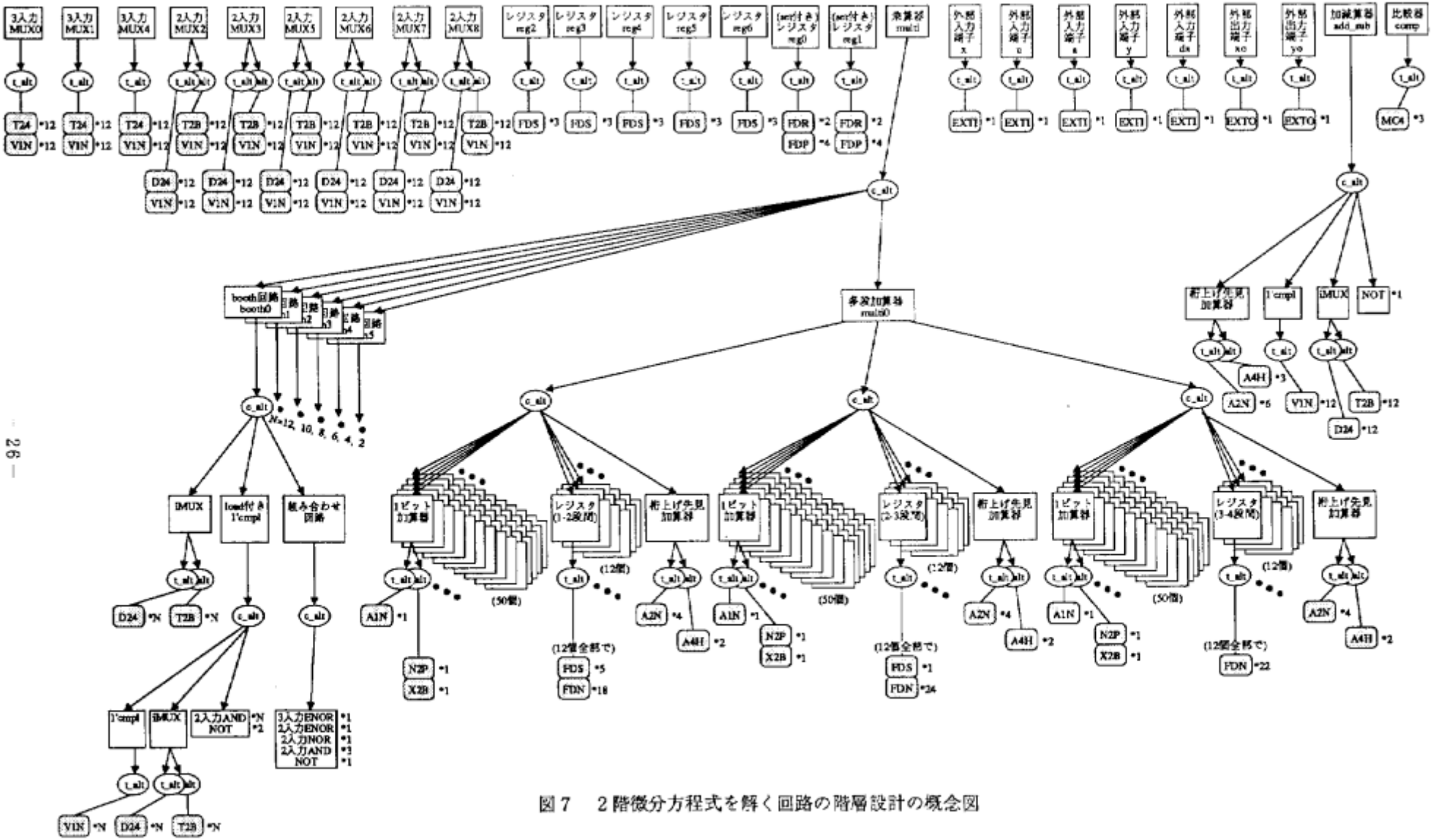


図7　2階微分方程式を解く回路の階層設計の概念図

の概念図を図７に示す。図中の四角は機能ブロックを、楕円はオルタナティブのノードを表す。

並列協調方式を実現した、試作システムを動作させるためには、入力仕様であるデータパスを分割し、エージェントへ割り当て、プロセッサへマッピングする必要がある。試作システムの性能は、

（1）エージェントへの分割方法。

（2）プロセッサへのマッピング。

に依存する。そこで、（1），（2）をいくつか変更した場合の実験を行なった。

５．１　実験（１）

エージェントへの分割方法において、高い台数効果を得るためには、エージェント間通信を抑える必要がある。そこで、我々はエージェントへの分割の方針として、『クリティカル・パス（の候補）上の機能ブロックをなるべく少ないエージェントで設計する。』と考えている。遅延時間制約は１クロックで信号が転送される、データパス上のいくつかの機能ブロックを通るパスに関する制約である。なるべく少ないエージェントに分割すると、一つの遅延時間制約に関係するエージェントが少なくなるから、エージェント間通信が少なくなると考えられる。この分割の方針において、設定するエージェントの数が問題である。エージェントの数を増やし、個々のエージェントの担当する機能ブロックの数を減らすことは、エージェントの設計作業を小さくし、並列実行の効果が期待できるが、一方で、エージェント間通信が全体として増加するので、並列実行の効果を妨げる要因となる。

表１に示した処理時間の（約133秒を１とした）比は、エージェントへの分割方法の実験結果である。

表１．エージェントへの分割方法の実験結果

制約＼分割	3 (1) agents	5 agents	10 agents
十分緩い	0. 09	0. 08	0. 09
↕	1 (133秒)	0. 18	0. 41
十分厳しい	4. 19	4. 01	——

表の各列は設定したエージェントの数を、各行は制約条件の違いを示す。設定したエージェントの数は、1，5，10個とした。ただし、試作システムの都合で1エージェントの代わりに3エージェントとし、一つのエージェントがほとんどすべての機能ブロック（26個）を担当するようにした。制約条件が十分緩いとは、再設計を行なわずに、一度の設計で制約を充足する回路が得られるような場合である。2行目の制約条件は、再設計を行なうことによって制約を充足する回路が得られるような場合の一例である。制約条件が十分厳しいとは、制約を充足する回路が存在しない制約を与えた場合である。このとき、システムは設計に失敗し、制約条件違反情報を生成する。また、用いたPEの数は、エージェントが3, 5, 10個のときそれぞれ4, 6, 11(PEs)である。

この結果で興味深いのは、２行目に示した処理時間—これは、我々が想定している使われ方に対応している—である。５エージェントの結果が0.18と短くなっている。3エージェントの結果は、一つのエージェントに処理が集中していることにより、10エージェントの場合は、分割し過ぎでエージェント間の通信が多いことにより、処理時間が掛かっていると考えられる。

この例題では、5エージェントが良い結果となるが、もっと規模の小さい回路の実験では、5より少ないエージェントで十分であるという結果を確認している。しかし、必要なエージェント数に対してプロセッサは（特にPIMでは）十分に多いので、エージェントにおける処理を分散させることが考えられる。

5．2　実験（2）

プロセッサへのマッピングの実験は、エージェントの処理を分散させる実験である。表2に示した処理時間の比は、プロセッサへのマッピングの実験結果である。この実験では、(a)データパス上の機能ブロックだけを分散させた場合、さらに、(b)階層設計上の機能ブロックすべてを分散させた場合について行なった。

表2．プロセッサへのマッピングの実験結果

分散 / 分割	なし	(a)データパス上の機能ブロック	(b)すべての機能ブロック
3 agents	1（133秒）	0. 37	0. 20
5 agents	0. 18	0. 18	0. 20

表2で用いた制約条件は、再設計を行なうことによって制約を充足する回路を得る場合（表1の2行目）である。また、用いたPEの数は、3エージェントのとき左から、4，16，16(PEs)、同様に、5エージェントでは、6，15，16(PEs)である。

3エージェントでは細かく分散させるに従って処理速度が向上している。これは、エージェント間の通信がないため、並列実行の効果がでているからと考えられる。一方、5エージェントの結果では、処理速度はほとんど同じである。これは、並列実行の効果と通信の増加による負荷（同期をとるための待ちを含む）の増大がほぼ釣り合っているからと考えられる。

6．おわりに

階層設計上の各枠組みをプロセスで実現することにより、協調型論理設計エキスパートシステムを自然な形で実現することができた。

実験では、適切なエージェントへの分割によって、約1／（エージェントの個数）の処理時間で設計できることを確認した。さらに、プロセッサを有効に利用することにより、もっと速く処理できる可能性がある。そのために、エージェント内の処理の並列化や、論理設計における効果的な負荷分散方式の実現を考えている。

謝辞

本研究は第五世代コンピュータプロジェクトの一環として行なわれているものであり、御支援頂いたICOTの生駒研究部長代理（現在、NTTデータ通信株式会社）、新田第七研究室長に深く感謝いたします。

参考文献

[1] 丸山他　「評価・再設計機構を備えた論理設計支援システム」　信学会論文誌　1989.8

[2] 箕田他　「協調型論理設計エキスパートシステムco-LODEX　—概要—」
情処学会第42回全国大会 1991.3

[3] 澤田他　「協調型論理設計エキスパートシステムco-LODEX　—試作—」
情処学会第42回全国大会 1991.3

[4] Forrest D. Brewer et al. Knowledge Based Control in Micro-Architecture Design,
24th ACM/IEEE Design Automation Conference (1989).

[5] H. Fujisawa et al. UHDL(Unified Hardware Description Language) and its support tools,
Int. J. Computer, Aided VLSI Design (1989).

バーチャルタイム による並列論理シミュレーション

松本 幸則 瀧 和男
(財) 新世代コンピュータ技術開発機構

概要

並列論理シミュレーションは、並列イベントシミュレーションの問題として取り扱うことができる。並列イベントシミュレーションでは、時刻の管理機構が重要な問題となる。並列処理に適した分散時刻管理機構を持つものとして、コンサーバティブ法とバーチャルタイムの概念に基づく方法 (バーチャルタイム法) がある。コンサーバティブ法は、デッドロック回避のオーバヘッドが問題である。バーチャルタイム法は、デッドロックの危険性がないという利点がある半面、ロールバック処理が必要になる。

我々は、分散メモリ型並列マシン「Multi-PSI」上にバーチャルタイム法による論理シミュレーションシステムを構築し、性能評価を行った。その結果、速度向上、絶対性能の両面で良好な結果を得た。

さらに、ヌルメッセージを用いたコンサーバティブ法、および、集中時刻管理機構による並列論理シミュレーションの実験も行い、バーチャルタイム方式との性能比較を行った。その結果、並列論理シミュレーションの方法としては、バーチャルタイム法が最も有効な方法であることを確認した。

1 はじめに

論理シミュレーションはLSI設計工程の中で設計仕様の検証、とくに回路の論理の検証、信号伝播遅延の検証に重要な役割を果たしている。しかしながら膨大な時間が費やされてしまう点が大きな問題であり、高速シミュレータに対する要求は強い。また、高精度のシミュレーションに対する要求も非常に強くなってきている。

ハードウェアエンジンを用いることは、シミュレーションの高速化という視点からは良い手段であるが、高精度シミュレーションの要求に柔軟に対応することは困難である。ソフトウェアによる論理シミュレーションを並列化することは高速且つ高精度シミュレーションを実現する有望な方法であると考えられ、大きな期待が寄せられている。

イベント駆動による並列論理シミュレーションの時刻管理機構は、タイムホイルと呼ばれる集中時刻管理機構 (タイムホイル法) と、分散時刻管理機構の二つに大別できる [4]。タイムホイル法は、大域的な時刻の同期をとるため、多数のプロセッサを使用する場合、十分な並列性を抽出することが難しいと考えられている。分散時刻管理機構の代表的なものとしては、コンサーバティブ法 [2] およびバーチャルタイムの概念に基づく方法 (バーチャルタイム法) [1] がある。コンサーバティブ法は、デッドロックの危険性があるために、その回避のための処理が大きな問題となる。バーチャルタイム法はデッドロックの危険性がないという利点がある半面、ロールバック処理が必要であるという欠点をあわせ持つ。

我々は第五世代コンピュータプロジェクトの一環として、分散メモリ型の並列推論マシン実験機「Multi-PSI」[3] 上にバーチャルタイムによる並列論理シミュレーション実験システムを構築し、性能評価を行うとともに、ヌルメッセージを用いた場合のコンサーバティブ法及びタイムホイル法との比較を行った。

本論文では、以下第2節でバーチャルタイム法についての説明を行う。また、第3節で本並列論理シミュレーションシステムの概要を述べる。続いて、第4節で本システムの性能、速度向上についての計測結果を報告する。最後に第5節でヌルメッセージを用いたコンサーバティブ法、および、タイムホイル法との比較結果を報告し、バーチャルタイム法の優位性を示す。

2 バーチャルタイム

イベントシミュレーションは、複数のオブジェクトがメッセージ通信を行なうことによって、次々と状態を変えていく形にモデル化できる。オブジェクトは状態オートマトンとして表現され、メッセージはイベント情報を持つとともに、イベントの発生時刻がスタンプされている (タイムスタンプ)。

Parallel Logic Simulation based on Virtual Time
Yukinori MATSUMOTO, Kazuo TAKI
Institute for New Generation Computer Technology

Jefferson は、バーチャルタイムの概念と、その並列イベントシミュレーションへの応用を提案している [1]。バーチャルタイムの概念による方法 (バーチャルタイム法) には、局所的な処理と大域的な処理がある。

2.1 局所的な処理

バーチャルタイム法では、各オブジェクトは、メッセージは正しい順序、すなわちタイムスタンプの小さい順に到着するという仮定に基づいて処理を進める。しかしながら、実際には誤った順序でメッセージが到着する場合が存在する。このような状況に備え、オブジェクトはメッセージに対する処理と共に、メッセージおよび状態の履歴保存を行う。

オブジェクトは、メッセージの到着順序の矛盾を発見したところで履歴を巻き戻し（ロールバック）、処理のやりなおしをする。さらに、その時点で誤って送信したことが判明したメッセージに対しては、メッセージを取り消す役割を持つアンチメッセージなるものを送信する。上記の処理によりシミュレーション結果の正当性が保証される。

2.2 大域的な処理

バーチャルタイム法ではロールバックに備えて履歴を保存するため、メモリ消費が重大な問題となる。このため、時々大域的なシミュレーション時刻 (GVT) を求める必要がある。GVT は、ある時点での、全オブジェクトのシミュレーション時刻、及びオブジェクト間を通過中のメッセージのタイムスタンプ値のうちの、最小値以下のものである。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することができる。

3 実験システム概要

3.1 実行環境

本実験システムは、並行論理型言語 KL1[8] で記述され、Multi-PSI[3] 上に実装されたシステムである。Multi-PSI は MIMD 型マシンで、要素プロセッサ (PE) 64 台が 2 次元メッシュ状のネットワークで結合されている。メモリは全て分散管理されており、他の要素プロセッサへのデータアクセスコストすなわち PE 間通信コストは高いが、台数拡張性に富む。

3.2 仕様

本シミュレーションシステムでは、ゲートレベルで記述された回路を扱う。回路としては、組み合わせ回路、同期回路のみならず、非同期回路も扱う。

信号値は Hi、Lo、X(不定) の 3 値モデルとし、遅延は各ゲートに単位時間の整数倍を割り当てるようなノンユニット遅延モデルとする。本システムの目的は並列論理シミュレーションの実験であることから、最低限の一般性を持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。

3.3 構成

本システムは、前処理部とシミュレーション部の二つの部分から成る。

前処理部は、並列シミュレーションに備え負荷分散を決定する。ここでは、今回提案した縦割り指向戦略に基づき回路データを分割し、各 PE に静的に割り当てている。

シミュレーション部では、バーチャルタイム法による並列シミュレーションを行う。ここではロールバック頻度低減のため各 PE に局所メッセージスケジューラを置く。また、ロールバックコスト低減のためアンチメッセージの削減処理も行う。

3.3.1 局所メッセージスケジューラ

本システムでは、各ゲートが第 2 節で述べたオブジェクトに対応する。通常ゲート数は PE 数に比べはるかに多いため、各 PE 内でメッセージのスケジューリングを行うことは、ロールバック頻度低減に有効である。

本システムでのスケジューリング戦略は、PE に到着している未処理メッセージのうち、最小タイムスタンプのものから処理を行う (以後タイムスタンプソート戦略と呼ぶ) ものである。

スケジューラは各単位時間に対応したスロットを持つ。スロットはスタック構造を持ち同一タイムスタンプ値のメッセージは、到着順にスタックに登録 (プッシュ) される。スケジューラは登録中のメッセージのもつ最小のタイムスタンプ値をクロックとし、クロックに対応したメッセージを順次ポップして受信側オブジェクトに送る。

3.3.2 アンチメッセージの削減

Jefferson によるバーチャルタイム法では、メッセージ送受信における順序保存性の仮定をおいていないため、送信側でロールバックが発生した場合、取り消すべきメッセージ全てに対しアンチメッセージを送る (図 1)。

本システムでは、信号線は、KL1 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、送信者と

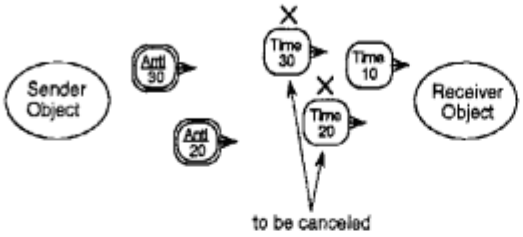


図 1: アンチメッセージ送信

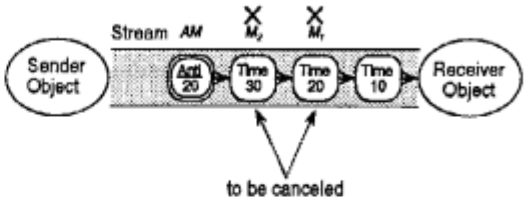


図 2: アンチメッセージ削減 (a)

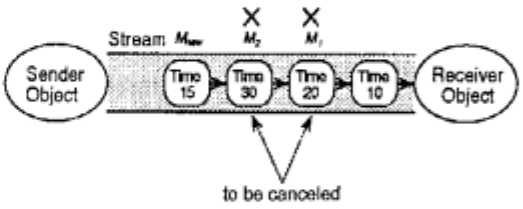


図 3: アンチメッセージ削減 (b)

受信者の間ではメッセージの送信順序は保存される。この環境では、以下のようにアンチメッセージを削減することができる [6]。

本システムでは取り消すべきメッセージのうち、最小のタイムスタンプ値を持つメッセージに対応したアンチメッセージ AM のみを送る。ここで AM のもつタイムスタンプ値を $TS(AM)$ とする。この時、受信者は、同一信号線上で受信したメッセージのうち、AM 到着以前に受信し、且つ $TS(AM)$ 以上のタイムスタンプ値を持つメッセージのみを取り消せばよい (図 2)。

本システムでは、更に次のような場合にもアンチメッセージの省略を行っている。送信側オブジェク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タイムスタンプ値以下のタイムスタンプを持つ新たなメッセージ M_{new} が発生する場合、単に M_{new} の送信を行うだけでアンチメッセージの送信は行わない。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、$TS(M_{new})$ 以上のタイムスタンプ値を持つメッセージについて取り消し処理をすれば良い (図 3)。

3.3.3 負荷分散方法

並列論理シミュレーションは一つのメッセージあたりの処理量が小さい、即ち小粒度であるためプロセッサ間通信量が問題になる。また、バーチャルタイム法では、ロールバックの発生量も問題になる。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、上記 3 点を満足するような評価関数を定義し、その値を最良にする負荷分散方法を求めなければならない。しかし、このような問題は一般に NP 困難であるため、何らかのヒューリスティックスを用いることになる。

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付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。

一般に、論理回路では、ゲートの複数ファンアウト部に多くの並列性を見出すことができる。縦割り指向戦略は、縦方向につながったゲートをグルーピングすることで回路を幾つかのクラスタに分割するものである。縦割り指向戦略は、連結したゲートはできるだけ同一クラスタとすると同時に、複数ファンアウト部の並列性を抽出することを意図している。

縦割り指向戦略によって生成されたクラスタのうち、大きさの小さいものについてはそれがつながっている別のクラスタにマージする。また、極端に長いクラスタは幾つかに切り分ける。最後に、生成されたクラスタをランダムに各 PE に割り当てる。

4 測定結果と考察

ISCAS'89 のベンチマークから 4 つの順序回路について、縦割り指向戦略に基づく負荷分散方法を用いて並列論理シミュレーションを行ない、性能、速度向上等を計測した。

対象回路のゲート数、信号線数及び平均ファンイン $\overline{Fanin}$、ファンアウト $\overline{Fanout}$ を表 1 に記す。なお、D フリップフロップはゲートに展開した。

今回の実験では、各ゲートには、全て 1 単位時間の遅延値を与えた。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列を与えた。

表 1: 対象回路

回路	s1494	s5378	s9234	s13207
ゲート数	683	3,853	6,965	11,965
信号線数	1,490	6,588	10,957	19,983
$\overline{Fanin}$	2.15	1.70	1.57	1.66
$\overline{Fanout}$	2.08	1.61	1.50	1.55

表 2: 性能 (イベント / 秒)

PE 数 \ 回路	s1494	s5378	s9234	s13207
1	1,460	1,410	1,313	1,246
2	2,240	2,624	2,642	2,570
4	3,613	4,929	4,715	5,424
8	5,702	8,731	7,605	10,753
16	7,498	16,024	11,294	20,385
32	8,857	25,210	13,958	36,930
64	8,980	40,034	21,133	59,974

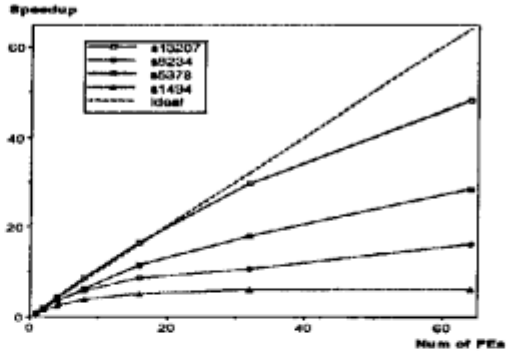


図 4: 速度向上

表 3: ロールバック頻度と深さ (64PE 使用時)

回路	s1494	s5378	s9234	s13207
f_r	0.6090	0.0696	0.0562	0.0227
d_r	2.700	3.691	11.107	7.310

4.1 測定結果

表 2に各回路のシミュレーションにおける処理性能を示す。また、図 4に速度向上のグラフを示す。

表 3には、64PE 使用時のロールバックの頻度 f_r と平均的深さ d_r を示す。f_r, d_r は、以下の式で定義する。

$$f_r = R/E$$

$$d_r = H_r/R$$

ここで、R: ロールバック発生回数、E: 真のイベント数 (最終的に巻き戻されなかったメッセージ数)、H_r: 巻き戻された履歴数である。

表 4には、64PE 使用時の PE 間メッセージ通信の頻度 f_c を示す。f_c は、以下の式で定義する。

$$f_c = M_c/M_{all}$$

ここで M_c: PE 間を移動したメッセージ数、M_{all}: 全メッセージ数である。

さらに、バーチャルタイム法における処理プリミティブのうち、以下に示すものについて処理時間を計測した。

GVT 更新処理

使用プロセッサ数を変えて、GVT 更新に必要な処理時間を計測した。表 5にその結果を示す。なお、各プロセッサでのシミュレーション時刻はメッセージスケジューラが管理しているため、GVT 更新処理時間は対象回路に依存しない。

履歴解放処理

一回の GVT 更新後の履歴解放処理時間 t_f は、ゲートオブジェクト数 G と解放履歴数 h_f に比例する。

$$t_f = k_f h_f + k_g G + C_f \qquad (1)$$

ここで C_f は定数分である。計測の結果、$k_f = 0.027$ (ミリ秒)、$k_g = 0.302$ (ミリ秒)、$C_f = 0.036$ (ミリ秒) であった。

ロールバック処理

一回のロールバック発生時の履歴巻き戻し処理時間 t_r は、巻き戻された履歴数 h_r に比例する。

$$t_r = k_r h_r + C_r \qquad (2)$$

ここで C_r は定数分である。計測の結果、$k_r = 0.027$ (ミリ秒)、$C_r = 0.367$ (ミリ秒) であった。

PE 間メッセージ通信

本システムでの一つのメッセージ (20 バイト) の PE 間通信にかかる処理時間は 0.256 ミリ秒であった。

4.2 考察

64PE 使用時の速度向上として、s13207, s5378 については良好な結果が得られたが、s9234, s1494 についてはやや不満足な結果となった。

速度向上に影響を与えるものとして 1 : GVT 更新と履歴解放処理、2 : 問題の並列性、3 : 静的

表 4: PE 間メッセージ通信頻度 (64PE 使用時)

回路	s1494	s5378	s9234	s13207
f_c	0.4016	0.1206	0.08458	0.02219

表 5: GVT 更新処理時間

PE 数	GVT 収集処理時間 (1 回あたり、ミリ秒)
8	6.286
16	13.598
32	23.851
64	49.168

負荷分散 (負荷の不均等)、4： ロールバック処理、5：PE 間メッセージ通信、の 5 点が考えられる。以下、これらについて考察を行う。

4.2.1 GVT 更新と履歴解放処理

表 5 より、GVT 更新 1 回あたりの処理時間は 64 PE 使用時でも約 49 ミリ秒と非常に短いことがわかる。GVT 更新頻度は低いため、GVT 更新処理はシステム全体の性能にはほとんど影響しないと考えて良い。

さて、PE i 番での履歴解放に関わる処理時間 $T_f(i)$ は、シミュレーションの開始から終了までの期間、n 回 GVT 更新されたとすると、式 (1) から

$$T_f(i) = \sum_{i=1}^{n} t_f(i) = k_f E_i + n k_g G_i + n C_f \quad (3)$$

ここで、E_i は PEi での真のイベント数であり、G_i は PEi に割り当てられたゲート数である。E_i および、G_i は使用 PE 数にほぼ反比例すると考えられる。

GVT 更新の目的はメモリ消費を少なくすることである。各 PE は等量のメモリを持っているとすると、GVT 更新回数 n も使用 PE 数にほぼ反比例すると考えられる。結局、式 (3) 右辺 nk_GG の項は使用 PE 数の 2 乗にほぼ反比例する。いま、各 PE に均等にゲートが分配され、均等にイベントが発生したとする。s13207 での真のイベント数は 2,341,374 であり、2PE 使用時の GVT 更新回数は 62 回であった。以上から、2PE 使用時の $k_{h_f} E_i$ の値はほぼ 31.6 秒、nk_GG_i の値は 115.6 秒、nC_{i_f} の値は 0.002 秒と見積もれる。したがって $T_f(i)$ に対する nk_GG の項の影響は無視できない。

図 4 において、2PE~16PE 使用時にはスーパリニアな速度向上が観測されているが、これは使用 PE 数が増えるに連れ、各 PE での履歴解放に要する処理時間が短くなるためと考えられる。

4.2.2 問題の並列性と静的負荷分散 (負荷の不均等)

ここでは問題の持つ並列性についての考察を行う。以下、簡単のためメッセージの処理以外のコストは全て無視できるような場合を考える。

論理シミュレーションの問題 Prb は、回路構造 c、入力信号列 v、及びシミュレーション期間 t によって決定される。

$$Prb = Prb(c, v, t)$$

このとき、問題 Prb を PE N 台への負荷分散方法 $Dst(N)$、スケジューリング戦略 Sch を用いて解く場合の計算時間を $T(Prb, Dst(N), Sch)$ とする。そして、問題自身の持つ並列度 $P_{[Prb]}$ を

$$P_{[Prb]} = \frac{T(Prb, Dst(1), Sch_{tss})}{T(Prb, Dst(\infty), oracle)}$$

と定義する。ここでの負荷分散方法 $Dst(\infty)$ は各ゲートオブジェクトを全て異なる PE に割り当てるものである。また、スケジューリング戦略 Sch_{tss} は、タイムスタンプソート戦略とする。タイムスタンプソート戦略は、全てのゲートオブジェクトが一つの PE に存在する場合、最適なスケジューリング戦略である。また $oracle$ は、常に最適なスケジューリングを与えるものとする。$P_{[Prb]}$ は、ゲート数以上の PE が存在する場合に得られる速度向上の上限となる。

現実には、PE 数は有限であり、しかもゲートオブジェクト数より遥かに少ないことが一般的である。通常は、何らかの戦略に基づいて回路を分割し、部分回路を PE に割り当てる負荷分散方法をとる。

PE N 台への負荷分散方法 $Dst(N)$、スケジューリング戦略 Sch を用いた場合の問題の並列度 $P_{[Prb,Dst(N),Sch]}$ を、

$$P_{[Prb,Dst(N),Sch]} = \frac{T(Prb, Dst(1), Sch_{tss})}{T(Prb, Dst(N), Sch)}$$

と定義する。

各メッセージの処理コストは全て等しいと仮定し、実際に行ったシミュレーションについて、それぞれの問題自身が持つ並列度 $P_{[Prb]}$ を実験的に求めた。

また、縦割り指向戦略に基づく静的負荷分散方法を用いて 64PE に負荷を分散し、スケジューリング戦略としてタイムスタンプソート戦略を用いた場合の、各問題についての並列度 $P_{[Prb,Dst(64),Sch]}$ も実験的に求めた。

さらに、一回の GVT 更新にともなう履歴解放の処理時間は、どの PE でも等しいと仮定し、表 2 およ

表 6: 並列度

回路	s1494	s5378	s9234	s13207
$P_{[Prb]}$	55.65	298.6	487.7	609.3
$P_{[Prb,Dst(64),Sch]}$	18.88	35.52	17.95	43.24
$S_{[T_f=0]}$	5.83	25.46	13.47	38.17

び式 (3) を用いて、履歴解放処理時間を除去した場合の 64PE 使用時の速度向上 $S_{[T_f=0]}$ を計算した。表 6 に、$P_{[Prb]}$, $P_{[Prb,Dst,Sch]}$, $S_{[T_f=0]}$ の値を示す。

表 6から、s1494 に関しては、問題の持つ並列度 $P_{[Prb]}$ 自体が非常に小さいことが分かる。また、s5378, s9234, および s13207 については、負荷分散及びスケジューリング戦略による並列度が速度向上を制限していることがわかる。とくに、s9234 については、負荷分散方法及びスケジューリング戦略によって極端に並列度が抑えられてしまっている。

表 6は、全てのメッセージ処理のコストが等しいと仮定していているため、必ずしも現実と完全に一致するものではない。しかし、問題自身の並列性、および負荷分散戦略、スケジューリング戦略による問題の並列度低下が、速度向上を抑える最大の原因であることが予測できる。とくに、負荷分散戦略については、改良の余地が大きいと考えている。

4.2.3 ロールバック処理

ロールバック処理は、バーチャルタイム法の処理速度を低下させる最大の原因と考えられがちである。しかし、発生したロールバックが全て処理速度を低下させているわけではない。一般にロールバックは、シミュレーション時刻が将来に進み過ぎている PE にのみ発生する。

極端な例として、シミュレーション時刻が常に遅れている PE が存在した場合を考えてみる。この場合、ロールバックは先行した PE においてのみ発生し、決して全体の処理を遅らせることはない。

s13207 の場合、平均ロールバック処理時間は表 3 および、式 (2) から、0.469 ミリ秒となる。一方、履歴解放処理時間を除去した場合の、1 メッセージあたりの処理時間は 0.618 ミリ秒である。ロールバック発生頻度 f_r は 0.0227 と非常に低いことから、s13207 の場合、ロールバック処理は全体の処理にほとんど影響を与えないと考えられる。s1494 については、平均ロールバック処理時間 : 0.440 ミリ秒、1 メッセージあたりの処理時間 : 0.647 ミリ秒であり、ロールバック発生頻度 f_r が 0.6090 と高いため、ロールバック処理はある程度全体の処理に影響を与えていると思われる。しかし、実際にシミュレーションの進行を遅らせているものがどの程度かを正確に計測することは非常に難しい。

4.2.4 PE 間通信

表 4より、s13207, s9234 の場合、PE 間通信頻度が比較的低く抑えられているが、s1494 ではかなり PE 間通信頻度が高くなっていることがわかる。一般に、プロセッサ数が一定の環境では、ゲートオブジェクト数が少ないほど、また、一つのオブジェクトのファンイン、ファンアウト数が多いほど PE 間通信頻度も大きくなると考えて良い。表 1からわかるように、s1494 は比較的ゲートオブジェクト数が少なく、平均ファンイン、ファンアウト数が大きい。s1494 の場合に PE 間通信が多くなるのはこれらの理由によるものであると考えられる。

PE 間通信がシミュレーション性能に与える影響については、s13207, s9234 の場合、比較的小さいと考えて良かろう。しかし、s1494 では、シミュレーション性能を低下させる要因の一つであると考えられる。

以上の考察から、速度向上が悪いものについては、負荷分散およびスケジューリング戦略による並列性低下がその主な原因であり、s1494 の場合を除いて、ロールバック処理、および PE 間メッセージ通信コストの影響は比較的小さいことが分かった。今回の実験では、バーチャルタイム法は、ほとんどの場合、問題の持つ並列性を比較的効率的に引き出していると考えられる。

5 時刻管理機構の比較

本節では、絶対性能の観点から、他の時刻管理機構とバーチャルタイム法との比較を行い、各時刻管理機構の有効性の検討を行う。

5.1 コンサーバティブ法

5.1.1 機構

コンサーバティブ法は、バーチャルタイム法と同様に分散時刻管理機構の一つである [2]。

コンサーバティブ法でも、各オブジェクトがメッセージ通信を行うことによってシミュレーションが進行するモデルを考える。この方法では、メッセージ通信において、同一信号線上メッセージの送受信順序が保存されている環境を考える。各オブジェクトは、自身の全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待ち合わせを行う。そして、各入力信号線上にメッセージが到着後、最小のタイ

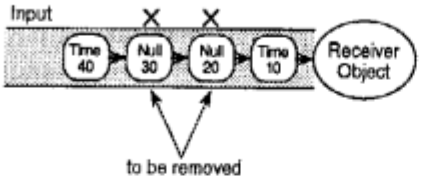


図 5: ヌルメッセージ削減機構

表 7: コンサーバティブ法 (回路 s13207)

PE 数	全メッセージ数	ヌルメッセージ数	性能 (イベント / 秒)
2	94,783,392	92,442,018	91
4	98,768,309	96,426,935	164
8	101,437,325	99,095,951	291
16	102,163,983	99,822,609	503
32	102,589,874	100,248,500	953
64	104,182,696	101,841,322	1,684

ムスタンプ値を持つメッセージに対して処理を行う。その後、再びメッセージの待ち合わせを行う。

5.1.2 ヌルメッセージ削減機構

コンサーバティブ法で最も大きな問題となるのが、デッドロックの発生である。ヌルメッセージを用いる方法はデッドロックを発生させない有効な方法であるが、現実にはヌルメッセージが非常に多く発生してしまう点が問題となる。ヌルメッセージを削減するために幾つかの方法が考案されている [2]。

図 5に一つの例を示す。この例は、同一信号線上でヌルメッセージを受信した直後に、更に別のメッセージを受信した場合を示している。この場合、ヌルメッセージを即座に消去できる。

5.1.3 計測結果および考察

上述したヌルメッセージ削減機構をもつコンサーバティブ法による論理シミュレーションを行い、性能、ヌルメッセージ数を計測した。

対象回路は s13207 であり、負荷分散方法およびスケジューリング戦略は第 4節と同様のものである。表 7に、各々の計測結果を示す。

ヌルメッセージ数

表 7から、発生したメッセージのほとんどがヌルメッセージであったことがわかる。

ヌルメッセージ削減機構のない場合、一つのメッセージ受信毎に必ずファンアウト数分だけメッセージを送信する。一般に論理回路でのゲートの平均ファンアウト数は 1 より大きいと考えられる。したがって、ヌルメッセージも含めた全メッセージ数は指数関数的に増大することが予想される。実際には、タイムスタンプ値、及び各ゲートでの遅延値が離散的であるために上限が存在する。このメッセージ数の上限は (シミュレーション時間長) × (全信号線数) で与えられる。

s13207 の信号線数は 19,983 である。また実験では期間 10,000 単位時間のシミュレーションを行ったため、メッセージ数上限は 199,830,000 となる。これに対し、実際のメッセージ数は 64PE 使用時で 104,182,696 である。ヌルメッセージ削減機構を用いても、上限値の約半分のメッセージ発生があったことになる。

速度向上と絶対性能

速度向上の面からみれば良好な値を得ることができている。しかしながら、絶対性能は非常に悪い。

PE1 台使用時のヌルメッセージを含めたメッセージ処理性能の測定結果は 2,006 メッセージ / 秒であり、この値はバーチャルタイム法を凌ぐ。それにもかかわらず、コンサーバティブ法によるシミュレーションの絶対性能が悪いのはヌルメッセージ数があまりに多いためである。

ヌルメッセージを用いたコンサーバティブ法は、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。

5.2 タイムホイル法

5.2.1 機構

タイムホイル法は、従来の逐次的なシミュレーションで最も一般的なものである。

タイムホイルは既に述べた、タイムスタンプソート戦略によるスケジューラとほぼ同じものと考えて良い。ただし、タイムホイルでは、クロックは時刻の増大方向にのみ進む点が異なる。これは、時刻を集中管理しているため、タイムホイルのクロックよりも若い時刻のタイムスタンプ値を持ったメッセージが到着することがないためである。

5.2.2 並列化方法

タイムホイル方式の並列化方法を簡単に述べる。

各 PE には一つのタイムホイルと、分割された部分回路が割り当てられる。タイムホイルは各々の部分回路に対応したメッセージを管理する。そして、すべてのタイムホイルは同期してクロックを進める。

表 8: タイムホイル法 (回路 s13207)

PE 数	性能 (イベント / 秒)
1	2,261
2	4,210
4	8,307
8	13,756
16	19,748
32	18,493
64	11,320

5.2.3 計測結果および考察

実験の対象回路として s13207 を用いた。負荷分散方法はやはり第 4節の方法を用いた。

計測結果を表 8に示す。

1PE 使用時の処理性能は 2,261 イベント / 秒であり、バーチャルタイム法に比べて高速である。これは、履歴保存処理及びロールバックへの対処が不要になるためである。

複数 PE を使用した場合の性能については、8PE 以下の場合、バーチャルタイム法の場合よりも高い性能を示している。そして、16PE 使用時で 19,748 イベント / 秒という最高値を示している。しかしながら、32PE 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 : タイムホイルのクロックを同期して進めるためのオーバヘッドが大きいこと、2 : 使用 PE 全てを効率的に稼働させるだけの十分な並列性がないこと の二つが考えられる。

タイムホイル法は、使用 PE 数が少なく、且つシミュレーション時間を粗く離散化する場合に限れば、比較的良好な性能を示すと考えられる。

6 おわりに

バーチャルタイム法による論理シミュレーションシステムを構築し、性能評価を行った。その結果、問題に十分な並列性があるものについては、64PE 使用時に約 60k イベント / 秒という性能および約 48 倍の速度向上を得た。これらの値は、本システムがノンユニット遅延モデルに基づくソフトウェアシミュレータとして高性能なものであることを示している。

一方、十分な速度向上及び性能を得ることができなかったものについては、問題に十分な並列性がない、或いは、静的負荷分散により、十分な並列性を引き出せないことが主な原因であると考えられる。

さらに、ヌルメッセージを用いたコンサーバティブ法、及び、タイムホイル法による並列論理シミュレーションの実験も行い、バーチャルタイムとの性能比較を行った。コンサーバティブ法は、速度向上は良いものの、ヌルメッセージの発生量が非常に大きく、絶対性能の面からはバーチャルタイムに遥かに劣るものであった。また、タイムホイル法は、使用 PE 数が少ない場合にはバーチャルタイム方式を凌ぐ性能を示したが、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、バーチャルタイム法は、並列論理シミュレーションの方法としては最も現実的な方法であることが確認できた。

今後は、静的負荷分散による並列性抽出が不十分な場合に対処するため、動的負荷分散導入の検討を予定している。また、実際の LSI 設計データを用いたシミュレーションを行い、性能評価をする予定である。最終的には、本システムを、Multi-PSI の約 20 倍の総合性能を持つ並列推論マシン PIM 上に移行する予定である。

参考文献

[1] D.R.Jefferson, "Virtual Time", ACM Transactions on Programming Languages and Systems,Vol.7,N0.3, 1985, pp. 404-425

[2] J.Misra, "Distributed Discrete-Event Simulation", ACM Computing Surveys,Vol.18,No.1, 1986, pp. 39-64

[3] K. Taki, "The parallel software research and development tool: Multi-PSI system", Programming of Future Generaiton Computers, North-Holland, 1988, pp. 411-426

[4] L. Soule', T. Blank, "Parallel Logic Simulation on General Purpose Machines", Proceedings of 25th Design Automation Conference, 1988, pp. 166-170

[5] 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, 1989, pp. 23-30

[6] 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, 1989, pp. 1547-1554

[7] R.M.Fujimoto, "Parallel Discrete Event Simulation", Communications of the ACM, Vol.33, No.10, 1990, pp. 30-53

[8] K. Ueda, T. Chikayama, "Design of the Kernel Language for the Parallel Inference Machine", The Computer Journal, Vol.33, No.6, 1990, pp. 494-500

電子回路レイアウトシステムｃｏ－ＨＬＥＸの開発状況

渡辺俊典　小松啓子
（株）日立製作所　システム開発研究所

1　緒言

大規模計算量を要する電子回路レイアウト問題を対象として、新たな並列協調算法の可能性、当算法への並列論理言語やマシンの適用性、並列推論マシンの計算力等の実証研究を行っている。レイアウトＣＡＤは、ＬＳＩの発展を支える基礎の一つであるが、そのソフト開発は未だに膨大な人的労力を要している。この問題の解決への一つの糸口を探求する目的で、並列論理言語による回路レイアウト問題解決ソフトウエアの出来るだけ簡潔な記述を与えることも試みている。

ICOT後期３年間での研究計画を下記の様に設定している。

初年度：電子回路を対象とした並列論理型言語によるレイアウト問題解決システム第１版開発
２年度：第１版を機能、性能両面で改良した第２版開発
３年度：第２版までに開発した各種機能を統合した第３版の開発と評価、研究の総合的まとめ。

以下、co-HLEX[*1]第２版のにつき概要を紹介する。

2　co-HLEXの基本概念

２．１　レイアウト問題と状態表現

co-HLEXが処理するレイアウト問題は、次のゴール節で定義できる。

:-mode solve_a_layoutproblem(+,-,+,+).
?-solve_a_layoutproblem(CirNet,LPlan,Proc,Constr).

ここに、

CirNet::=回路モジュールとモジュール間の配線ネットデータ。
LPlan::=[PQtree,Wires].（図2-1）。
PQtree::=長方形状区画の再帰的分割構造（スライス）を現わす四分木。木の各ノードは、配置モジュール名称、北、西、南、東辺上の配線コネクタ名称を持つ。Samet[1,2]，Otten[3]，Luk[4]らによるQtree（Quadratic Tree：四分木）の拡張概念である。
Wires::=[Conns,Lines].
Conns::=配線端子、Viaと呼ばれる配線層間の貫通穴、および配線が長方形状区画の周辺境界を横切る点に敷設する仮端子等のリスト。
Lines::=２つのコネクタを結ぶ配線のデータ。両端コネクタ名称、配線幅等からなる。
Proc::=ＬＳＩ製造プロセス名称。モジュール形状や、配線幅などに影響する。
Constr::=モジュール間近接条件、及び計画レイアウト（目標外形と外部配線端子目標位置）。

２．２　制約付き階層レイアウト問題の再帰解法

スライス構造表現によって、レイアウト問題解決アルゴリズムを以下の簡潔な再帰形式で実現する可能性が生じる。

（１）目標外形、外部配線端子目標位置、及びレイアウト対象回路ネットから成る原問題を、高々４個の、親と相同構造の部分問題に分割する。
（２）分割した部分問題を、各々並列にレイアウトする。
（３）得られた部分問題の解を（１）での計画位置に設置して統合し、親問題の解とする。

２．３　課題

（１）問題分割の複雑さ

回路ネットCirNetの分割はネットのみでなく、チップの計画形状の分割をも考慮したものとする必要があり、処理内容も計算量も複雑となる。

（２）部分問題間の干渉

部分問題相互間で、ブロックの対向外辺長や配線端子位置の整合を取らないと、チップ上の空きエリアや配線の屈曲を招いてしまう。これらはチップ面積の増加につながる。

２．４　解決策

（１）回路ネットデータの事前階層化

与えられたCirNetを四分木に変換する事前処理を設ける。計画レイアウト形状を考慮した四分木生成を行うと共に、近くに配置するモジュールを四分木の下位階層にまとめることによって、co-HLEXの処理結果においても近接性を満足させうるようにする。

（２）モジュール形状の縦方向管理による相互整合化

親問題のチップ目標外形をいくつかのスライスに分割し、各々に子回路をうめ込み、各子供を再帰的にレイアウトする時、各子供が親から与えられた目標形状を出来るだけ遵守することにすれば、各子供を互いに並列に処理しても最後に得られる親のレイアウトは、予定に近いものになると期待できる。類似の機構は、社会組織における予算管理機構にみられる。

（３）配線における横方向の協調

異なるブロック間での共通配線の引き出し口の自動整合問題は、コネクタ整合問題としてＬＳＩレイアウトの有名な難問である。並列走行プロセス間で引き出し口位置設定をめぐる協調を行わせることによって、この問題を解決する。配線引き出し点に設けた仮端子を協調処理におけるプロセス間通信の手段とする。たとえとしては、プロセス間通信用メール箱として利用すると言ってもよい。各々の仮端子は、任意の時点での配置予定範囲と、自分にアクセスした隣接プロセス名などを記憶している。各親ブロックで問題分割を行うとき、これらの仮端子情報と自分の内部にうめ込んだ子回路のネットとの接続要否を参照しながら仮端子の配置予定範囲を狭めることを試みる。自分に範囲を狭める資格が有るか無いかを考慮し、無い場合には隣接ブロックに決定機会を譲る。

3　並列推論マシン上の回路レイアウトシステムｃｏ－ＨＬＥＸ

３．１　co-HLEXの概要（図３-１）

（１）問題解決基本機能

回路の四分木表現から、与えられたチップ目標形状と目標外部端子のもとでレイアウト四分木表現を生成する。問題解決基本機能はこの生成処理における推論規則に対応するものであり、階層再帰並列協調算法ＨＲＣＴＬ[*2]として実現している。ＨＲＣＴＬの概要は以下の通り。

（１．１）問題分割　平面を分割し、一つの親問題から複数の子問題を作成する。一つの問題は、（レイアウト対象回路木、ブロック目標形状、目標外部端子）の組である。回路木の末端では、セルを取り込む。

（１．２）再帰処理　各子問題を各々並列走行するプロセスとし、レイアウトの再帰処理を行なう。並列プロセス間で横方向の通信を行ない、再帰処理のもたらす縦割り問題解決の弊害を解消する。このことは、特に配線操作に於ける配線引込位置の決定や、配線制約の遵守のために重要となる。

（１．３）問題統合　レイアウトされた各子問題を、問題分割時の計画に沿って統合し、親問題の解を形成する。

（２）レイアウト基本機能

レイアウト基本機能は、生成処理における公理に対応するものであり、問題解決基本機能によって適切なものが選択され使用される。

（2．1）分割統合機能　親問題を子問題に分割したり、解かれた子問題を統合するための平面分割テンプレート（レイアウト問題解決用のタイプ）である。分割数として2、3、4を許容し、さらに内部の分割境界線にも多様さを与えている。また、各テンプレートに内部ブロック間の配線パターンを定義しており、子回路の数、面積、アスペクト比、等の多様さに対応可能としている。

（2．2）形状テンプレート　基本素子や、配線コネクタ、配線等の形状を定義したもの。

（2．3）プロセス制約　配線間隔、配線可能層等を定義したもの。

（3）入出力機能

（3．1）回路木生成機能　原データとして与えられる回路ネットを分析、階層化してレイアウトシステムへの入力データである回路木を作成する。処理概要を以下に示す（図3-2）。

（3．1．1）回路内素子位置の分析

チップの計画レイアウトデータの一部である北、西、南、東辺上の外端子（信号入力点、出力点、高位電源点、低位電源点等）から見た回路内素子位置指標を作成する。現状ではグラフのパス探索法を用いている。

（3．1．2）回路分割

素子位置指標から求めた回路の縦横比とレイアウト目標外形状の縦横比とを基に分割テンプレートを選択し、これを用いて分割を行う。結果として、各分割ブロックへの各素子の帰属、各分割ブロックの目標形状が得られる。

（3．1．3）外端子設定

各分割ブロック間を接続するネットを分析し、ブロック外端子を設定する。

（3．1．4）再帰

各分割ブロックについて分割を再帰的に行う。

（3．2）回路図復元機能　レイアウト結果の結線チェックのために、レイアウト結果から回路図を復元描画する。レイアウト結果得られた素子位置と素子間相互配線とを描画する際の形状テンプレートを回路図復元用のものにスワップすることで実現している。

（3．3）レイアウト評価機能　レイアウト結果の電気特性を分析する。

（4）メモリーアーキテクチャと負荷分散法

（4．1）分散メモリー　問題解決時に回路木、レイアウト木の各ノードをＰＥ格子上に分散することによって分散メモリーを実現した（実際には、ＫＬ１の述語のサイズ制限から、レイアウト木の各ノードプロセスが使用するデータをデータプロセスとして定義し、ノードプロセスと同じＰＥに割り当てている）。

（4．2）負荷分散　”回路をレイアウトするための総計算量に見合った計算力を与える”という原則で負荷分散を行なっている。子回路に含まれる部品量や推定面積を総計算量の目安として使用し、問題分割時に親ノードでＰＥ集合を各子回路の総計算量に応じて分配する。

3．2　開発現況

昨年度のco-HLEX第１版に対して、主として以下の改良を実施した。

（1）システム機能の充実

原回路ネットから回路木を自動的に作成する機能を実現すると共に、異形状ブロック配置、素子間アイソレーション確保、配線層考慮、配線制約条件考慮機能などを部分的に実現した。

（2）システム性能の向上（図3-3）

第１版で採用していた分散プロセス集中メモリー方式を、分散プロセス分散メモリー方式に改良することによって計算速度をO（指数）－>O（$N^{0.8}$）と大幅に改善することが出来た。ここでNは、シ

ンボル数である。さらに、目標としたO（１０００）シンボルのレイアウトを実現した。
通信ロスを無視したときの計算速度理論値は次のようになる。

O（log（PE）＋N／PE）.

ここに、PEはプロセッサ総数である。

4　レイアウト実験

co-HLEX第２版による回路レイアウト過程を示す。

（１）レイアウト対象回路図：（図４-１）.
（２）レイアウト前処理（回路図の自動階層化）：（図４-２）.
（３）レイアウト処理：（図４-３）.
（４）回路図復元処理：（図略）.
（５）大規模データのレイアウト（模擬データ使用）：（図４-４）.

5　おわりに

co-HLEX第２版では第１版に対して大幅な性能向上を実現した。これには、分散メモリーによる寄与が大きい。第１版アーキテクチャは集中メモリー分散プロセスであった。本年度の経験からこれでは本来の並列処理は無理であり、分散メモリーが並列処理の要であると感じている。レイアウト機能の今後の充実によって並列プロセス間の通信ロードが増し、応分の性能劣化があるとは予想しているが、それらへの対応を含め、個別に開発してきたサブシステムの機能、性能改良をさらに行った上で、co-HLEX第３版として統合し、システムの総合評価を行うことが今後の課題である。

機能実証性を高めるために現実的な回路に対するレイアウトに取り組んでいるが、現下の開発環境ではプログラムの虫取りは逐次処理に比べて極めて困難である。現実的システムを開発するためには、虫取り環境の整備が極めて重要であり、これ無くしては多くの魅力を持った並列処理も実用化は困難であろうと感じている。並列ハード、ＯＳ、言語が一揃いしつつある現在、次の課題はこの問題では無いかと感じている。

＊１　co-HLEX：CO-operative Hierarchical Layout EXpert
＊２　HRCTL：Hierarchical Recursive Concurrent Theorem Prover for Layout

謝辞

研究推進に当たり日頃ご支援いただく、ＩＣＯＴ新田克巳第７研究室長、市吉伸行室長代理、瀧和男第１研究室長を始め、ＰＩＣ、ＫＳＡワーキンググループの方々、及び研究機会を与えていただいている（株）日立製作所システム開発研究所、堂免信義所長に深謝します。

参考文献

[1] Samet, H., Region Representation: Quadtrees from Boundary Codes, C.ACM, Vol.23, No.3, pp163-170, March, 1980.

[2] Samet, H., The Quadtree and Related Hierarchical Data Structures, Computing Survey, Vol.16, No.2, pp187-260, June, 1984.

[3] Otten, R, Automatic Floorplan Design, Proc. 19th DAC, pp261-267, 1982.

[4] Luk, W.K., Tang, D.T., and Wong, C.K., Hierarchical Global Wiring for Custom Chip Design, Proc. 23rd DAC, pp481-489, 1986.

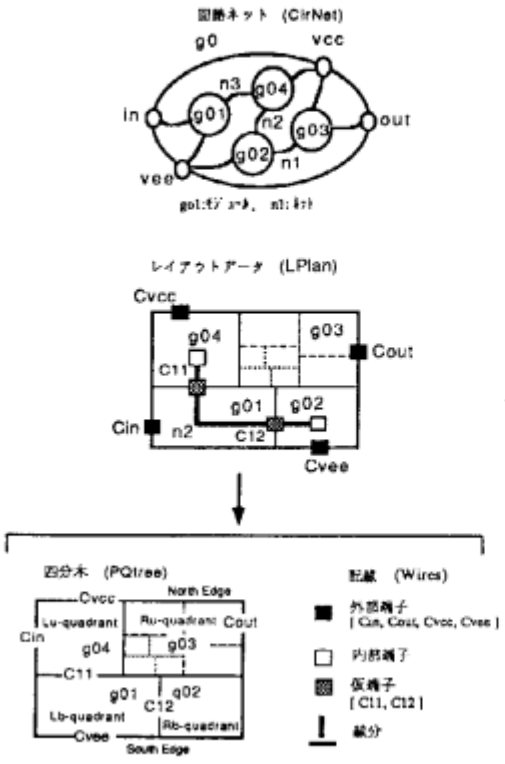


図 2-1 レイアウト状態の4分木表現

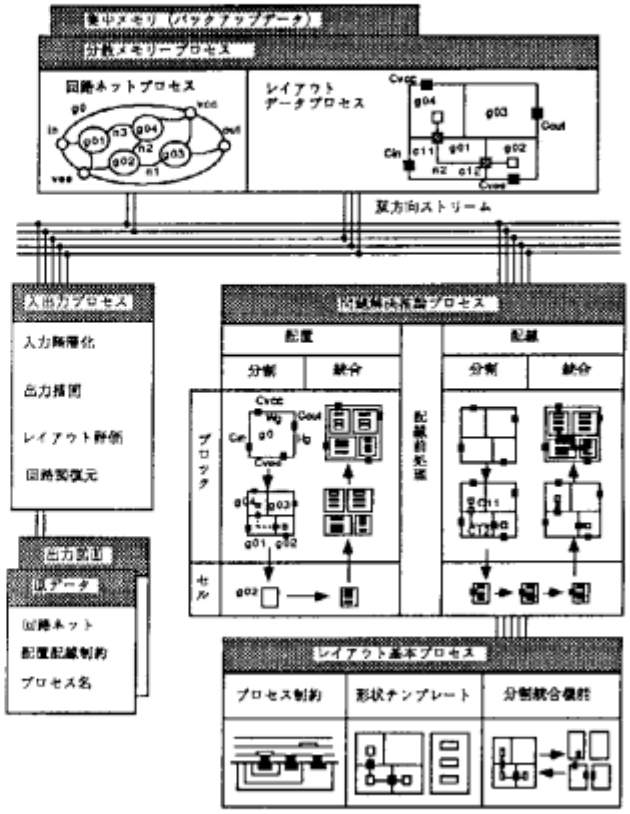


図 3-1 並列レイアウトシステム（ｃｏ－ＨＬＥＸ）の構成

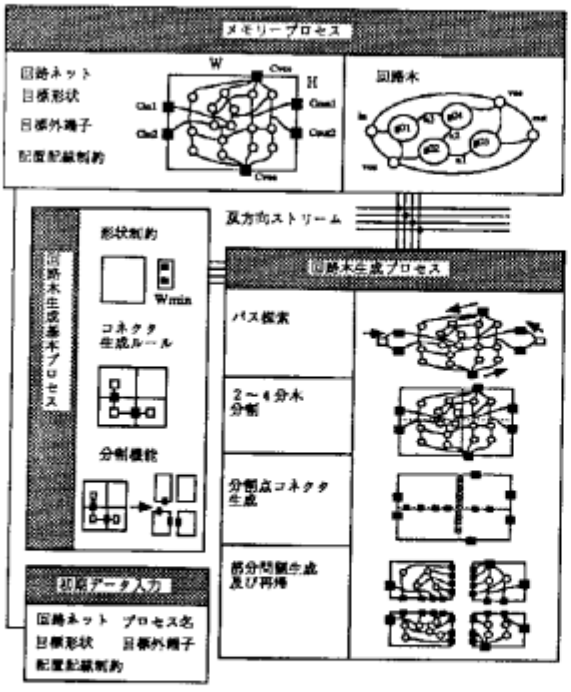


図 3-2 回路木生成機能の概要

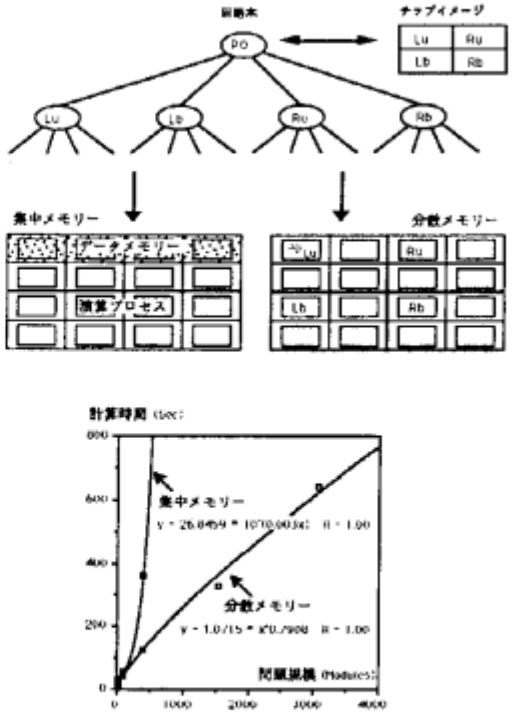


図 3-3 co-HLEXのメモリー構成と性能

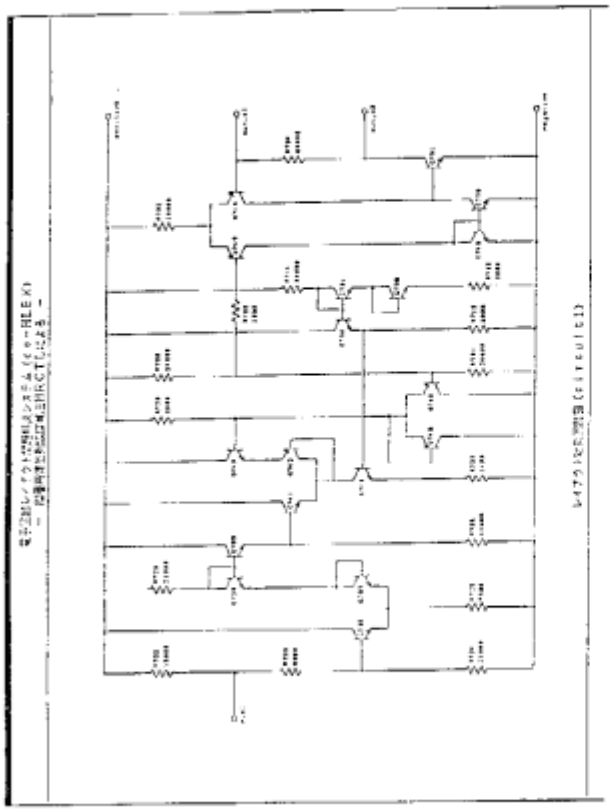


図4-1　入力回路図

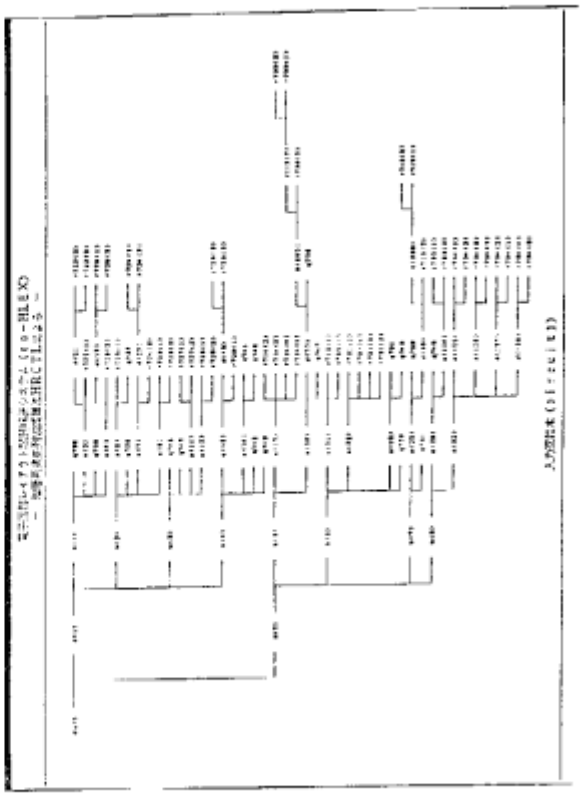


図4-2　階層回路木

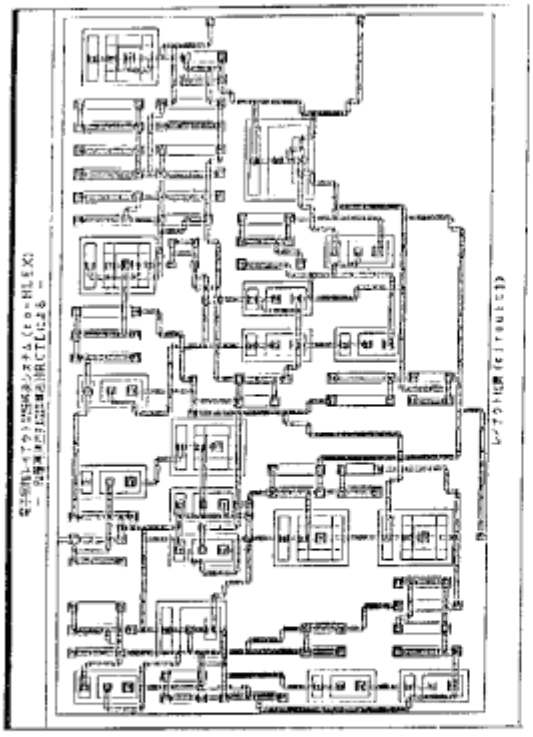


図4-3　レイアウト結果

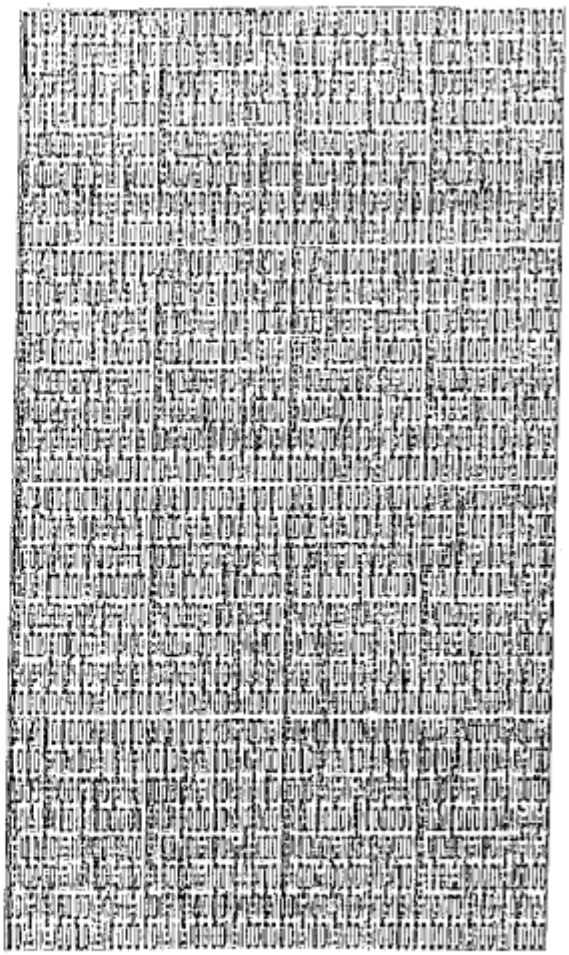


図4-4　1536素子のレイアウト

並列オブジェクトモデルに基づくLSI配線プログラム *

伊達 博　　大嶽 能久　　瀧 和男 †
(財) 新世代コンピュータ技術開発機構 ‡

E-mail: date@icot.or.jp

概要. LSI配線は、LSI CADの処理の中では、多大な計算時間を必要とする問題として知られており、並列処理による速度向上は、設計期間の短縮に効果が大きいと期待されている。本論文では、分散メモリ型の大規模な並列マシンに適用できることを目標に、並列オブジェクト指向モデルに基づいた新しい並列配線手法を提案する。この手法は、配線領域における線分のすべてをオブジェクトに対応させ、それらがメッセージを交換しながら与えられた端子間の経路を探索していくものであり、高い並列性を内在しうる。探索の基本アルゴリズムとしては、予測線分探索法を並列化して用い、それをKL1言語を用いて分散メモリ型並列マシンのMulti-PSI上に実装した。LSIの実データを用いて性能評価を開始しており、その初期結果を報告する。

1 はじめに

LSIのチップ表面に配置された多数の素子間を結線するために、その配線経路を決めることを、「LSI配線設計」と呼ぶ。LSI配線設計は、LSI設計の各工程の中でも、計算機による自動化の進んでいるものの一つである。また、論理シミュレーションやテスト生成と並んで、多量の計算を必要とする応用問題としてもよく知られている。

LSIの集積度は、およそ2年で2倍の割合で、着実に増加をつづけている。その一方で、配線設計に要する計算時間は、素子数に比例するのではなく、多くの場合、その2乗、あるいはそれ以上の割合で増加する。このことは、LSIの集積度の向上にともなって、その設計時間が急激に増大することを意味しており、高速度の自動配線設計システムの実現が強く望まれる理由となっている。

また素子数の増大により、自動設計で未結線が出た場合の人手介入が、しだいに難しくなっており、未結線を出さない高品質の自動配線設計への要求も高まっている。

我々は、並列処理を用いてこれらの問題を解決すべく研究を続けているが、本稿では、配線設計の高速化を目的とした、新しい並列配線方式を提案するとともに、それを分散メモリ型並列マシン、Multi-PSI上に実装し、初期評価を行った結果を報告する。

以下では、過去の並列配線の事例と本方式の特徴、基本アルゴリズム、並列オブジェクトモデルとKL1言語によるプログラミング、プログラム設計、負荷の割り付け、測定と初期評価、考察、の順に述べる。

2 過去の並列配線の事例と本方式の特徴

配線設計の並列処理では、特定のアルゴリズムを実行する特別のハードウェアを設計・試作した例[3, 5, 8]と、汎用的に使用可能な並列マシンの上に、ソフトウェアで並列配線プログラムを実現した例[1, 6, 7, 11, 12]などがある。前者は、処理性能がきわめて高いかわりに、処理の柔軟性に乏しく、後者は、アルゴリズムの変更に対処しやすい反面、処理性能は前者に劣るという傾向がある。今後高い処理性能が実現できたあとは、配線品質の向上に向かいたいことから、今回は、処理の柔軟性を重視し、後者のアプローチをとった。

また後者の中では、SIMD型の並列マシンを用いた例[11]と、MIMD型を用いた例[1, 6, 7, 12]がある。今回の提案は、(1) MIMD型並列マシンは、SIMD型よりも適用領域が広く、将来の汎用並列マシンに成り得ると考え、将来のMIMD型汎用大規模並列マシンに適用できるような並列配線処理方式の実現を目指したこと、(2) 大規模なMIMD型並列マシンに適用できるためには、問題が、大きな並列性を内在する必要があるため、計算の粒度の小さい、並列オブジェクトモデルに基づく新しい配線問題のモデル化を試みたこと、(3) その上で、逐次アルゴリズムの中でも、処理効率と配線品質の両方に優れたアルゴリズムを選んで、それを分散アルゴリズムに設計しなおしたこと、などの特徴を有する。

---

*A Parallel Router based on a Concurrent Object-oriented Model

†Hiroshi DATE, Yoshihisa OHTAKE, Kazuo TAKI
‡Institute for New Generation Computer Technology

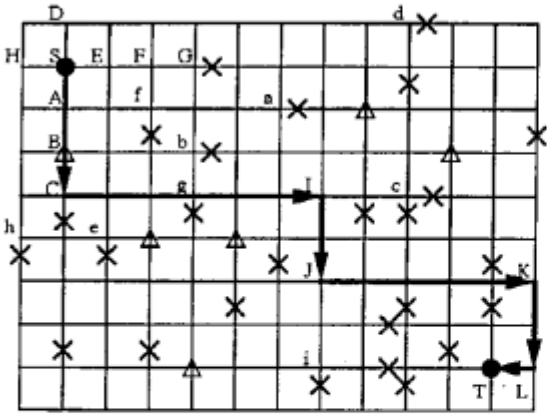


図 1: 先読み線分探索法

具体的には、高い並列性を実現するため、未配線、既配線を含めたすべての配線格子 (LSI 配線では、通常あらかじめ定めた格子上でのみ配線が許される) 上の線分を互いに能動的に動作しうるオブジェクトとしてモデル化した。この場合、配線アルゴリズムは、必然的にオブジェクトが互いにメッセージ交換しながら良い配線経路を決定してゆく分散アルゴリズムとなる。

また線分をオブジェクトにすることとの親和性から、基本アルゴリズムは、計算効率の高い線分探索法がよいと考えた。そしてその中でも迷路法のように配線経路が存在するときの結線可能性を保証している予測線分探索法 [4] を選んだ。予測線分探索法は、逐次アルゴリズムであるため、これを、互いに交差する線分オブジェクトが通信しながら短い配線経路を決定してゆく分散アルゴリズムに設計しなおした。

3　基本アルゴリズム

基本アルゴリズムとして採用した予測線分探索法 [4] について簡単に説明しておく。

この予測線分探索法とは、線分探索に”先読み”を加えた手法を基本として、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た手法である。本手法では 2 層配線を対象とし、縦方向・横方向で配線層を使い分け、配線は全てあらかじめ固定された配線格子上を通るものとする。更に配線の通過禁止、スルーホール禁止（線の折り曲げの禁止）などの制約も扱う。ただしこれは逐次アルゴリズムである。図 1を例としてその基本処理を示す。Sから探索を開始してTへ接続する場合、Sから下方に向かった線分がA点で折り曲げられたとすると、到達できるTに最も近い点はaである。同様にして、C, Dで折り曲げられたとき到達できる点はc, dとなる。これらの点（a, c, d）をSからの“期待位置”と呼ぶ。B点はスルーホール禁止であり、ここで折り曲げることはできないため、b点は期待位置としない。水平方向に進んだ場合も同様にして期待位置（e, f, g, h）が得られる。これら期待位置（a, c, d, e, f, g, h）の中で最もTに近い点はcである。これより、S－C間を接続する。次にCから水平方向に探索を行ない、Tに最も近い期待位置を求めるとiが得られる。これより、C－I間を接続する。なお、この場合は、I点は前回の期待位置cとは異なっている。同様に探索を続けることによりS－T間を接続する一つの経路S－C－I－J－K－L－Tが求められる。

4　並列オブジェクトモデルと KL1

基本アルゴリズムを並列化するにあたって、高い並列性を実現すること、およびプログラミング言語との親和性から、問題を互いに通信しあいながら並列に処理を進めて行く基本要素の組み合わせとしてモデル化した。これはすなわち、並列オブジェクトに基づくモデル化であり、問題から並列性を引き出そうとする時に有効な方法の一つである。

KL1[2, 10] では、このようにモデル化された問題を極めて自然に記述できる。即ち、オブジェクトをプロセスと呼ばれる処理の基本要素とし、それらのプロセスがメッセージストリームを用いて通信し合うように実現することが容易にできる。

例えば図 2に KL1 による並列オブジェクトの実現イメージの一例を示す。並列オブジェクトはそれに対応する述語を、それが受け取るメッセージに対応する一連の節によって定義する。メッセージ列はストリームと呼ばれるリストによって表現される。リストの car 部分が最初のメッセージ、cdr 部分がその後に届くメッセージ列を表す。逆にメッセージを送出するには、それを送り付けたいオブジェクトのメッセージストリームを指す変数に、「 car 部分がこれから送るメッセージ、cdr 部分が後に送るメッセージ列を入れるための変数」からなるリストをユニファイすればよい。

5　並列プログラミング

線分をオブジェクトとみなす実行モデルに基づき、このような線分のオブジェクトをプロセスとして実

```
concurrent_object ( [ Message_1 | Rest ], 内部状態変数, ストリーム変数 ) : -
  true |
    Message_1に対応する処理,
    内部状態変数の更新,
    ストリーム変数 = [ メッセージ | 新しいストリーム変数 ],          %他のオブジェクトへのメッセージ送出
    concurrent_object ( Rest , 新しい内部状態変数, 新しいストリーム変数 ).

concurrent_object ( [ Message_2 | Rest ], 内部状態変数, ストリーム変数 ) : -
                                    ⋮
```

図 2: KL1 による並列オブジェクトの実現例

現する。以下では、オブジェクト＝プロセスとして、プログラムの説明を行う。

配線格子の各格子に対応するプロセスをマスターライン・プロセス、線分に対応するプロセスをライン・プロセスと呼ぶ。これらのライン・プロセスが互いに通信しながら、それぞれの状態に応じた処理を行なうことによって、探索・配線処理が進められる。

本プログラムでは、２通りの並列性を実現している。第１は、１対の端子間を配線するアルゴリズムの中で、予測線分探索法の先読み部分を並列化している。第２は、複数の配線処理を同時並行に実行することによる並列性である。個々の線分プロセスは並列に動作することができるので、探索・配線処理を複数ネットについて、同時に並行して進めることができる。

処理途中の配線状況は各線分プロセスの内部状態として表現される。配線過程では未配線領域が既配線領域と残りの未配線領域に分割されたり、バックトラックによって既配線領域が未配線領域に戻されたりする。それに対応してライン・プロセスは動的に分裂／結合する。すなわち新たな配線の際には、一つの未配線領域のライン・プロセスが、既配線領域と残りの未配線領域のライン・プロセスにそれぞれ分割される。一方バックトラックを行なう場合には、既配線領域を再び未配線の状態に戻さなければならないので、配線の際分割されたライン・プロセスは一つの未配線領域のライン・プロセスとして再結合される。このようにライン・プロセスは探索過程で動的に分裂／結合が行なわれる。

先述の基本アルゴリズムの説明から分かるように、期待位置を求めるためにはライン・プロセスは自分と直交する線分のライン・プロセスと通信しなければならない。図3に示すようにマスターライン・プロセスは互いに直交する線分のライン・プロセス間の通信を仲介し、ライン・プロセスが動的に変化しても、正しく通信が行なわれるようにメッセージの送

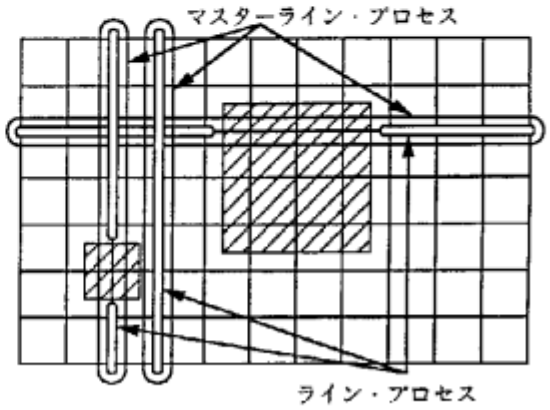


図 3: マスターライン・プロセスとライン・プロセス

り先を管理する。

以下に１ネット探索での先読み処理の並列化の様子を示す。Ｓをスタート、Ｔをターゲットとする。Ｓから垂直方向に期待位置を求める場合、図4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、各領域中でターゲットに最も近い位置、即ち期待位置の計算を依頼する。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はいる。

依頼されたライン・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・プロセスは回答が全て集まった時点で、その中から最もターゲット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5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。

それ以降の探索は図6に示すように、先の最もターゲットに近い期待位置を回答してきたライン・プロセスに制御が移され、同様に進められる。ターゲッ

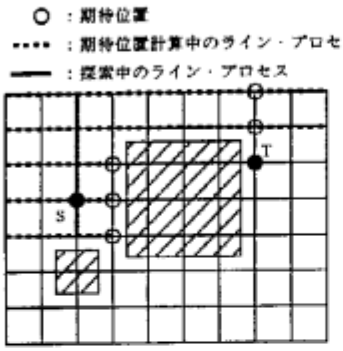


図 4: 期待値の並列計算

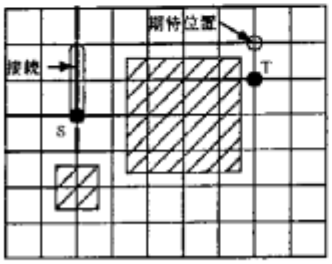


図 5: 経路接続

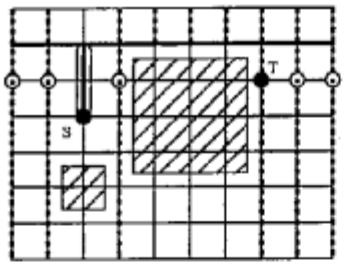


図 6: 並列期待値計算の終了条件

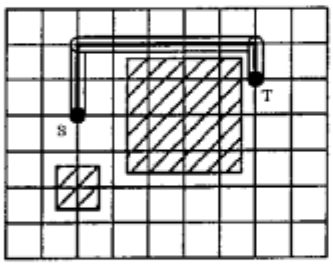


図 7: 配線完了図

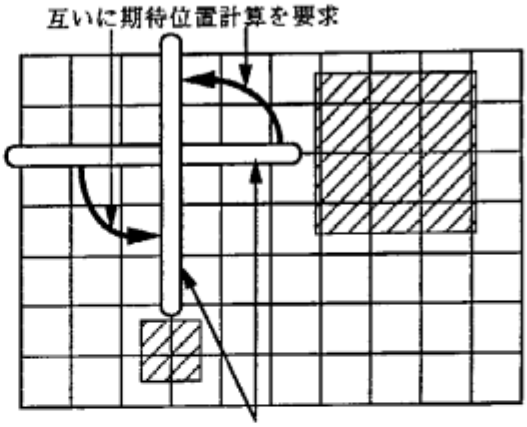


図 8: デッドロックの発生例

ト＝期待位置となるライン・プロセスまで制御が移れば図 7に示すように 1 ネットの配線が終了する。

複数のネットの配線を同時に行おうとする場合、複数のネットが同じ配線領域を取り合うような状況が発生し得る。このような場合、従来行われていた手法として、配線領域をビットマップで表現し、共有データとして刻々の配線状況を管理するものがある。この方法を用いる場合、上記のような競合時にデータの矛盾を発生させないようにするためには、データをある処理単位の間ロックする必要が生じる。しかしながら、共有データのロックは、しばしば並列性を阻害する要因となる。本手法では、すべての配線領域 (線分) を独立のオブジェクトとし、個々のオブジェクトはプロセスによって実現していることから、基本的に一時に 1 個のメッセージしか処理しない。またメッセージは、プロセスへの入力ストリーム上で直列化 (serialize) される。これによりオブジェクトに対する排他制御は、プログラムレベルでの何ら特別な記述を必要とせず、きわめて自然にかつ美しく実現できる。

6 デッドロックの回避

上記のような 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、全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。

例えば図 8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、先の並列化アルゴリズム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全て返って来るのを待っ

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出来ない。従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。

一方、一つのメッセージに対する処理を必ず一定時間内に終了するとすることが保証されるならば、このようなデッドロックは起こらない。そこで、オブジェクト内でのメッセージ処理方式をこの条件を満たすように修正する。

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後から到着してくるメッセージをすべて監視し、その中に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に対する処理を行い応答を返すことにする。これにより上記デッドロック回避の条件を満足する。

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。

7 複数ネット間の競合

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場合が生じることがある。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われる。後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、後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、使われずに残る場合がある。それが配線品質の低下（遠回り）や配線率の低下を招く場合がある。

このような相互干渉が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ことによって、配線率を確保するなどの対処方法が考えられるが、処理の並列性を落とす可能性がある。

8 マッピング

KL1プログラムを現在のターゲットマシンであるMulti-PSI[9]上で効率良く実行させるためには、（1）アルゴリズムの並列性（2）負荷の均等化（3）通信の局所化の三つについて十分に検討しなければならない。この負荷の均等化と通信の局所化を決定付けるのがマッピング方式である。

マッピングとは、各プロセッサへの処理の分配のことである。マッピングでは各プロセッサの負荷の均等をまず考えるが、本試作プログラムのターゲットマシンであるMulti-PSIやPIMのような分散メモリ型並列計算機では、プロセッサ間の通信コストが比較的大きいため、プロセッサ間通信を極力抑えることも考慮しなければならない。

先述の並列化アルゴリズムの説明からも分かるように、本アルゴリズムは、配線・探索に必要なメッセージ通信量が多く、それらはプロセッサ間通信となる可能性がある。現在は初期評価のために、負荷の均等化を主に考えたマッピングを行なっており、通信の局所化については、マスターライブン・プロセスとその上のライン・プロセスを同一プロセッサに配置するだけに留めている。そしてそれらのプロセスをプロセッサにランダムに割り当てている。従って本プログラムは、プロセッサ間通信の抑制についてはまだ改良の余地が残されている。

9 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二種類の実データを用いて実験を行った。それらのデータの仕様を表1に示す。DATA1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、DATA2は、局所的に端子が集中しているデータである。

9.1 問題の規模と台数効果

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

表 1: データの仕様

データ名	DATA1	DATA2
格子規模	262×106	322×389
ネット数	136	71
IO端子	ネット無し	ネット有り
提供元	日立製作所	NTT
特徴	端子が全体に分散	端子が局所的に集中

表 2: 実行時間と配線性能 (DATA1)

データ規模	PE数	T (秒)	S	W (%)
1 × 1	1	123	1.1	99
	2	135	1.0	100
	4	92	1.5	96
	8	51	2.6	97
	16	32	4.2	97
	32	22	6.1	97
	64	18	7.5	94
1 × 2	1	289	1.1	99
	2	304	1.0	100
	4	243	1.3	98
	8	121	2.5	96
	16	69	4.4	96
	32	41	7.4	96
	64	33	9.2	96
2 × 2	1	N.A.	N.A.	N.A.
	2	829	1.0	100
	4	616	1.3	96
	8	301	2.8	96
	16	181	4.6	96
	32	98	8.5	96
	64	67	12.4	95
2 × 4	1	N.A.	N.A.	N.A.
	2	1769	1.0	99
	4	N.A.	N.A.	N.A.
	8	N.A.	N.A.	N.A.
	16	380	4.6	96
	32	197	9.0	96
	64	115	15.4	97

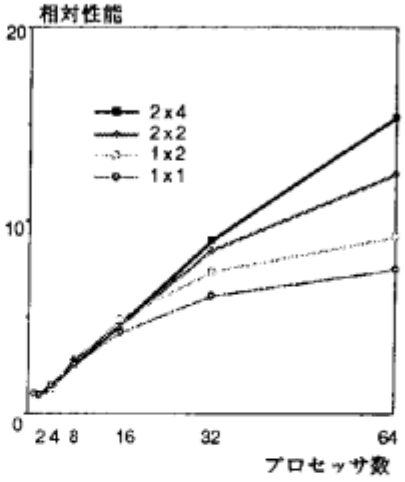


図 9: データ規模と相対性能

なると考えられる。そこで、我々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。用いたデータは、DATA1を縦方向にn個、横方向にm個コピーしたもので、ここでは、四つの場合（1 × 1, 1 × 2, 2 × 2, 2 × 4）に対して測定した。コピーしたデータに配線プログラムを適用する場合、その配線可能空間が増えるので、計算量も増えることとなる。その測定結果を表2に示す。表中、Sは相対性能を示しており、プロセッサ2台を用いた時の実行時間と比較して何倍速度が速くなったかを表している。またTは、実行時間、Wは、配線率である。これらの結果に対して、それぞれのデータに対する相対性能を一つのグラフとしてまとめたのが、図9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相対性能が向上することがわかる。2 × 4データに対して、2台のプロセッサを使用した場合と比較して、64台のプロセッサを用いると約15倍の速度向上が得られることがわかった。

9.2 配線率と並列度

前節の実験では、比較的端子が分散しているデータに対して配線率を余り落とさずにかなりの台数効果が得られることがわかった。しかし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、局所的に端子が集中しているデータであるDATA2に関して、64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との関係を測定したのが表3である。そしてこれらの結果を図10にまとめた。この図にお

表 3: 同時投入ネット数と配線率 (ＤＡＴＡ２)

N	W (%)	T (秒)
1	97	25
2	97	21
3	97	22
4	95	20
5	90	21
10	88	18
20	81	13
30	75	14
40	75	15
50	78	14
60	71	13
71	69	13

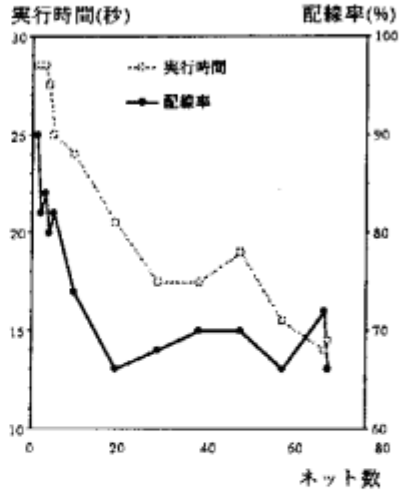


図 10: 配線率と並列度

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、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。また、このデータに関して、１台のプロセッサを用いたときの実行時間は、１４２秒であった。このことから、期待位置計算部の並列処理による効果は、約５倍であることがわかる。

9.3 汎用計算機との性能比較

ＮＴＴＬＳＩ研究所の北沢氏の協力により、IBM 3090/400 (15MIPS) 上に実装された予測線分探索法による配線プログラムを用いて、ＤＡＴＡ２を配線させた結果、7.45 秒で１００％の配線率であった。我々の配線プログラムと比較すると、現バージョンでは、約1.7 倍から３倍、我々のプログラムの方が遅いことがわかった。配線率の違いは、ＮＴＴのプログラムがアルゴリズムの細部に渡って、配線率向上のための努力をしていることによる。

10 まとめ

並列オブジェクト指向モデルに基づく新しい並列配線手法を提案するとともに、それを分散メモリ型マシンの Multi-PSI 上に実現した。LSI の実データを用いて性能の初期評価を行った。その結果、速度向上としては、２台のプロセッサを使用した場合と比較して、64 台のプロセッサを用いると約 15 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、更に高い効率が期待される。また、性質の異なる二種類のデータで評価した結果、端子の位置が分散しているデータでは、台数効果及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、期待位置の並列計算の効果はあるが、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。

今後、更に高速化するために、プロセッサ間通信を抑制するためのプロセスのプロセッサへのマッピング方法を検討して行く予定である。また、端子が局所的に集中しているデータに対して、同時に複数のネットを配線する時に生じる配線率の低下に対しては、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していく。また、大規模な LSI の実データに適用できるようプログラムの改良を行う予定である。

謝辞

本研究を行うにあたり、LSI のデータを提供して頂いた日立製作所中央研究所の白石洋一氏、データの提供と共に、有益な助言を頂いたＮＴＴ ＬＳＩ研究所の北沢仁志氏、また、活発に御討論いただいたＰＩＣワーキンググループの諸氏に深謝します。

A 配線出力結果

我々の配線プログラムをＤＡＴＡ１及びＤＡＴＡ２に適用した出力結果を次に示す。

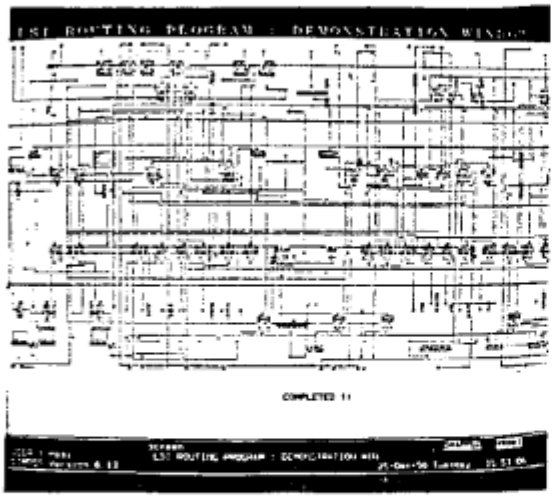


図 11: ＤＡＴＡ１の配線結果（一部分）

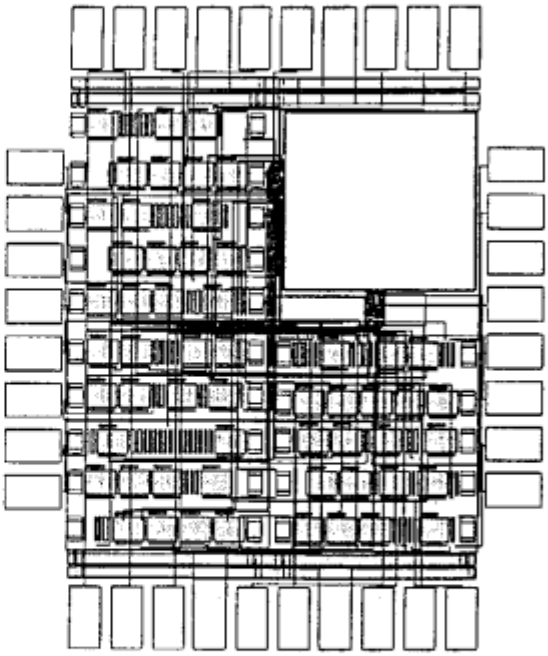


図 12: ＤＡＴＡ２の配線結果

参考文献

[1] R.J. Brouwer and P. Banerjee, "PHIGURE:A Parallel Hierarchical Global Router", *Proc. 27th Design Automation Conf.* , pp.650–653, 1990.

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

[3] K. Kawamura, T. Shindo, H. Miwatari and Y. Ohki, "Touch and Cross Router," *Proc. IEEE ICCAD90* , pp.56–59, 1990.

[4] 北沢 仁志, "高配線率線分探索の一手法" , 情報処理 , 第 26 巻第 11 号,pp.1366–1375, 1985.

[5] R. Nair, S. J. Hong, S. Liles and R. Villani, "Global Wiring on a Wire Routing Machine" , *Proc. 19th Design Automation Conf.* , pp.224–231, 1982.

[6] O.A. Olukotun and T.N. Mudge, "A Preliminary Investigation into Parallel Routing on a Hypercube Computer" , *Proc. 24th Design Automation Conf.* , pp.814–820, 1987.

[7] J. Rose, "Locusroute:A Parallel Global Router for Standard Cells", *Proc. 25th Design Automation Conf.* , pp.189–195, 1988.

[8] K. Suzuki, Y. Matsunaga, M. Tachibana and T. Ohtsuki, "A Hardware Maze Router with Application to Interactive Rip-up and Reroute" , *IEEE Trans. on CAD* , vol.CAD-5, no.4, pp.466–476, 1986.

[9] K. Taki, "The parallel software research and development tool: Multi-PSI system", *Programming of Future Genecraiton Computers*, North-Holland, pp. 411–426, 1988.

[10] K. Ueda, "Guarded Horn Clauses: A Parallel Logic Programming Language with the Concept of a Guard" , Technical Report TR-208, ICOT, 1986.

[11] T. Watanabe, H. Kitazawa, Y. Sugiyama, "A Parallel Adaptable Routing Algorithm and its Implementation on a Two-Dimensional Array Processor" , *IEEE Trans. on CAD* , vol.CAD-6, no.2, pp.241–250, 1987.

[12] Y. Won, S. Sahni and Y. El-Ziq, "A Hardware Accelerator for Maze Routing" , *Proc. 24th Design Automation Conf.* , pp.800–806, 1987.

スタック分割動的負荷分散方式とマルチPSI上での評価

古市 昌一、中島克人、中島浩
三菱電機株式会社
情報電子研究所

市吉 伸行
(財) 新世代コンピュータ技術開発機構

概要

スタック分割動的負荷分散方式は、深さ優先探索アルゴリズム等スタックを用いた問題の並列処理に適した負荷分散方式である。本論文では、スタック分割動的負荷分散方式のマルチPSI上での実現、並列深さ優先探索アルゴリズムに適用して計測した結果の評価、及びマルチレベル動的負荷分散方式との比較について述べる。

スタック分割動的負荷分散方式はミネソタ大学のV.Kumarらによって考案されたもので、各種のMIMD型並列計算機上に実現されて、良い台数効果が得られると同時に台数拡張性に優れていることが報告されている。本方式の特徴は、各プロセッサはスタックに仕事を保持しておき、仕事がなくなると他のプロセッサに仕事を要求する要求駆動による動的負荷分散を行なう点で、要求を受けたプロセッサは、自分のスタック中の仕事の一部を分け与えて負荷の均等化を行う。

本方式をICOTで開発した並列推論マシン・マルチPSI/V2上に実現し、詰込みパズルの全解探索問題に適用して評価を行なったところ、32台プロセッサで約27倍、64台で約46倍の台数効果が得られた。

1 はじめに

並列処理における最も重要な研究課題の一つが、負荷の均等化である。理想的には、負荷の均等化はシステムが自動的に行なうのが望ましいが、これは特に疎結合並列マシンにおいては難しく、現状では問題毎に最適な負荷分散方式をユーザが組み込む場合がほとんどである。

筆者らは、先にOR並列型問題に適したマルチレベル動的負荷分散方式(Multi-Level Dynamic Load Balancing : MLB方式) を提案し[2, 3]、マルチPSI上に実現して詰込みパズルの全解探索問題に適用してほぼ線形的な台数効果を得られた。MLB方式は、特定のプロセッサは互いに独立な仕事に分割する作業を行い、これを動的に検出した暇なプロセッサに対して割り付ける方式で、要求駆動による負荷分散を行なう[1]。ここで、単純な要求駆動方式ではプロセッサの台数が多くなると仕事を分割するプロセッサがボトルネックになるが、これを解消するために階層的に分割及び割り付けを行なうことによって高い台数拡張性を得たのがMLB方式である。

一方、スタック分割動的負荷分散方式はミネソタ大学のV. Kumarらによって考案された並列深さ優先探索アルゴリズム (Parallel Depth First Search : PDFS) [5, 6] の中で用いられている動的負荷分散方式で、各種のMIMD型並列計算機上に実現されて良い台数効果が得られると共に、台数拡張性に優れていることが報告されている。スタック分割動的負荷分散方式 (Stack Splitting Dynamic Load Balancing: STB方式) は、MLBと同様プロセッサが暇になると仕事を要求する要求駆動による負荷分散を行なうが、特定のプロセッサが仕事の分割及び供給を行うのではなく、全てのプロセッサが仕事を供給すると同時に仕事を実行する。従って、MLB方式では暇になった時に仕事を要求するプロセッサは決まっているが、STB方式ではある一定の戦略によって決めたプロセッサに対して仕事を要求し、得られなければ別のプロセッサに要求する。すなわち、STB方式では仕事を要求しても実際に仕事が得られるまでには時間がかかるという短所があるが、1レベルのMLB方式のように、供給がボトルネックになる事はない。

本論文では、STB方式をマルチPSI上に実現し、詰込みパズルの全解探索問題に適用して計測した結果と、MLB方式との違いについて述べる。

以下第2節で並列深さ優先探索アルゴリズムとスタック分割動的負荷分散方式について述べ、第3節でそのマルチPSI上への実現方式と詰込みパズルの全解探索問題への適用例について述べ、第4節で性能の計測とその結果の考察を行い、第5節ではSTB方式とMLB方式の比較を行う。最後に、第6節にてまとめを行なう。

2 スタック分割動的負荷分散方式

本節では、スタックを用いる代表的なアルゴリズムである並列深さ優先探索アルゴリズム (Parallel Depth First Search: PDFS) について述べた後、スタック分割動的負荷分散方式STBについて述べる。

深さ優先探索アルゴリズム (Depth First Search:

[1] 1レベルのMLB方式、あるいは単純な要求駆動方式と呼ぶ。なお、仕事を供給するプロセッサも暇になった時には仕事を実行する

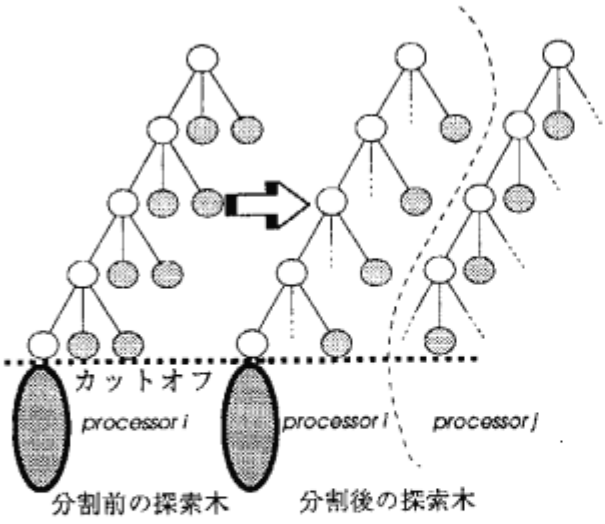


図 1: スタックの分割

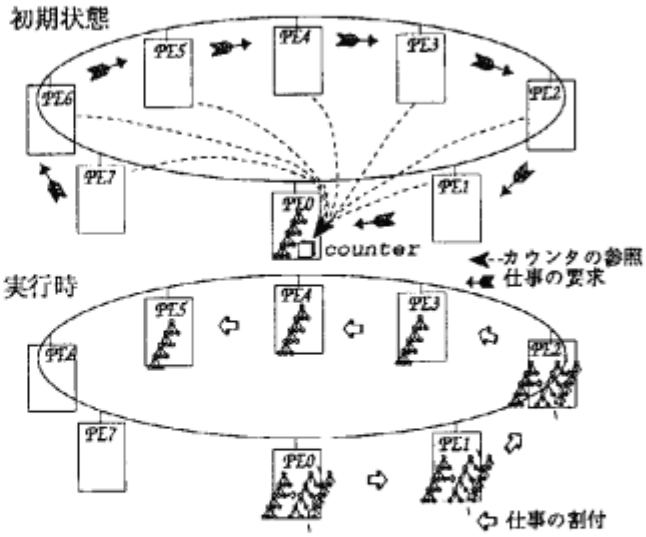


図 2: 分割したスタックの割り付け

DFS) では、探索木によって表現される問題を、まずルートノードよりその子ノードを生成することにより探索を開始し、次に子ノードの一つより更に孫ノードを生成し、この繰り返しをリーフノードに達するまで行なう。リーフノードに達しても解が見つからなかった場合には、一番近くのまだ探索されていないノードまでバックトラックして、そのノード以下を探索する。

DFS の並列化に際して、負荷分散は次のように要求駆動で行なう。暇になったプロセッサは、他のプロセッサに対して仕事を要求するメッセージを送信し、探索木の一部分を得る。この時、仕事を要求されたプロセッサが充分な仕事を持たない場合には、渡す仕事はないというメッセージを返す。この場合は、仕事を要求したプロセッサは別のプロセッサに対して再度要求メッセージを送信する。

V. Kumar らが行なった DFS の並列アルゴリズム PDFS は、ノードを保持したスタックによって探索木を表現し、仕事の分割はこのスタックを分割することにより行なう。以下、仕事の分割方式と仕事の要求先の決定方式について述べる。

2.1 仕事の分割方式

スタックは探索の各深さ毎に用意され、それぞれにはまだ実行していないノードが保持される。スタックを分割する際には、要求をしたプロセッサと要求を受けたプロセッサの 2 者間でなるべく仕事が均等になるように、探索の各深さ毎のスタックをそれぞれ半分に分割する。なお、リーフに近いノードを暇なプロセッサに割り付けた場合には、そのノードは直ちに終了してしまう。従って、ある一定の探索の深さにカットオフを設けて、その下のノードは分割しないようにした方が効率が良い。図 1 には、今回実現した STB 方式におけるスタック分割方式を示す。

2.2 仕事の要求先の決定方式

仕事を要求するプロセッサを決定する方式には、次のようなものが考えられる。

(1) **ローカルカウンタ方式** 各プロセッサがそれぞれプロセッサ番号をラウンドロビンに生成するカウンタを持つ方式。さらに、次の 2 種類のカウンタが考えられる。

  (a) **try next 方式** カウンタを参照するたびに 1 加算するカウンタ。

  (b) **try again 方式** 仕事を要求しても得られなかった場合にのみ 1 加算するカウンタ。

(2) **グローバルカウンタ方式** 全てのプロセッサが特定のプロセッサ上におかれた同一のカウンタを参照する方式。参照する度に 1 加算する。

(3) **乱数方式** プロセッサ番号を乱数で得る方式。

図 2 には、初期状態で PE0 にルートノードが与えられ、仕事の要求がグローバルカウンタ方式でなされて、分割されたスタックが拡散されていく様子を示されている。なお、上記のいずれの方式においても実行の初期段階では仕事を要求しても得られない確率が高く、次第に得られる確率が高くなっていく。また、システム全体で生成される仕事の要求メッセージの数は、グローバルカウンタ方式では他の方式より少ないことが予想される。ただし、カウンタ値を参照するためにプロセッサ間通信が行われるため、仕事を要求してから得られるまでのレスポンス時間は他の方式と比べると悪いと予想される。

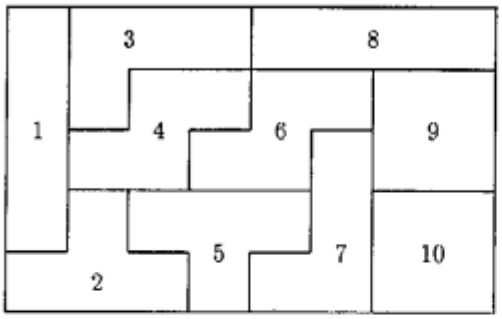


図 3: 詰込みパズル

3 マルチ PSI 上での実現と詰込みパズルへの適用

本節では、マルチ PSI 上でのスタック分割動的負荷分散方式の実現方式と、詰込みパズルへの適用例を述べる。

3.1 マルチ PSI 上での実現

マルチ PSI 上に実現した STB 方式では、スタックは各探索のレベル毎に用意し、自分の手持ちのノードが 2 個以上の場合には各スタックのそれぞれ半分のノードを分割して分け与える。また、仕事を要求するプロセッサを決定する方式に関しては、ローカルカウンタ、グローバルカウンタ、乱数の全方式を実現した。

3.2 詰込みパズルへの適用

3.2.1 問題の説明

詰込みパズルは、様々な形をしたピースを長方形のケースに詰め込むパズルである (図 3)。ここでは、ピースをケースに詰め込む全ての方法を求める全解探索問題として計測に用いた。なお、このパズルはピースが全て 5 つの四角からなる時にはペントミノとして知られている。

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

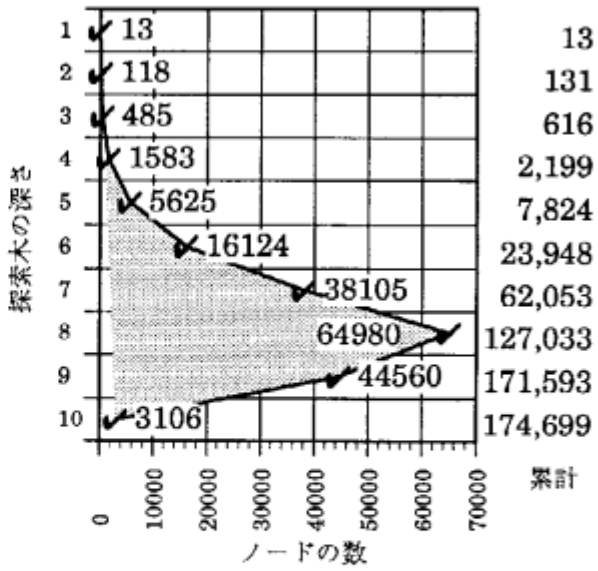


図 4: 各探索の深さにおけるノードの数

3.2.2 問題の特徴

詰込みパズルはピースの数が探索木の深さと等しく、探索の各レベルで生成される OR 部分木の数は図 4 に示した通りである。深さ 10 におけるノードの数 3,106 が解の個数を示す。また、カットオフ以下の OR 部分木の大きさにはばらつきがある。図 5は詰込みパズルの場合でカットオフを深さ 3, 4, 5 に設定した時に、カットオフ以下のノードを探索するのに要する粒度[2]のばらつきを示したもので、各粒度のものがいくつあったかを示している。図からわかるように、カットオフが 3、あるいは 4 の時にはばらつきが大きいが、5 になると比較的粒度の小さいものに集中していることがわかる。

4 性能の測定とその評価

スタック分割動的負荷分散方式をマルチ PSI/V2[8] 上に実現し、詰込みパズルの全解探索問題に適用して性能評価を行った。計測にあたっては、PIMOS[1] 及び KL1 で提供されている計測機能を利用した。

4.1 計測項目

4.1.1 実行時間と台数効果

実行時間は、詰込みパズルの全解探索問題の実行を開始してから全ての解が一台のプロセッサに返される

[2] リダクション数で表す

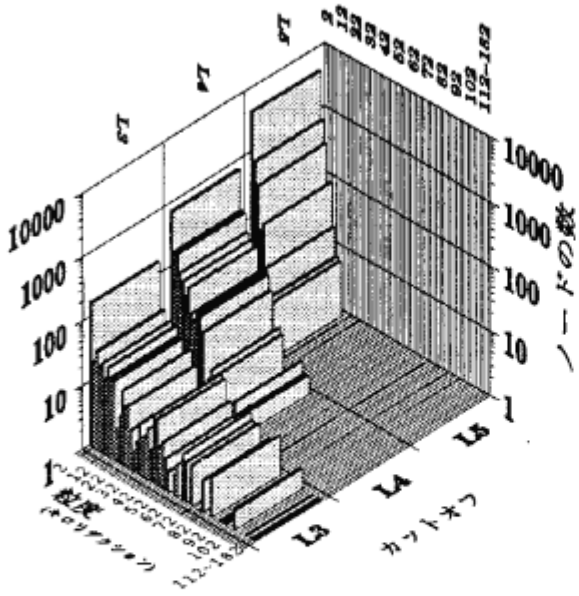


図 5: カットオフと粒度のばらつき

までの時間である。比較のため、MLB 方式で 64 台プロセッサの時に一番性能が良かった 3 レベルの負荷分散を行なった時と、STB 方式でカットオフを 3、5、9 に設定した時の実行時間を計測した。

また、台数効果は一台のプロセッサで実行した時の実行時間を T_1 とし、N 台のプロセッサで実行した時の実行時間を T_N とした時、$\frac{T_1}{T_N}$ で表される実行時間の比である。ここでは、1 台のプロセッサ上で逐次探索を行なった時の実行時間を T_1 とする。

4.1.2 リダクション数

リダクション数は全体の仕事量をあらわし、2 台以上で実行した場合のリダクション数は、実際に問題を解くのに要した仕事量と負荷分散のために新たに必要となった仕事量 (負荷分散オーバヘッド) の合計である。

4.1.3 プロセッサ毎の稼働率

実行を開始してから終了するまでに要した時間のうち、各プロセッサがアイドルになっていた時間の合計の占める比率がそのプロセッサの平均稼働率である。また、開始してから終了するまでの間各プロセッサが一定の時間毎にどの程度忙しかったか、その度合の時間変化の様子を示すものがプロセッサ毎の稼働率の時間変化である。これらは、KL1 ファームウェアで提供されるプロセッサプロファイル機能を用いて計測した。

4.1.4 プロセッサ間通信

STB 方式では、仕事を要求するメッセージを送信しても必ずしも仕事が得られるとは限らない。全要求メッセージのうち、どの程度の割合で仕事が得られたか、及びどれぐらい時間がかかったかを、KL1 ファームウェアで提供されるプロセッサプロファイラ機能を用いて計測した。

4.2 計測結果と考察

4.2.1 実行時間及び台数効果

表 1 に実行時間の計測結果を示す。なお、MLB の 1 台の時の実行時間は実際には 1 台のプロセッサで逐次探索を行なった時の実行時間である。また、カットオフを変えた時の台数効果の違いを比較するために、STB 方式のローカルカウンタで try again 方式の台数効果を図 6 に示す。更に、STB の各方式毎の違いを比較するために、カットオフを L5 とした時の各方式毎の台数効果を図 8 に示す。

1 台の時の性能は、各方式ともカットオフが L3、及び L5 に時にはほぼ同じ実行時間である。しかし、L9 の時には極端に実行時間が長い。これは、図 7 に示したリダクション数からわかるように、L9 の時には総リダクション数が大きいのが原因である。カットオフを深くすると負荷バランスするノードの数が増えるため、負荷分散オーバヘッドは増大する。特に詰込みパズルの場合においては、STB 方式を適用するにあたって、ノードを分散する際の通信量を少なくするために、ノードデータをスタックに出し入れする際にデータの圧縮・伸張を行なうように作っており、これが大きな負荷分散オーバヘッドとなっている。

表 1 より、プロセッサ台数を増やすといずれの方式でも実行時間は短くなり、図 6 よりカットオフが L5 の時にはほぼ線形に近い台数効果が得られていることがわかる。また、MLB 方式と STB 方式はカットオフを同じにした場合にはほぼ同程度の台数効果が得られていることがわかる。

また図 8 より、STB 方式においてはランダム方式が一番性能が良く、次にグローバルカウンタ方式、ローカルカウンタ方式の try again 方式、そして try next 方式の順でわずかずつ性能の差が出ていることがわかる。

4.2.2 プロセッサ毎の稼働率及びその時間変化

図 9 には、MLB 方式と STB 方式でローカルカウンタの try again 方式を 64 台で実行した時の、実行を開始してから終了するまでの間の各プロセッサ毎の平均稼働率を示す。MLB 方式では稼働率 80% の付近で負荷はバランスされているが、STB 方式の L3 では極端に稼働率が高いプロセッサがいくつかある他、多くのプロセッサの稼働率は大変低い。これは、稼働率が高いプロセッサには粒度の大きな仕事が割り付けられたために、他のプロセッサは全ての仕事を終えた後に実行すべき仕事がないという状態をあらわしてい

表1　実行時間　（単位：秒）

			Cutoff	1PE	8PE	16PE	32PE	64PE
MLB			L1-3-5	225.46	29.87	15.43	8.29	4.64
STB	Local counter	Try again	L3	230.58	28.81	20.39	10.83	10.51
			L5	234.89	29.39	15.23	8.50	5.16
			L9	343.28	42.83	22.28	11.63	6.89
		Try next	L3	227.95	29.69	16.68	13.33	11.29
			L5	234.58	30.04	15.09	8.11	5.20
			L9	339.17	43.99	22.30	11.63	6.86
	Global counter		L3	229.30	30.11	18.44	11.08	10.53
			L5	240.88	29.63	15.07	8.73	5.12
			L9	344.58	43.53	22.57	12.46	8.02
	Random		L3	229.48	30.20	18.10	11.59	11.36
			L5	233.88	29.87	14.98	8.28	4.93
			L9	351.58	43.63	22.18	11.53	6.64

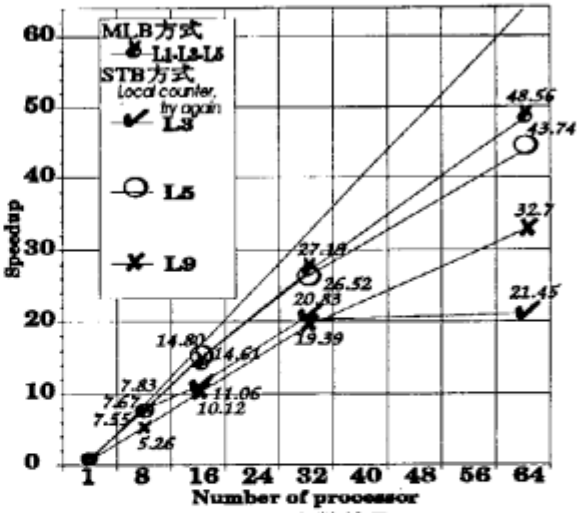


図6　台数効果
(Local counter, try again)

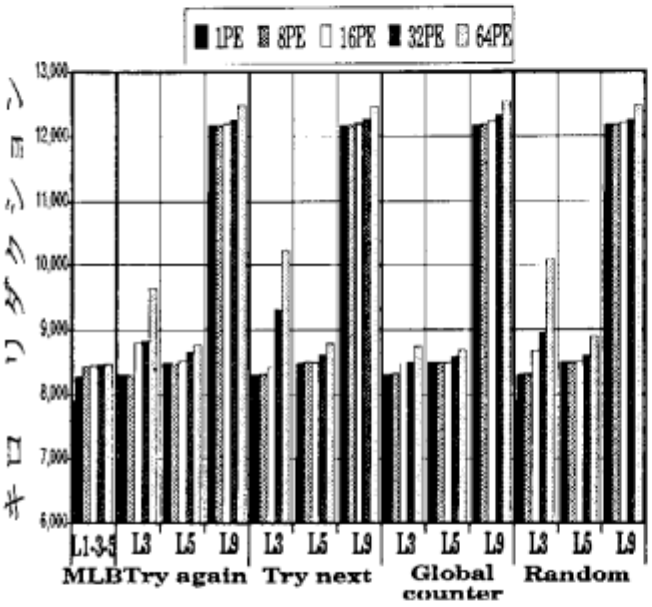


図7　各方式、カットオフ毎のリダクション数

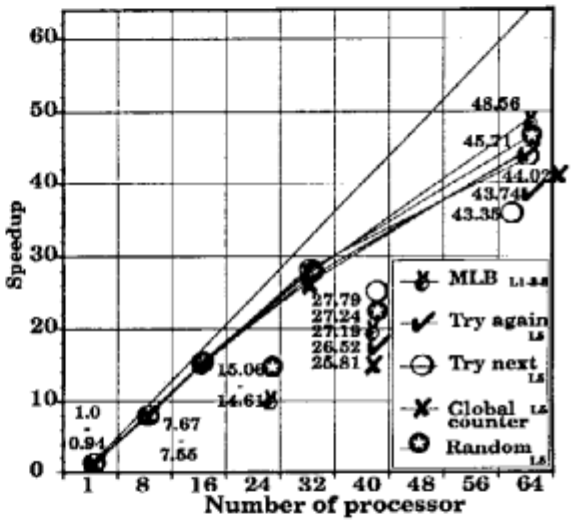


図8　台数効果
(カットオフL5の全方式)

る。これは、各プロセッサの稼働率の 100 ミリ秒毎の変化を 0% から 100% までの 5 段階の色で示した図 10 を見ると明らかである。L3 以外の場合には、実行が始まると同時に全てのプロセッサが忙しくなり、終盤では仕事のなくなったプロセッサがほぼ一斉に暇になっていく様子がわかる。

ここで着目しなければならないのは、L5 も L9 もどちらも負荷バランスは大変良く、全てのプロセッサは忙しく動いているのに、L9 の方が約 5 割も余分に実行時間がかかっている点である。総リダクション数を比較してみると、L9 は L5 よりも 42% 多い。従って、実行時間が多くかかるのは負荷分散オーバヘッドのために総仕事量が多くなるのが原因であることがわかる。

4.2.3　プロセッサ間通信

図11には、STB の各方式についてカットオフを変えて 64 台のプロセッサで実行した時の、仕事の要求メッセージの総数をグラフの高さで示すとともに、そのうち実際に仕事が得られた応答メッセージの数と、仕事が得られなかった応答メッセージの数の比率を示している。L3 と L5 の場合をみてわかるのは、グローバルカウンタ方式は他の方式と比べてメッセージの総数が少ないことと、全てのカットオフの場合で仕事が得られた応答メッセージの比率が他の方式より比較的多いことである。しかし、要求メッセージを送信してから応答メッセージを受信するまでの平均時間を示した図 12からわかるように、グローバルカウンタ方式は他の方式と比べるとその時間が 3 倍から 8 倍ほど多いことがわかる。これは、他の方式では不要な、カウン

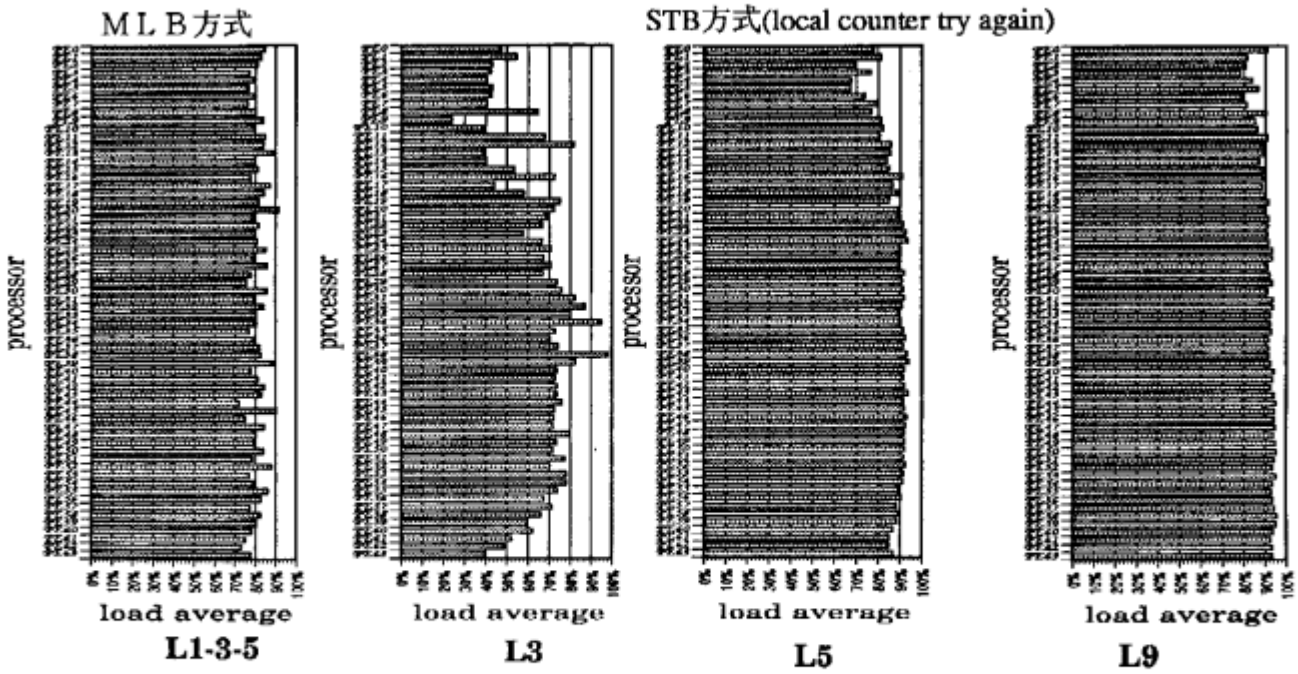


図9　プロセッサ毎の平均稼働率

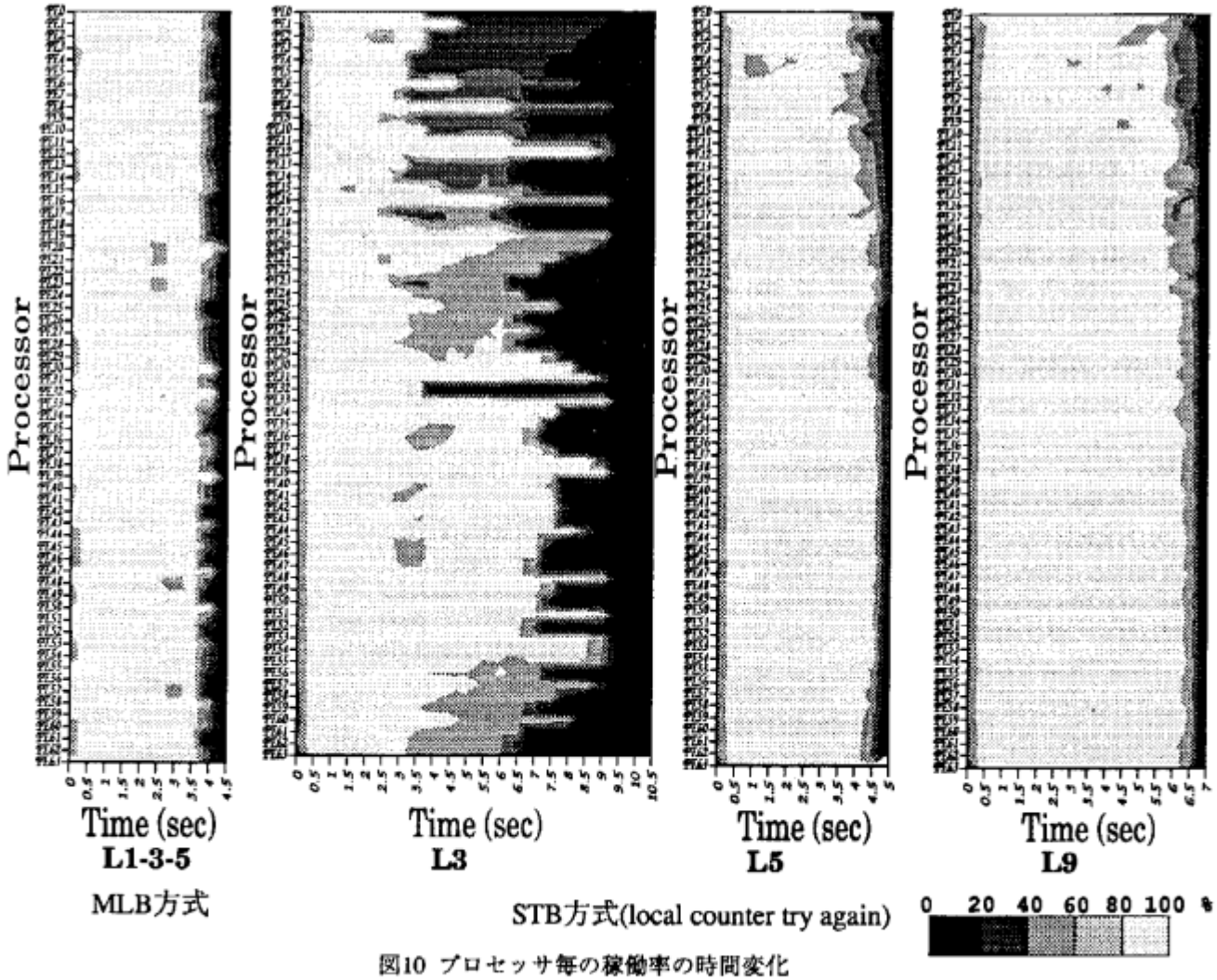


図10 プロセッサ毎の稼働率の時間変化

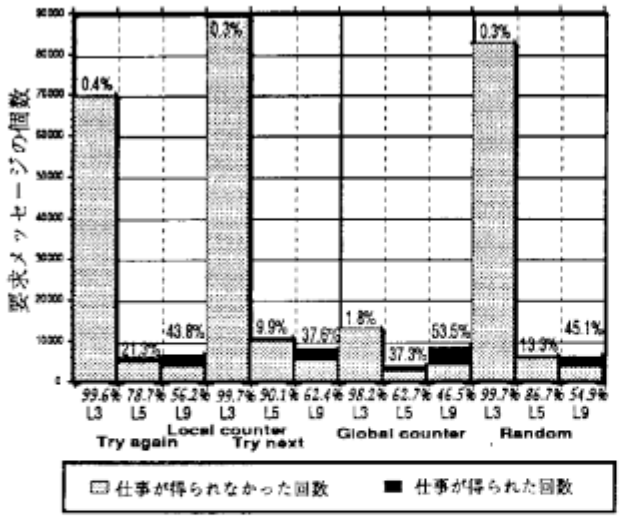


図11 仕事の要求メッセージの個数

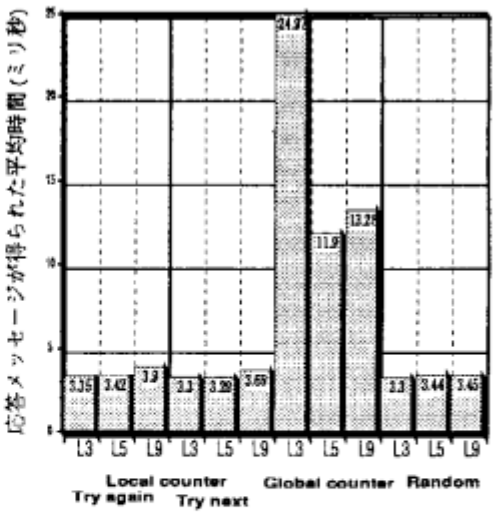


図12 応答メッセージが得られた平均時間

タのあるプロセッサとの通信がボトルネックとなるのが原因である。

5 STB 方式と MLB 方式の比較

本節では、STB 方式と MLB 方式がそれぞれどのような問題に適しており、KL1 でプログラムを作成する場合にどのように使い分ければ良いかの指針を示す。

5.1 性能の比較

前節で述べた計測結果からわかるように、チューニングによって最適なパラメタを得た場合には、STB 方式と MLB 方式には性能的にはほとんど差はない。ただ、MLB 方式ではスタック管理を KL1 言語処理系(ファームウェア)が行なうのに比べて、STB 方式ではこれをソフトウェアで行うため、このためのオーバヘッドによって STB 方式の方が性能が悪い場合が多いが、その差はわずかである。

5.2 適した問題

いずれの方式も、深さ優先探索問題一般に適している。ただし、プログラミングスタイルの面からみると、両者は大変異なる。MLB 方式の場合には、次のプログラム例のように、負荷分散を行う部分 (5) にプラグマを付加する。

```
search(Node,BL):- true |                    %(1)
   BL = {BL0,BL1},                          %(2)
   BL0 = [get_pe(PE)|BL2],                  %(3)
   expand(Node,NewNode1,NewNode2),          %(4)
   search(NewNode1,BL2)@node(PE),           %(5)
   search(NewNode2,BL1).                    %(6)
```

ここで、説明を簡単にするため探索木は 2 分木であるとする。割り付けるプロセッサ、すなわち仕事を要求してきたプロセッサの番号は、マルチレベル負荷バランサーへのストリーム (BL) に対してメッセージ get_pe(PE) を流すことによって得られるとする。このように、ユーザがプラグマを付加するため、マルチレベル負荷バランサーを別のものと入れ換えることによって、色々な負荷分散方式を試す事ができる。

一方、STB 方式の場合には次のプログラム例のようにユーザはスタックからノードを取り出して探索を一手進め、その結果をスタックに入れるだけで、後はスタック分割動的負荷バランサーが自動的に負荷分散を行なってくれる。

```
search(Stack):- true |                              %(7)
   Stack=
     [pop(Node),
      push(NewNode1),push(NewNode2)|Stack1],%(8)
   expand(Node,NewNode1,NewNode2),                  %(9)
   search(Stack1).                                  %(10)
```

ただし、探索の実行開始前には全プロセッサの上に search プロセスを 1 個ずつ常駐させておき、また全プロセッサ間でストリームによる全結合ネットワークを張っておく必要があるが、それらはスタック分割負荷バランサーが自動的に行う。従って、STB 方式を使う場合にはシステムが自動的に行ってくれる部分が多く、ユーザにとって使い易いと思われるが、プラグマの付け変えによって色々な負荷分散方式を試したい場合には適さない。

以上のことより、MLB 方式と STB 方式を使い分けるに際しては、対象とする問題によって使い分けるよりも、目的によって使い分けるのが良いと思われる。目的が探索問題を解くことであれば、STB 方式を採用して負荷分散はシステムに任せれば良い。負荷分散方式を色々試して高速性能を追求する場合には、まずは MLB 方式を適用して基本的な性能を追求し、その後各ユーザで色々な方式を試せば良い。

5.3 パラメタチューニングの容易さ

STB 方式と MLB 方式を、チューニングの容易さの観点から比較してみる。両方式とも、より良い性能を得るためはユーザはパラメタのチューニングを行なわねばならない。MLB 方式の場合には、マルチレベルを行なう段数とそれぞれの深さがパラメタである。一方、STB の場合にはカットオフのみである。従って、チューニングの容易さの観点からは、パラメタの数が少ない STB 方式の方が容易であると考えられる。

6 おわりに

以上、スタック分割動的負荷分散方式のマルチ PSI 上での実現と評価について述べた。本方式は、深さ優先探索問題に代表されるスタックを用いた問題に適用可能な動的負荷分散方式であり、各プロセッサは自分の仕事がなくなると他のプロセッサに仕事を要求するメッセージを送り、スタックの一部を得ることによって負荷の均等化を行なう。どのプロセッサに対して要求メッセージを送るかを決める方式にはローカルカウンタ方式、グローバルカウンタ方式、乱数方式があり、いずれの方式でもマルチ PSI 上で詰込みパズルの全解探索問題に適用したところ、マルチレベル動的負荷分散方式を適用した場合に近い性能が得られ、ほぼ台数に比例した台数効果を得ることができた。

また、STB 方式と MLB 方式の違いを比較し、新たな問題が与えられた時にどちらの方式を採用するか決める際の指針を示した。

本方式を、スタックを用いた各種の応用問題に適用して評価することが今後の課題である。なお、STB 方式はマルチ PSI 及び PIM のユーザが簡単に使えるような形で PIMOS のユーティリィテとして提供することを予定している。

7 謝辞

本研究においては、スタック分割動的負荷分散方式を提案すると同時にマルチ PSI 上での評価を助言していただいたミネソタ大学の V. Kumar 氏、数々の貴重な助言をいただいた ICOT 第 1 研究室の 瀧和男 室長らに感謝致します。

参考文献

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

[2] M. Furuichi, K. Taki, and N. Ichiyoshi "A Multi-Level Load Balancing Scheme for OR-Parallel Exhaustive Search Programs on the Multi-PSI" In *Proc. of the 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 50-59, Mar. 1990

[3] 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」 *KL1 Programming Workshop '90* pp. 1-9, 1990 年 5 月

[4] K. Kimura and N. Ichiyoshi "Probabilistic Analysis of the Optimal Efficiency of the Multi-Level Dynamic Load Balancing Scheme." *to appear at DMCC6*, 1991.

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

[6] V. Kumar and V. N. Rao. "Scalable Parallel Formulations of Depth-First Search" In *Parallel Algorithms in Machine Intelligence and Vision*", Springer-Verlag, pp. 1-41, 1990

[7] K. Kumon, H. Masuzawa, A. Itashiki, K. Satoh, and Y. Sohma. "Kabuwake: A new parallel inference method and its evaluation" In *Proceedings of COMPCON 86*, March 1986.

[8] K. Nakajima, Y. Inamura, N. Ichiyoshi, K. Rokusawa, and T. Chikayama. "Distributed implementation of KL1 on the Multi-PSI/V2". In *Proceedings of the Sixth International Conference on Logic Programming*, pp. 436–451, 1989.

囲碁対局システム・並列版『碁世代』の試作

清 慎一[1], 沖 廣明[2], 瀧 和男[1]
1:(財) 新世代コンピュータ技術開発機構, 2:(株) 未来技術研究所

概要

我々は、並列推論マシン"マルチ PSI"上の応用プログラムとして、並列論理型言語 KL1 を用い、囲碁の対局を行なうシステム、並列版「碁世代」を作成中である。並列版「碁世代」は、ICOT が逐次型推論マシン PSI 上に開発した囲碁の対局を行なうシステム「碁世代」の並列化を目指したものである。

我々は並列版「碁世代」の開発の初期段階として、逐次マシン上のプログラムと並列マシン上のプログラムとが通信を行なって一つの機能を果たすような試作システムを実現した。その構成、並列化の要点、並列論理型言語 KL1 を用いて実装する上での工夫点について報告する。また、そのシステムでは、動的負荷分散の中で仕事の大きさによる実行順序付けを行ない負荷バランスを改善する方式を試みたので、それについても報告する。最後に、並列処理を前提とする新しい知識処理の枠組の 1 つとして、「遊軍処理方式」を提案する。

1 はじめに

我々は 1985 年から、逐次推論マシン PSI 上で囲碁の対局を行なうシステム「碁世代」(以降、逐次版「碁世代」と呼ぶ) を開発している。現在、その棋力はアマチュアの初級者レベルである 10 級程度である。このシステムの棋力が低い理由の主な原因の一つに、扱う処理を制限していることが挙げられる。それは、囲碁の対局はリアルタイム性を要求されるために、一回の対局にかかる時間を数十分に抑えることが必要なためである。例えば、探索処理のように処理時間が大きな処理においては、その探索の起動条件を厳しくして、なるべく探索を起こさないようにして時間の節約を図っている。また、システムに取り入れたいが、処理時間が増えるために扱っていない知識も多い。そこで、このシステムの棋力向上を目指すため、より多くの処理を取り込むことのできる並列マシンへの移行が必要になり、並列版「碁世代」の開発を 2 年前から始めた。なお、この並列版「碁世代」は、並列論理型言語 KL1 を用いて記述しており、現在の実験システムは並列推論マシン実験機"マルチ PSI"[7] 上で動作している。また近い将来に、並列推論マシン "PIM"[3] 上へ移行する。

この論文では、並列版「碁世代」の開発の始めの段階として作られた実験システムの構成と、逐次プログラムから並列化を行なう際に工夫した点について報告する。さらに、並列化に際して特に工夫した、動的負荷分散の中で仕事の大きさによる実行順序付けを行ない負荷バランスを改善する方式について、試作・評価結果を報告する。最後に、将来の並列版「碁世代」の中で用いる並列知識処理のアプローチの一つになると思われる「遊軍処理方式」について提案する。

2 囲碁システム「碁世代」

2.1 「碁世代」の概要

我々は、1985 年から AI の例題として大規模知識処理システムである囲碁の対局を行なうシステム「碁世代」の開発を行なっている。この開発の目的は、探索問題、曖昧性の処理、例外処理、協調問題解決などの人工知能の基本的な問題に対する新しいアプローチを探るとともに、大規模知識処理システムの並列化方式の研究を行なうことである。現在このシステムの棋力は、アマチュアの初級レベルであり、かつ、コンピューター囲碁プログラムのトップレベルである 10 級程度である。

なお、「碁世代」は ICOT で開発されたオブジェクト指向型言語 ESP で書かれ、約 3 万 6 千ステップ (コメント除く) の大規模システムであり、逐次推論マシン PSI 上で動作する。[5, 6]

2.2 「碁世代」の処理内容

我々のアプローチは、人間プレイヤーの思考をできるだけ忠実にシミュレートしようとするものであり、それに基づいて「碁世代」の着手決定方法を設計した。

「碁世代」の着手決定の手順を簡単に説明すると、まず着手により変化した盤面上の石の生死などの局面の認識を行ない、それに基づいて候補手を列挙し、その候補手の評価値の合計がもっとも高い点の座標を次の着手位置とする。(図 1)

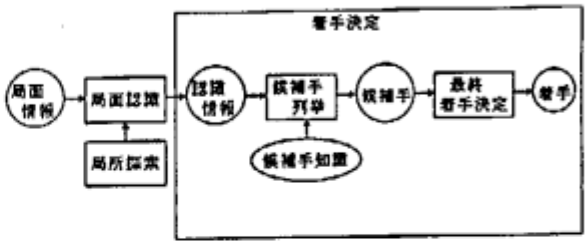


図 1: 着手決定の流れ

- 局面認識

  人間は石の配置を単に座標点の配置として捉えているわけではない。戦術的、戦略的に意味のある石の集団の形態を、経験的に、あるいは誰かに教えられて学習する。そしてそれらを用いて局面の認識を行っている。この「意味のある石の集団形態」は、人間が学習獲得する過程を反映して、階層的に形成される。

「碁世代」では、形態の対象として点、連、群、族を導入し、またその他に連結認識の対象として結線を定義した。(図2)

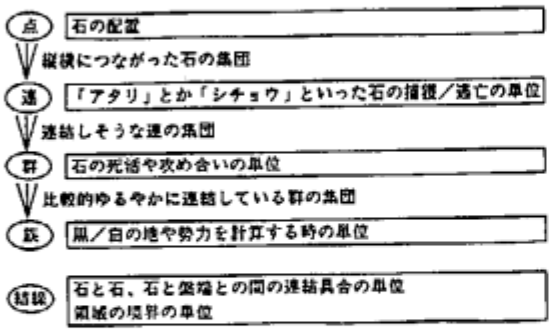


図2: 意味のある集団形態とデータ構造の関係

このようにデータ構造は、"点"から"連"、"群"、そして"族"へと階層構造を形成している。各データ構造の定義及び主な属性を表1に示す。

尚、データ構造の更新は原則としてインクリメンタルな更新、即ち、最終着手の近傍に対してのみ行われ、時間の節約を図っている。

表1: 局面データ構造とその属性

名称	定義	主な属性
点	19 × 19 の格子点	色、ポテンシャル値、隣接点、各種候補手
連	隣接する同色の石の極大集合	石数、ダメ数、種石フラグ、死活タイプ
群	強連結した同色の連の極大集合	石数、中地、眼数、包囲度、手数、強度、重要度、分類
族	ポテンシャル一定値以上で隣接する点の極大集合	石数、辺長、中地、包囲度、強度、重要度
結線	同色の石、または石と盤端との間の仮想線	結線種別、連結種別

- 局所探索

碁の対局中では、局面において局所的に不安定な部分が発生する。例えば捕獲されそうな連、切断されそうな結線、死にそうな群などである。そして、これらの部分の決着次第で局面の状態が大きく変動する時は、これらの結末を実際に先読みする必要がある。

コンピューターチェスのように次の一手を決めるために盤面の全ての手を読むことは、時間的に不可能であり、システムの目的からも外れる。よって、探索は、局所的なもののみが組み込まれている。

局所的な先読みとして、"シチョウ"、"捕獲"、"連結"、"死活"、"攻め合い"という5種類の目的毎の探索を用意した。これは、目的によって候補手生成や終了判定のためのヒューリスティックが異なるため、目的毎に個別に作成した方が高機能が得られるからである。

探索アルゴリズムはアルファ・ベータ枝刈り法を採用した。探索は読み切り型であり、結果は成功か失敗かに分類される。一般に、探索ルーチンの行使は高価であり、頻繁には使用できない。この局所探索が全体の処理時間の約半分を占めている。従って、探索ルーチンを駆動する条件と探索の範囲を限定して(手数200, 深さ20, 候補手・ターゲットの近傍のみ)行っている。

- 候補手列挙

我々は、「与えられた問題解決の場で、人間は判断の基準として過去に遭遇した典型的状況に関する知識に基づき行動している」という仮定を設けた。そこで、方針を決定する着眼点の選出は、碁の知識によって設定された幾つかのケース(典型的状況)の観点から行われる。「碁世代」は、現在どのようなケースが生じているか調べ、これを処理する手段としての候補手及びその手の価値を、候補手知識より導く。一般に、候補手は複数挙げられる。

「碁世代」が持っている候補手は、"定石"、"辺点"、"ダメ点"、"打ち込み"、"ヨセ"、"フトコロ拡大/縮小"、"模様接点"、"捕獲/逃亡"、"連結/切断"、"包囲/脱出"、"群の分離/連結"、"地模様拡大/削減"、"攻め合い"の13種類である。

- 最終着手決定

候補手は、各ケースが互いに独立に挙げてくるため、異種ケース間の不調和を局所的に調整してより効果を高めたり、不調和を削除したりすることもある。このように候補手間の調整を行った後で、各ケースからの候補手の評価値の合計によって、最も評価値の高い点を打着する。

2.3 「碁世代」の並列性

「碁世代」の処理のうち、各処理段階で盤面上に多数発生し、独立に実行可能なものがあり、それらは並列に実行可能だと思われる。

- 局面認識における各オブジェクト(連、群、族など)毎の認識作業
- 局所探索における各オブジェクトの探索。また、その探索自体の並列実行
- 各種の候補手知識毎の候補手列挙と評価値の計算

また、今後取り入れていく処理の中でも並列性のあるものが存在すると期待される。

反対に逐次性をもった処理としては、(1) 連の認識と群の認識、(2) 認識とある種の候補手生成、(3) 候補手列挙と着手決定の間など、前の処理の結果が求まってからでないと、次の処理に進めない部分がある。

3 並列版「碁世代」実験システム

囲碁の対局に必要な処理には、逐次的な処理と並列に行なえる処理とが存在する。そこで、並列に実行可能な大きな部分を並列マシン上のプログラムに実行させ、逐次マシン上で実行したその他の処理の結果とマージさせて、着手を決定する構成をとったシステムを作成した(図3)。

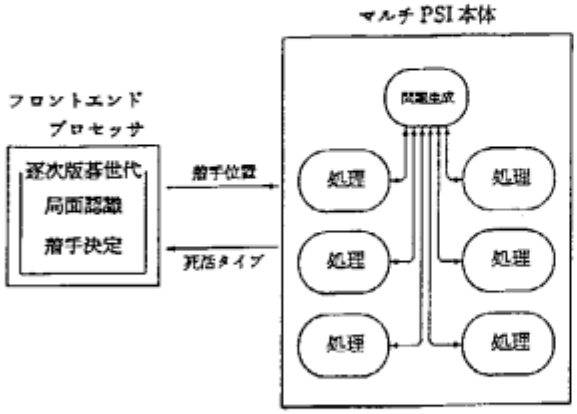


図 3: システムの構成

3.1 マルチ PSI の構造

マルチ PSI[7] は、ICOT が開発している並列推論マシン PIM[3] の実験機である。マルチ PSI は、逐次型推論マシン PSI-II の CPU を要素プロセッサとしてこれらを 2 次元メッシュ状に接続したマシンである。これをマルチ PSI 本体と呼ぶ。このマルチ PSI 本体上で並列論理型言語 KL1 が実行される。入出力機能はフロントエンドプロセッサ (FEP) として接続された PSI-II によって実現される。なお、マルチ PSI の処理系の性能は append で 128KRPS(Kilo Reduction Per Second)。マルチ PSI の各プロセッサのメモリ容量は 80MB(16MW) である。

3.2 逐次プログラムの並列化

我々は ESP 言語で逐次版「碁世代」を開発してきたが、マルチ PSI 上で並列版「碁世代」を動かすためには、KL1 言語に書き直さなければならない。しかしながら、逐次版「碁世代」は約 3 万 6 千行の大規模なシステムであり、すべてを一時に KL1 言語に書き直し並列化するのは容易ではない。

マルチ PSI の FEP には PSI-II が使われていて、マルチ PSI 本体側から PSI-II 上の ESP 言語を実行することができる。我々はこの機能を利用して、マルチ PSI 本体上の KL1 言語で書かれた部分が FEP 上の ESP 言語を呼び出す構成をとることにした。これにより、ESP 言語で書かれた「碁世代」の一部を取り出して、KL1 言語で書いて並列化し、マルチ PSI 本体から FEP 上の ESP 言語で書かれた述語を呼びだし、本体と FEP が合わさって囲碁の対局を行なうことができる。

このような構成をとることによって、徐々に KL1 言語に書き直しても、囲碁の対局を行なうことができるので、並列化した部分の効果を確かめながら作成することができる。

3.3 実験システムの処理内容

本実験システムでは並列化の始めの段階として、並列化による効果が大きいと思われる、囲碁において重要でかつ大きな処理である石の生き死にを判定する捕獲探索を取り出し、並列化した。この探索は盤面上に複数存在し、各々の処理は独立に実行できるので、各探索は並列に実行できる。また、この実験システムでは逐次版「碁世代」では扱っていなかった「消し候補手」の生成処理も並列に行なわせている。

一方、次の一手の候補手を生み出す為には、盤面の認識ができていないといけないので、盤面の認識と候補手生成の間には逐次性が存在する。今回の試作では、フロントエンドプロセッサである逐次推論マシン PSI には逐次の囲碁システム「碁世代」が存在し、並列推論マシンであるマルチ PSI には盤面上の石の生死を探索する処理と消し候補手の生成処理を行なうプログラムが存在する。逐次版「碁世代」は石の生死の探索と消し候補手生成の命令を並列版「碁世代」に通信して、その処理結果と逐次版「碁世代」が行なった処理結果とを検討して、次の着手を決定している。

相手の打着から自分の着手を決めるまでの処理の流れは次の通りある。(図 4)

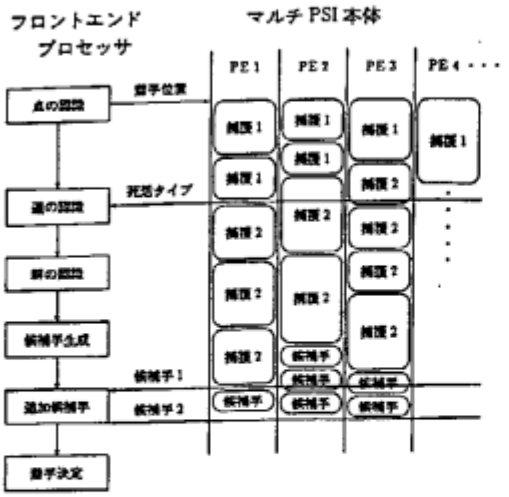
1. フロントエンドプロセッサ (FEP) は、相手の着手位置をマルチ PSI に通信する。
2. マルチ PSI は、着手によって生じた盤面上の複数の捕獲探索を処理する。
3. マルチ PSI は、ダメ[1]が 3 以下の連の捕獲探索の結果 (死活タイプ) を FEP に通信する。
4. FEP は、マルチ PSI から返ってきた捕獲探索の結果に基づき、群 (ある程度つながっていると思われる石) や族の認識処理などの高度な認識を行なう。
5. FEP は、それらの認識結果に基づき候補手の生成を行なう。
6. マルチ PSI はその間にダメが 4 の捕獲探索問題を実行する。
7. マルチ PSI は、ダメが 4 の連の捕獲探索の結果 (ダメが 4 の連への先手候補手) を FEP に通信する。
8. さらにマルチ PSI は、消し候補手生成処理を実行する。
9. マルチ PSI は消し候補手を FEP に通信する。
10. FEP は、マルチ PSI から得られた候補手と FEP 上の碁世代がつくり出した候補手から、着手を決定する。

なお、並列実行した捕獲探索と消し候補手生成処理を以下に触れておく。

- 捕獲探索

  囲碁では、相手の石によって隣接点すべてを囲まれると盤上から取り除かれてしまう (最善手を打ち続けて

[1]囲碁用語でつながった石の隣接点にある空点のこと。ダメの数が少ないほど敵に取られやすいことを示す



捕獲 1: ダメ 3 の連の捕獲探索, 捕獲 2: ダメ 4 の連の捕獲探索
候補手: 消し候補手生成
候補手 1: ダメ 4 の連への先手候補手, 候補手 2: 消し候補手

図 4: 処理の流れ

も最終的には相手に囲まれてしまう石を「死に」といい、その逆を「生き」という)。(図 5) 縦横につながった石 (我々はこれを連と呼んでいる) は、囲碁において石の捕獲の最小単位であり、最終的な石の生き死にの単位となる群の構成要素でもある。したがって、この連の生き死にを判断することは、囲碁を打つ上で大変重要で、多少時間がかかっても正しい判断を行なうべき処理である。この実験では、連のダメが 4 以下のものについて、探索によってその生死の判断を行なう処理を行なった。逐次版「碁世代」では、対局時間を増やさないためにダメが 3 以下の連しか扱っていないほど、大きな処理である。

図 5: 石の捕獲

- 消し候補手生成

  囲碁用語で「消し」とは、相手の勢力圏がそのまま相手の地になることを防ぐために、敵の勢力によって取られない程度の勢力圏の境目あたりに着手し、勢力圏を狭めることである。一般に、敵の勢力の端点を結ぶ線の中心に打つのが良いとされている。(図 6)

  このシステムでは、盤上の全部の点に対して、その周りの状態を調べ、そこが敵の勢力の端点を結ぶ線の中心であるかどうかを判断することにより「消し候補手」を求めている。この処理は、数は多いが、探索処理と比べると小さな仕事である。

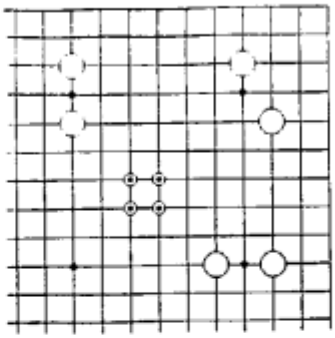


図 6: 消し候補手の例

3.4 実装上の工夫

3.4.1 プロセッサ間の通信

一般に並列プログラムの作成にあたっては、プロセッサ間の通信量を少なくすることが重要である。我々は通信量を減らすために、必要最小限の情報だけを通信するように、あらかじめ各プロセッサに盤面情報を持たせ、変化分だけを通信させるようにした。

プログラムは、処理を行なう単位であるクライアントと、各クライアントに仕事を与えたり、フロントエンドプロセッサ (FEP) との通信を行なうサーバーとからなる。まず対局が始まる際に、各プロセッサに 1 つずつクライアントが配置される。サーバーは、FEP から来た相手の着手位置を各クライアントに送り、認識作業を実行させる。但し、大きな処理である局所探索の中のつながった石の捕獲探索は、サーバーが暇なクライアントに割り振り実行させる。その捕獲探索の結果はサーバーに集まり、全部のクライアントに結果を知らせる。これにより、各クライアントは常に同じ情報を持つ。通信される内容は、相手の着手・処理実行の命令・捕獲探索結果であり、盤面全体の情報ではなく、着手による盤面の変化分の情報だけとなる。

これは各クライアントが同じ情報を持つというデータの重複を招くが、一つの盤面情報を集中管理することによるボトルネックを防ぎ、PE 間の一回に送る通信量が少なくてすみ、通信量の低減となる。

3.4.2 負荷分散

並列プログラムの実行においては、プロセッサ間の負荷を均等にすることも重要である。

本実験システムでは、動的負荷分散方式を使用して暇なプロセッサを検出し、それに仕事を与えることにより均等な負荷分散を実現した。また、動的負荷分散の中で仕事の粒の大きさによる実行順序付けをすることにより、負荷の均等化を図った。これらの方式については次章で詳細に述べる。

3.4.3 データ構造

囲碁システムでは、局面の認識の為に大量で複雑なデータ構造を保持する必要がある。そこで、その複雑なデータ構造を保持する方式として、局面認識データをベクタ上に表現した。また、局面認識データ内のあるデータへのアクセスは、ポインターではなく効率を考えて連番号などのベ

クタの第N要素を表す識別子とした。

KL1言語によるデータ構造の記述としては、各データをプロセスにして表す方法とベクタで表現するする方法が一般的である。もし「碁世代」の局面データの表現方法を、全ての点・連をプロセスとする持ち方にすると、並列性が上がるが、各々の粒度が小さく、あるデータにアクセスする際にプロセス間の通信量が増えるという問題が生じる。しかしながら、ベクタによってあらゆる属性を一つの共有データとしてまとめると、並列性は下がるが、あるデータにアクセスする際の処理時間はプロセス形式より短い。並列版「碁世代」では、各処理段階での並列性が十分にあるため、各データをプロセスにすることによってさらに並列性を上げる必要はないので、データは、ベクタによって表現した。

マルチPSIの処理系では、データへの参照が一つしかないデータと、複数の参照があるかもしれないデータとを区別し、前者について廃集め処理の効率化を行なう方式を採用している。[1] そこで、複数参照をせずに要素をアクセスする方法としてデータのポインタを持つことによる参照数の増加を避け、識別子を使ってデータをアクセスすることにした。

3.5 試作結果

我々が作成したこの実験システムでは、それまでに作成された逐次版システムに、ダメが4の連の捕獲候補手の生成と、消し候補手の生成処理が加わっている。また、逐次版では着手位置の近傍しか行なっていなかったダメが3の連の捕獲探索処理を、盤面上のすべてのダメ3以下の連の捕獲探索処理に拡張した。

そこで、実験システムで扱っている処理時間の総量を計算すると以下のようになる。ダメが3以下の連の捕獲探索処理量は、逐次版の約10倍。ダメが3以下の連とダメが4の連の探索処理量と消し候補手生成処理量の合計は、ダメが3以下の連の捕獲探索処理量の約10倍。逐次版のすべての処理時間中の連の捕獲探索の時間は約$\frac{1}{10}$。また、マルチPSIのプロセッサ1台上でのKL1の実行速度は、PSI-II上のESPの実行速度の$\frac{1}{3}$と言われている。ゆえに、$10 \times 10 \times 3 \times \frac{1}{10} = 30$倍の時間がかかるはずである。もし、台数効果が理想的(NプロセッサでN倍速い)ならば、32台のプロセッサを使うと、PSI-II上の「碁世代」とほぼ同じと処理時間はほぼ同じになる。

処理時間の計測結果は表2の下表の通りである。逐次型「碁世代」での測定では処理時間が約3秒であったのに対し、32台のプロセッサでは2~5倍程度遅くなった。理由は、台数効果の値が理想値より低く、32台で6~17倍の高速化に留まっているためである。

この実験システムの強さについては詳しい解析はしていない。それは、候補手が増えたことによる他の候補手の評価値とのバランスを考えた調整作業をしていないからである。増やした候補手自体は効果があると思われるので、評価は今後の課題である。

以上から、このような部分的に並列実行を行なうシステムでも、実行時間はほぼ同程度で、システムの棋力を上げるための処理を増やすことができるといえる。

4 仕事の粒の大きさによる実行順序付けの実験

並列実行において重要な問題の一つに、如何に均等にプロセッサに負荷を分散させるか、と言うのがある。我々は、この実験システムにおいて並列実行する部分の負荷の均等化のための実験をおこなった。それは、基本的な方式として動的負荷分散方式[4]を用いつつ、並列実行させる処理の粒の大きさが予想できる場合には、粒の大きな仕事を小さな仕事より優先的に実行させることにより稼働率を上げるというものである。

4.1 動的負荷分散方式

並列マシンを有効に利用するためには、負荷の均等化が重要な問題である。動的負荷分散方式は、暇なプロセッサをみつけ、その暇なプロセッサに対してのみ仕事を送り付ける方式である。暇なプロセッサの検出には、各プロセッサに予め最低プライオリティのプロセスを常駐させておき、暇になったらその旨をストリームを介して報告させる。動的負荷分散方式では、どのプロセッサも同程度の忙しさとなり、負荷の均等化のための有効な方式である。

4.2 処理の終了間際の負荷の不均衡

しかし動的負荷分散方式を用いても、全体の処理が終了する間際には、あるプロセッサだけが忙しくなり他のプロセッサは暇になってしまうことがある。そのうえ最後に残った仕事の粒が大きいと、その時間が大きくなるという問題がある。

この終了間際の負荷の不均衡を回避するための方法として、プロセッサに割り当てる仕事の数と粒度をあらかじめ調整すると良いことが、すでに論じられている。[2]

4.3 仕事の粒の大きさによる実行順序付け

我々は、仕事の終了間際の負荷を均等にする方法として、仕事の大きさがあらかじめ予想でき、大きな仕事のグループと小さな仕事のグループに分けられるならば、大きな仕事を始めに投げて、小さな処理は大きな処理を全て投げ終ってから配ることを考えた。

動的負荷分散方式を用いて、大きな仕事を先に与え小さな仕事を後から与えるようにすると、大きな仕事の終了する時刻のばらつきを小さな仕事が埋める形になり、各プロセッサがほぼ同時に処理を終了することができる。我々は、仕事を大きさによって2種類に分け、大きな仕事は投げる優先順位を高く、小さな仕事は低くすることにより、それを実現させた。(図7)

我々は、大きな処理としてつながった石の生死を判断する捕獲探索を、小さな処理として消し候補手(敵の模様を侵略する手)の生成を例に取り、実験を行なった。

4.4 測定結果

我々は実際の対局から中盤の局面を取りだし、次の一手を求める処理を行ない、その実行時間の測定を行なった。(表2,3, 図8,9)

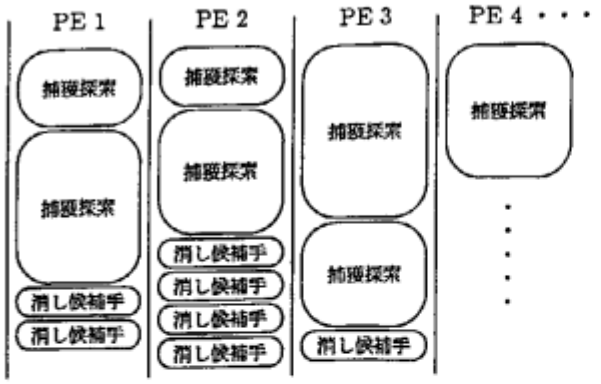


図 7: 大きな仕事と小さな仕事の実行の様子

表 2: 実行時間 (秒)

大きい粒の仕事 (捕獲探索) のみ					
問題図	t1	t2	t3	t4	t5
1 PE	88.5	81.3	43.4	42.9	85.7
2 PE	44.9	42.2	23.6	21.7	43.2
4 PE	23.2	21.5	13.5	13.6	22.6
8 PE	13.9	14.7	8.7	10.8	17.2
16 PE	7.9	10.3	5.8	9.4	15.7
32 PE	7.7	9.3	5.8	8.7	15.7
大小両方の粒の仕事 (捕獲探索と消し候補手) を実行した時					
問題図	t1	t2	t3	t4	t5
1 PE	102.9	92.0	58.0	58.8	108.2
2 PE	52.1	46.1	29.3	29.7	54.1
4 PE	26.2	23.2	14.8	15.2	27.1
8 PE	13.9	14.7	8.7	10.8	19.9
16 PE	7.9	10.3	5.8	9.4	16.2
32 PE	7.6	9.3	5.8	8.7	16.0

ここでの稼働率は以下の式で求めた。

1 台での実行時間/(N台での実行時間 × 台数)

測定の結果、大きな粒の処理のみを実行させた場合と、大きな粒と小さな粒の処理の両方を実行させた場合とを比べると、処理時間は殆ど増えずにプロセッサ稼働率が上昇した。これは、我々の方法が効果があったことを示している。

4.5 考察

我々の方法のように、大きい粒の仕事群を先にプロセッサに割り付け、小さい仕事群を後からプロセッサに割り付けることが有効なのはどのような場合であろうか。

動的負荷分散を行なった場合に、最悪の稼働率が出るのは次の場合である。始めに最大粒の処理以外の処理がすべてのプロセッサに均等に分けられ、すべてのプロセッサが同時に処理を終った後、1 台のプロセッサが最大粒の仕事を割り当てられ実行した場合である。ただし、通信の手間や仕事の生成がボトルネックとなっていないと仮定する。

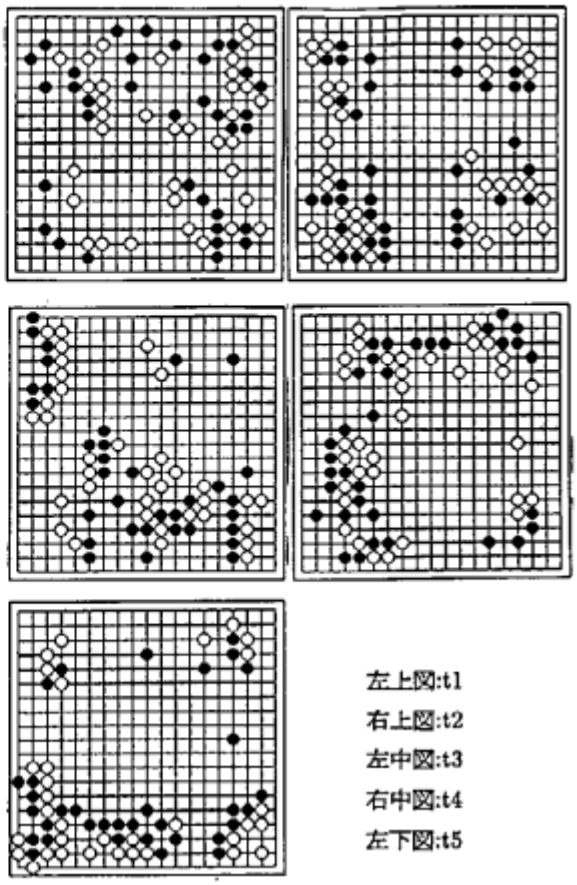


図 8: 測定に使った中盤の局面 (出典:1988 年天元戦挑戦手合)

$$\text{最悪稼働率} = \frac{\text{全処理量}}{\text{最大粒以外の全処理} + \text{最大粒} \times \text{PE 台数}}$$

では、我々の方法のように、大きい粒の仕事群を先にプロセッサに割り付け、小さい仕事群を後からプロセッサに割り付ける場合の、稼働率はどうなるであろうか。これは、小さい粒の仕事の総量により分けて考えられる。

一つは、小さい粒の仕事が少ない時である。言いかえると、小さい仕事の粒が大きい仕事の終了時刻のバラツキを埋められなかった場合である。もう一つは、小さい粒の仕事が大きい粒の仕事の終了時刻のバラツキを完全に埋めつくす場合である。

小さい粒が大きい粒の終了時刻のバラツキを完全に埋めるためには、以下の式を満たさなければならない。

最大粒×(全プロセッサ数−1) ≤ 小さい粒群の仕事の総量

勿論、大きい粒の仕事群の実行で、最大粒の仕事が最後に実行されないこともあるので、上の不等式を満たさなくても我々の方法が有効な場合はある。また、大きい粒のバラツキを完全に埋めなくても、ユーザーが期待している稼働率を満足する場合もあるので、その時も上の不等式を満たす必要はない。

表 3: 稼働率 (%)

大きい粒の仕事 (捕獲探索) のみ					
問題図	t1	t2	t3	t4	t5
1 PE	100	100	100	100	100
2 PE	98	96	92	99	99
4 PE	95	94	80	79	95
8 PE	79	69	63	49	62
16 PE	70	49	47	29	34
32 PE	36	27	23	15	17
大小両方の粒の仕事 (捕獲探索と消し候補手) を実行した時					
問題図	t1	t2	t3	t4	t5
1 PE	100	100	100	100	100
2 PE	99	100	99	99	100
4 PE	98	99	98	97	100
8 PE	92	78	84	68	68
16 PE	81	55	62	39	40
32 PE	42	31	31	21	21

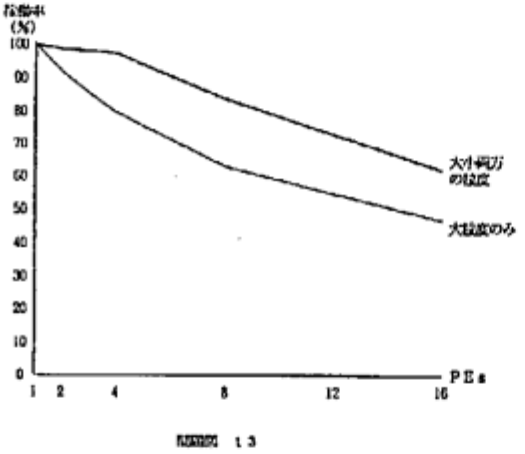


図 9: 稼働率の向上が大きかったもの

我々が実験したデータでは、どのケースも 2 ～ 4 台程度のプロセッサを使うと、上の不等式を満たしており、期待通りの稼働率が得られる。(表 4,5) 稼働率の実測値では、4 台以上でも 100% に近い稼働率も出ている。(表 3)

ある処理を並列実行しようとしたら、並列実行される粒の大きさと、プロセッサの台数から最悪の稼働率が計算できる。もし、算出された最悪稼働率が期待した稼働率よりも低ければ、仕事群を大きな粒群と小さな粒群に分けることを試みる。勿論、すべての処理を小さな粒の処理にすることができれば理想的であるが、小さな粒の処理にすることによる手間や、粒が増えることによる通信量の増加が大きいものについては、すべての処理を小さな粒にすることはできない。そこで、すべての処理を小さな粒にするのではなく、大きな粒と小さな粒とに分けることを試みるのである。そして、小さな粒が大きな粒の仕事の終了時刻のバラツキを完全に埋めるかどうかを、上の不等式より判定する。もし、不等式を満足しないならば、小さな粒の仕事を増やすように、仕事の分け方を再考する。

また、小さな粒の仕事群が、大きな粒の仕事の終了時刻のバラツキを完全に埋めても、小さな粒の仕事の終了時刻

表 4: 実測に使ったデータの大きさ (msec)

最大粒の大きさ					
問題図	t1	t2	t3	t4	t5
大きさ	7012	8503	4568	7807	16527
小さい粒の仕事の総量					
問題図	t1	t2	t3	t4	t5
大きさ	14548	15414	14873	16101	22562

表 5: 最大粒の大きさ × (プロセッサ台数 −1)(msec)

問題図	t1	t2	t3	t4	t5
2 PE	7012	8503	4568	7807	16527
4 PE	21036	25509	13704	23421	49581
8 PE	49084	59521	31976	54649	115689
16 PE	105180	127545	68520	117105	247905
32 PE	217372	263593	141608	242017	512337

のバラツキにより稼働率が 100% にはならない。従って、小さな粒の大きさも期待する稼働率を満足するように調節する。

このように、実行前に並列実行される処理の大きさが前もって見積もれ、処理の粒を大きさによって分けることができるならば、我々の方法は稼働率を上げる有効な手段となるであろう。

4.6 深さ優先探索への適用の試み

今回の実験のように処理する仕事の粒の大きさがあらかじめわかるものは、実際にはあまり存在しないかも知れない。しかし、次のような並列の深さ優先探索では以下のように工夫すれば、大粒度と小粒度の仕事ができ、しかも大粒度の仕事を先に実行してもかまわないと思われるので、我々の方法が適用できる。

> [問題] : あるレベルまではマスタープロセッサが実行し、それより深いレベルの枝をその他のプロセッサに配って並列実行する並列深さ優先探索
>
> [適用の仕方] : 左の枝は浅いレベルで、右の枝は深いレベルで切り、その先をプロセッサに配る。これによって左の枝は大きい粒の仕事に、右の枝は小さい仕事の粒になる。

他の問題に対しても、少し工夫することによって、我々の方法が適用できる可能性があると思われるので、負荷の均等化のための一つの方法となるだろう。

5 今後の予定 - 並列知識処理の新しい試み

我々は現在、表示以外の部分を全て KL1 化し、逐次版よりもさらに多くの処理を取り込んだ並列本格的な並列版「碁世代」を作成中である。この中で我々は、囲碁の対局というリアルタイム性を要求されるシステムにおける並列知識処理の一つのアプローチとして、「遊軍処理方式」を

提案している。ここに、その遊軍処理方式を取り入れたシステムの設計を紹介する。

遊軍処理方式とは、処理をその性質から2種類に分け、重要かつ処理の小さい仕事には高いプライオリティを付け、重要ではあるがもしその結果がなくても何とかなる大きな処理には低いプライオリティを付けてプロセッサに投げて実行させるものである。我々は、前者の仕事を実行するクライアントを本軍と呼び、その仕事を本軍ジョブと呼ぶ。また、後者の仕事を実行するクライアントを遊軍と呼び、その仕事を遊軍ジョブと呼ぶ。囲碁の対局に際しては、必要な処理及び最低限の囲碁の強さを保証する処理と、さらに強くなるには必要だが多くの時間を要するので一手にかかる時間内にいい結果が得られなくてもしょうがない処理の2つの処理に分けて考えることができる。並列版「碁世代」では、前者を本軍ジョブ、後者を遊軍ジョブとする。

一つのプロセッサには、本軍と遊軍が一つづつ配置される。本軍ジョブは高いプライオリティで、遊軍は低いプライオリティで実行される。本軍ジョブの個数は、囲碁を打つという一連の処理の流れの中で増減する。減った時点で、遊軍ジョブが動き出すために、暇なプロセッサがなくなり台数効果が上がる。そのうえ、本軍ジョブの実行が優先され遊軍ジョブの実行結果を待たずに対局を進められるので、多くの機能を取り込んでいるが一手にかかる処理時間は増えない。(図10)

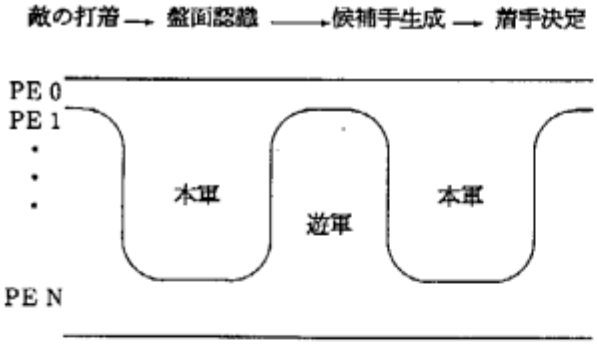


図 10: 本軍ジョブと遊軍ジョブの処理割合予想

処理の流れは以下の通りである。(図11)

1. 自分の番 (相手着手入力後)

本軍サーバーは、

・相手着手を各本軍クライアントに通信し、盤面の更新を行なわせる。
・相手着手を遊軍サーバーに通信する。
・本軍ジョブと遊軍ジョブを生成し、本軍クライアントには本軍ジョブを、遊軍サーバーには遊軍ジョブを送る。
・本軍ジョブがすべて終了し結果がクライアントから返ってきたら、遊軍サーバーからそれまでに返ってきた処理結果と合わせて、次の着手を決める。

遊軍サーバーは、

・相手着手を各遊軍クライアントに通信し、盤面の更新を行なわせる。
・遊軍ジョブを、遊軍クライアントに送る。
・遊軍ジョブで処理が終ったものは順次、本軍サーバーにその結果を通信する。

各遊軍用クライアントは、

・相手の着手位置が、自分が実行しているジョブの処理結果に影響を与えるような位置ならば、処理の実行をやめる。そうでないなら、実行を続け、処理が終ったら盤面の更新を行なう。

2. 相手考慮中 (自分の着手決定後)

自分の着手を決定するまでの間に終らなかった遊軍ジョブの実行が引続き行なわれる。

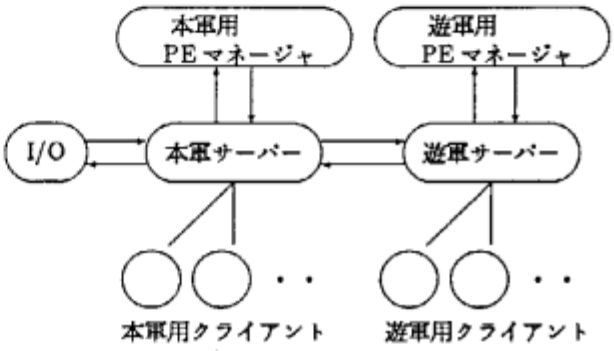


図 11: 遊軍処理方式を取り込んだシステム構成

6 むすび

既に逐次のシステムが存在しそれを並列化する場合、処理の一部を取り出して並列化し、残った逐次プログラムと通信することによって全体として起動する形のシステムを仲介にすることは、並列化した部分の評価を行ないながら並列システムの作成ができるので、開発の手順としては望ましいと思われる。

また、仕事の大きさによって実行順序を制御する方式は稼働率を上げる効果があることがわかった。

今後は、前述した遊軍方式を取り入れるなどして、囲碁システム「碁世代」をさらに強くしていくつもりである。最終的には、アマチュアの中級レベルまで到達させることを目標としている。

7 謝辞

本研究の機会をいただいたICOTの渕一博所長、内田俊一研究部長、新田克巳第7研究室室長に感謝します。「碁世代」の研究開発の当初より御指導を頂いている実近憲昭氏(AI言語研究所所長)に感謝します。また、助言を頂いたICOT7研の市吉伸行氏、CGSタスクグループの皆様に感謝します。

参考文献

[1] Chikayama,T. and Kimura,Y. : Multiple Reference Management in Flat GHC, In Proceedings of ICLP'87, pp. 276-293, 1987

[2] 古市 昌一, 他 : 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, Proceeding of KL1 Programing Workshop '90, pp 1-9, 1990

[3] Goto,A. et al : Overview of the Parallel Inference Machine Architecture(PIM), Proceedings of FGCS'88, Vol.1, pp. 208-229, 1988

[4] 沖 廣明, 他 : マルチ PSI における詰め碁プログラムの実現と評価, JSPP'89, pp 351-357, 1989

[5] 実近 憲昭, 他 : 囲碁システム「碁世代の方法」, ICOT 研究速報 TM-618, 1988

[6] Sanechika,N. : "Go Generation" A Go Playing System, ICOT Technical Report TR-545, 1990

[7] Taki,K. : "The parallel software research and development tool: Multi-PSI system", Programming of Future Generation Computers, North-Holland, pp. 411-426, 1988

Iterative-Deepening A* の並列化とその評価 *

和田 正寛　　市吉 伸行 †

(財) 新世代コンピュータ技術開発機構 ‡

E-mail: mwada@icot.or.jp

概要．探索問題は多くの問題に含まれる問題であり、様々な解法が知られている。その一つに Iterative-Deepening A* がある。この探索手法をさらに並列化によって計算時間を縮めるアプローチを実験している。マルチレベル動的負荷バランサーを利用して並列化を行っているが、処理の粒度が一定でないため、プロセッサに分配するタスクをいかにグループ化するかが大きな問題となってくる。負荷分散段数を変化させて実験し、マルチレベル負荷分散の与える影響を調べている。この探索モジュールをマルチＰＳＩ上に実現し、８パズルの問題を適用して評価を行ったところ、６４台プロセッサで約４０倍の台数効果が得られた。

1　はじめに

探索問題解決のための一手法として Iterative-Deepening A* というものが知られているが、この探索手法をさらに並列化によって計算時間を縮める実験を行なった。この実験システムをマルチＰＳＩ上で実現し、８パズルの問題を適用して評価を行ったところ、６４台プロセッサで約４０倍の台数効果を得ることが出来た。

2　探索アルゴリズム

探索問題を解くために用いられるアルゴリズムには様々なものがあるが、中でもよく知られているものは深さ優先探索と幅優先探索である。深さ優先探索とは、探索枝が複数に分かれている場合、任意の枝を選択して深さを深める方向に探索を続けていき、最深部に達しても解が見つからなかった場合は枝を遡って探索をやりなおすという戦略である。一方、幅優先探索とは、分岐している枝をその深さにおいて解であるかどうかをすべて調べ、解が見つからなかった場合深さを深めるという戦略である。これらの戦略には、それぞれ短所が存在する。

深さ優先探索の短所：

- 選択された枝が無限に伸びている場合、他の枝に解が存在しても発見できない。
- 解が見つかっても、それが最適解である保証がない。

幅優先探索の短所：

- 探索履歴をすべて記憶していなければならないため 記憶のための資源を爆発的に消費する。

3　Depth-First Iterative-Deepening

深さ優先／幅優先の探索戦略を組み合わせ、短所を克服したのが Depth-First Iterative-Deepening（以後 ID と表す）である。この探索戦略のアルゴリズムは以下のようになる。

1: Depth=0 とする。
2: Top Node から深さ Depth まで
　深さ優先探索を行う。
3: 解が見つかれば終了。
4: 解が見つからなかったなら、その
　探索の履歴はすべて忘れる。

A Parallel Iterative-Deepening A and its evaluaton

†Masahiro WADA, Nobuyuki ICHIYOSHI

‡Institute for New Generation Computer Technology

5: Depth を 1 増やす。
6: 2: に戻る。

この戦略を取れば、たとえ枝が無限に伸びていたとしても解が存在すれば必ず発見でき、また最初に発見される解は必ず最も浅い所にある解である。そして探索動作としては深さ優先探索を繰り返すだけであるから、記憶のための資源消費は深さ優先探索で消費されるのと同じ程度ですむ。よって前述の問題点は回避されている。

しかし、同じ探索を繰り返す事から、探索時間が増大する恐れがある。そこで、探索時間がどの程度になるかを検討してみる。探索時間の量の目安としては、生成されるノードの数を用いる。各ノードから伸びる枝の数 branching factor の平均を b 、深さ d まで探索する場合、幅優先探索の生成するノード数は、

$$
\begin{aligned}
&1 + b + b^2 + b^3 + ... + b^d \\
&= (1 - b^{d+1}) \,/\, (1 - b) \\
&< (b^{d+1}) \,/\, (b - 1)
\end{aligned}
$$

となり、そのオーダーは b^d である。また、深さ優先探索では、探索空間が tree である場合、一般にオーダーは b^d である。
一方、ID の探索戦略では、深さ d まで探索した場合、深さ d のノードは b^d 個生成されている。深さ d-1 のノードは、この繰り返し探索の時に生成された b^{d-1} 個と、一回前の繰り返し探索の時に生成された b^{d-1} 個が生成されている。同様に深さ d-2 のノードは今回の b^{d-2} 、一回前の b^{d-2} 、二回前の b^{d-2} 個が生成されている。よって全部で、

$$b^d + 2b^{d-1} + 3b^{d-2} + ... + db$$

個となる。この式を変形すると、

$$
\begin{aligned}
&= b^d(1 + 2b^{-1} + 3b^{-2} + ... + db^{1-d}) \\
&= b^d(1 + 2x^1 + 3x^2 + ... + dx^{d-1}) \\
&\qquad \text{at } x = 1/b \\
&< b^d(1 + 2x^1 + 3x^2 + 4x^3 + ...) \\
&= b^d(1 - x)^{-2}
\end{aligned}
$$

となる。ここで $(1-x)^{-2}$ は d に対して constant である事から、生成されるノードの個数のオーダーは b^d であり、探索時間はオーダーが変わるほど長くなる事はない。

4 Iterative-Deepening と Heuristic Search の結合

ID のアルゴリズムは、探索深さを限った深さ優先探索を何度も繰り返し、繰り返しの度に深さを 1 段ずつ増加させていく。しかし、実際には、よりゴールに近いであろうと考えられる枝をより深くまで探索するという戦略を取った方が効率がいい事が経験的にわかっている。そこで枝刈りを行う為に、Heuristic Search の A* アルゴリズムを取り入れる。A* アルゴリズムとは、トータルコストの低い枝を優先して探索するという戦略である。トータルコストとしては、そのノードに達するまでに要したコスト g と、そのノードからゴールにいたるまでのコストの見積もり値 h の和を用いる。
具体的なアルゴリズムとしては、以下のようになる。

1: Threshold=0 とする。
2: Top Node から、
Threshold $\geq$ g + h である間、
深さ優先探索を行う。
3: 解が見つかれば終了。
4: 解が見つからなければ、深さ優先探索で
Threshold を越えた値を集め、
その中から最小値を選ぶ。
5: その最小値を新たな Threshold の値と
設定する。
6: 2: に戻る。

ただし、Threshold の値が不適当であると、最適解よりも先に別解が見つかってしまう恐れが有る。これでは ID の長所が削がれてしまう。探索枝にコストの差が無い場合は g の値は単位量ずつの単調増

加であるから、問題になるのは h の値である。しかし一般に、h の値を見積もり過ぎる（ゴールまでの実際の距離が a であるのに、a 以上であると見積もる）事がなければ、その問題は生じない事が保証されている [1]。

5 Iterative-Deepening A* の並列化

多くの探索問題では、探索空間は樹状に広がっていく。そこで、IDA* で探索を行ないながら、ある深さまで達した時にそこから先の探索作業を別のプロセッサに投げる事によって IDA* を並列化する事が出来る。KL1 言語を用いて Parallel-IDA* モジュールを記述し、並列化による IDA* の高速化の実験を行った。

P-IDA* の主な探索戦略は深さ優先探索から深さ優先探索を行うように探索動作を制御してやらなければならない。ただし現在マルチＰＳＩにおいては、KL1 は同一プロセッサ内では選択された述語が終了／失敗／サスペンドするまではその述語の実行が続くようにインプリメントされている。そのために特に制御してやらなくても深さ優先探索が実現できる。現在はその機構を利用して深さ優先の制御を行なっている。

P-IDA* モジュールの負荷分散機構としては、マルチレベル動的負荷バランサーを利用している [2]。動的負荷分散は以下のようにして実現している。

1: 負荷分散する深さを、リストで渡す。
2: 深さ優先探索を実行する。
3: 探索深さが、負荷分散深さに達した時：
　　A: 負荷分散を行い、新たに割り当てられたプロセッサ上で探索を継続する。
　　B: 元のプロセッサ上では、別の枝探索を開始する。

負荷分散深さがリストになっているのは、複数段で負荷分散を行うためである。

P-IDA* モジュールは、問題解決部とは完全に分離されている。そのため、問題解決部を取り換える事が出来、汎用性がある。

P-IDA* の内部では、探索の為に処理を行なっている状態から新たな状態を作り出したり、その状態のトータルコストを算出しなければならない。これらは外部の問題解決部を呼び出す事によって実現されるので、問題解決部側ではこのインターフェイスを準備しておく必要がある。

6 P-IDA* の実験結果

実験の対象問題としては、8パズルを用いた。問題も固定し、初期状態・ゴール状態とも同じもので、負荷分散深さだけを変えて解が求まるまでの時間の変化を調べた。8パズルの問題解決部としては、今回は特別な戦略は取っていない。すなわち、与えられた問題状態から、次に生成できる状態を無制限に生成している。そのため、一手前の状態にもどったり、探索途中で同じ状態が生成されてループになってしまったりしている。しかしこれらの問題は問題解決部で解消すべき問題である事から P-IDA* モジュールとしては対応していない。

問題としては、次のものを与えている（図１）。

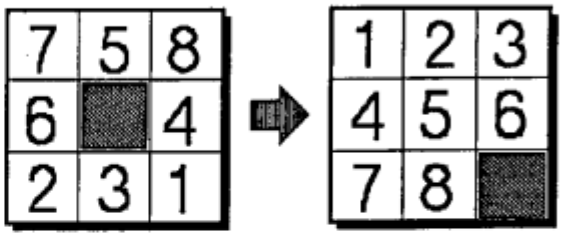


図１　実験に用いる８パズルの問題

この初期状態の場合、解までの深さは２８段（図５）、１プロセッサ上で解が求まるまでの時間は2,210秒である。

6.1　１レベル負荷分散の結果

負荷分散レベルを段々に深くして、実行時間の変化（図２）とパフォーマンスメータの動作を観察してみた。

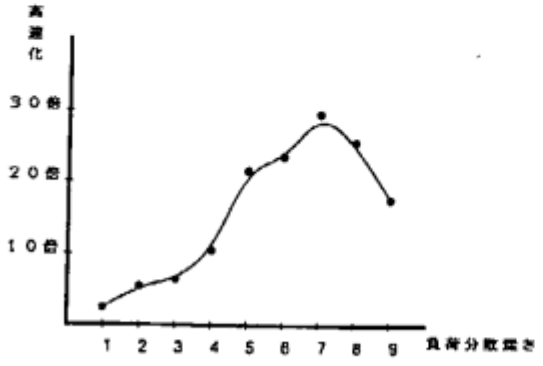


図２　１レベル負荷分散における
負荷分散深さと実行速度の変化

深さ１段では４台、２段では７台、３段では３２台のプロセッサしか動いていなかったが、これは探索木を描いてみたところ（図６）、各探索深さにおいては生成される状態数がそれぞれ深さ１段では４、２段では１２しかなく、プロセッサに割り当てられる部分問題の数が少な過ぎたためである。

負荷分散深さを４段にすると初めて６４台のプロセッサが全て稼働した。しかも全てのプロセッサが稼働状態がほぼ１００％で、プロセッサへの割り当てがうまくいったかのように思われた。しかし実際には高速化は約１１倍に留まっているが、これはA*アルゴリズムを取り入れているため各プロセッサの探索終了時期にばらつきがあり、次々にプロセッサが動作を終了し、最終的には全実行時間の１／３近くを２～３台のプロセッサだけが動いているという状態になっているためである。

負荷分散深さをさらに深くしていくと、さらに高速化が実現できた。これは各プロセッサに割り当てられる部分問題が増加したためである。そして７段の時に約２９倍の高速化を達成したが、それ以上負荷分散深さを深くすると効率が落ちている。これは、負荷分散深さを深くし過ぎたために、負荷分散を実行するまで稼働している最初の一台のプロセッサの負荷が大き過ぎるためである。実際、この時にパフォーマンスメータを見ると、全実行時間の半分近くを最初の一台のプロセッサだけが動いているという状態になっている。

6.2　２レベル負荷分散の結果

前述の、稼働しているプロセッサが徐々に減少していく問題を回避する他の方法として、残って動作しているプロセッサがさらに負荷分散を行う手段がある。この効果を検討するために２レベル負荷分散で実験を行った。（図３）

二回目の負荷分散深さ（一回目との深さの差）								
10	20.5							
9	29.1	18.9						
8	26.6	26.3	19.9					
7	34	32.5	25.4	19.7				
6	31.1	30.7	31.6	24.8	18.9			
5	18.4	29.5	31.1	30.3	26.3	18.3	13.2	
4	15.8	22.1	29.5	27.3	29.9	24.0	16.6	
3	9.5	13	22.3	26	33.0	26.3	19.7	
2	4.4	6.1	10.7	20.3	16.1	25.7	16.5	
1	3.5	4.9	6.3	8.6	11.9	10.4	12.5	8.0
	1	2	3	4	5	6	7	8

一回目の負荷分散深さ

図３．１　２レベル負荷分散における
負荷分散深さと実行速度の変化

２レベル負荷分散を行なって最も高速化出来たのは[1,7]の段数で負荷分散を行なった場合で、この時約３４倍ある。しかしこの場合も、１レベルの約２９倍に対して大幅な向上とは言い難い。これは、２レベルで負荷分散した場合も１レベルの時と同様に

- 二回目の負荷分散深さが浅い時は探索後半で稼働しているプロセッサが減少していく。
- 二回目の負荷分散深さが深い時はすべてのプロセッサに作業が割り当てられるまで時間がかかる。

という現象が生じているためである。

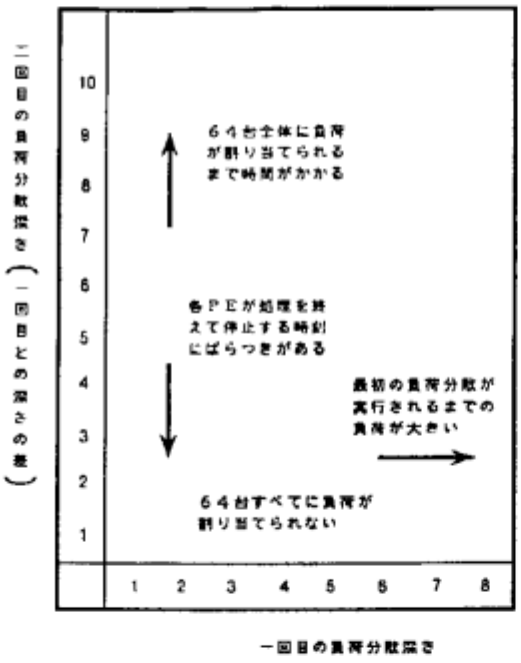


図３．２　２レベル負荷分散における
プロセッサ稼働状態の傾向

図４はプロセッサの稼働状態の変化のイメージ図である。

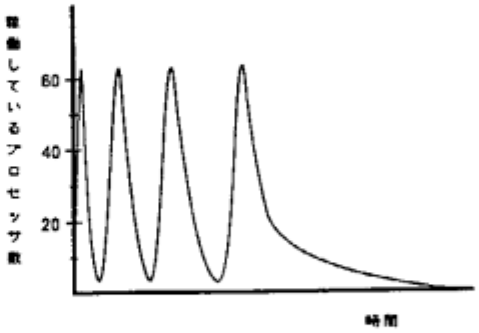


図４．１　２レベル負荷分散における
プロセッサの稼働状況
（２回目の負荷分散が浅い場合）

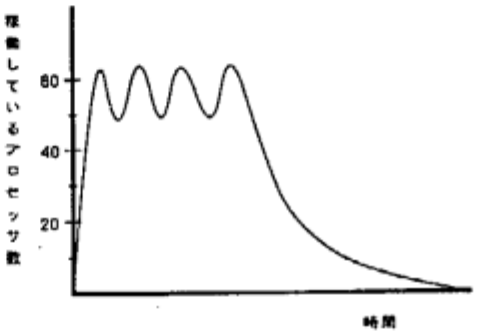


図４．２　２レベル負荷分散における
プロセッサの稼働状況
（２回目の負荷分散を少し深くした場合）

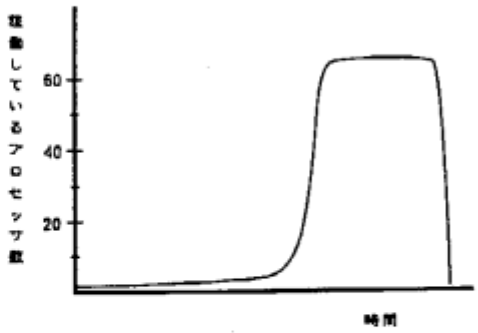


図４．３　２レベル負荷分散における
プロセッサの稼働状況
（２回目の負荷分散を深くした場合）

6.3 3レベル以上の負荷分散の結果

2レベル負荷分散では高速化は約34倍に留まった。しかしプロセッサの稼働状態を見てみると、仕事をしていないプロセッサはまだたくさん残っている。これらのプロセッサに負荷を分散してやればまだ高速化は可能である。そこでさらに負荷分散レベルを増やし、さらなる高速化を目指した。

3レベル以上ともなると、実験する負荷分散レベルの組み合わせが増大する。2レベルまでのように一つ一つ調べている余裕がなかったので、かなり試行錯誤的に段数を変えて実験を行なった。
その結果、[1,4,9,15]の組み合わせで実験を行なった時、約40倍の高速化を達成した。この時のプロセッサの稼働状態を見てみると、実行後すぐに64台すべてに作業が分配され、途中で停止するプロセッサもほとんどなく、終了も64台ほぼ同時に終了するという望み通りの状態を示していた。ただし、試行錯誤的に見つけた組み合わせであるから、より効率の良い組み合わせが存在する可能性は十分ある。

7 まとめ

IDA* の並列動作を実現し、64台の並列マシンを用いて8パズルの一問題を解き、1台のプロセッサでの動作に比べて約40倍の高速化を実現する事が出来た。今後は、効率の良い負荷分散段数を見つけるための手助けとするために探索木の特徴分析を行ないたい。また、このP-IDA* モジュールと組み合わせるのに適した負荷分散方式を検討したい。

8 謝辞

本研究を行うにいたり、有益な助言を頂いたKL1－TGの諸氏に感謝いたします。

9 参考文献

[1] Richard E. Korf,
"Depth-First Iterative-Deepening: An Optimal Admissible Tree Search", Proc. 27th Artificial Intelligence, pp.97-109, 1985.

[2] ICOT PIMOS 開発グループ,
"負荷バランスユーティリティ使用説明書 マルチレベル動的負荷バランサー （第1.0版）",
July 1990.

{7,5,8,6,0,4,2,3,1},{7,5,8,0,6,4,2,3,1},
{0,5,8,7,6,4,2,3,1},{5,0,8,7,6,4,2,3,1},
{5,6,8,7,0,4,2,3,1},{5,6,8,7,3,4,2,0,1},
{5,6,8,7,3,4,2,1,0},{5,6,8,7,3,0,2,1,4},
{5,6,0,7,3,8,2,1,4},{5,0,6,7,3,8,2,1,4},
{5,3,6,7,0,8,2,1,4},{5,3,6,0,7,8,2,1,4},
{5,3,6,2,7,8,0,1,4},{5,3,6,2,7,8,1,0,4},
{5,3,6,2,7,8,1,4,0},{5,3,6,2,7,0,1,4,8},
{5,3,0,2,7,6,1,4,8},{5,0,3,2,7,6,1,4,8},
{0,5,3,2,7,6,1,4,8},{2,5,3,0,7,6,1,4,8},
{2,5,3,1,7,6,0,4,8},{2,5,3,1,7,6,4,0,8},
{2,5,3,1,0,6,4,7,8},{2,0,3,1,5,6,4,7,8},
{0,2,3,1,5,6,4,7,8},{1,2,3,0,5,6,4,7,8},
{1,2,3,4,5,6,0,7,8},{1,2,3,4,5,6,7,0,8},
{1,2,3,4,5,6,7,8,0}

図5 初期状態からゴール状態までの解の経路

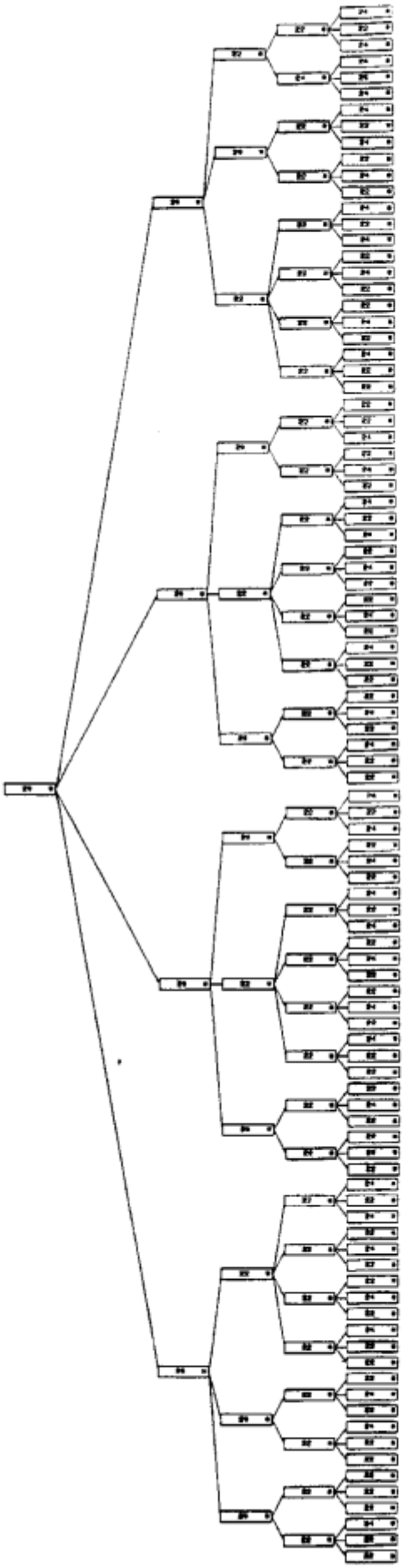


図6　初期状態から深さ4までの
探索木の状態

疎結合マシン上での確率的アルゴリズムの並列化

岩山 登, 佐藤 健
(財) 新世代コンピュータ技術開発機構

1 はじめに

並列計算機を用いて処理速度の向上を目指すためには, プロセッサの稼働率が高いことが望まれる. プロセッサの稼働率を高くするには, 計算の負荷を均等にプロセッサに割り付けなくてはならない. 一般に負荷の分散方法は, タスクの末端手部分を暇なプロセッサに割り当てていく動的な方法 (例えば [1, 2]) と, プロセッサは始めに与えられたタスクだけを計算する静的な方法 (例えば [3]) の, 二つに分けられる. 静的な方法は, タスクの大きさやタスク間の通信量などが予測できる場合には有効であるが, それは一般には難しい. 動的な方法では, 計算しようとしている問題と並列計算機のアーキテクチャごとに, 通信のオーバーヘッドが大きくならないような負荷分散戦略を工夫してやる必要がある. 特に, 疎結合計算機では通信のコストが計算のコストに比べ高いので, 通信量を少なく保ったままいかに負荷を分散するかが, 並列化の利益を享受するための鍵である.

本稿では, 疎結合並列計算機で探索問題の単解を求める場合について, 上の 2 つの負荷分散方法とは異なる方法を提案する. それは探索問題の確率的アルゴリズムの並列化である. この方法は計算の途中ではまったく通信する必要がなく, すべてのプロセッサが常に稼働しているという特徴を持つ. この方法によって, 静的負荷分散を行うとタスクを均等に分割できず, 疎結合並列計算機で動的負荷分散を行うとそのコストが大きくなる問題に対して, ある程度の並列性を得ることができる.

以下では, 2 つの確率的アルゴリズムの並列化手法について述べる (2 節, 3 節). 4 節では, 我々の方法とバックトラック法の確率的並列化との比較を行い, 5 節でまとめと課題について述べる.

2 確率的アルゴリズムと疎結合計算機での並列化 (その 1)

疎結合並列計算機で問題を解くとき, 通信のコストは計算のコストに比べ高いので, なるべく通信量の少ない方法で負荷を分散しなくてはならない.

切り分けたタスクを割り付けるための通信を, 計算の始めに行なうだけでよい静的負荷分散法は, まず試すべき方法である. しかし, タスクの大きさを事前に予測し均等なサブタスクに分割しなければ, プロセッサの稼働率は低くなってしまう. 一般にはタスクの大きさなどを予測することは難しいので, 静的負荷分散法ではうまくいかないことが多い. そういった場合, 動的負荷分散法が次に試される. 計算量に対して通信量がずっと少なくて済むような場合はよいが, そうでない場合, 通信コストがかさみプロセッサの稼働率が下がってしまう. このような従来の負荷分散方法ではうまくいかなかった問題に対しても, 並列計算によって処理時間を早くしたいという要請がある.

探索問題で単解を求める場合, 全てのプロセッサがそれぞれ独立に問題を解くが, プロセッサごとに解き方が違えば, 一番早く解を見つけたプロセッサで掛かった時間で問題が解けることになる. この方法では, 各プロセッサの仕事は全く独立なので, 一度問題をすべてのプロセッサにコ

ピーした後は，計算の途中では通信をする必要がまったくなく，通信コストの面からみると疎結合並列計算機に適している．この方法を具体化したのが，本稿で提案する確率的アルゴリズムの並列化である．

ここでいう確率的アルゴリズムとは，非決定的アルゴリズムの探索の途中で取り得る計算のパスが複数あるとき，そのうちの一つをランダムに選ぶアルゴリズムである．もし選んだ計算のパスが失敗であることが分かったら，初期状態からもう一度ランダムに選び直し，1 つ解が見つかるまでこれを繰り返す．これは，ナイーブな逐次探索アルゴリズムで何らの工夫も必要ない．

次に，我々の提案する確率的アルゴリズムの並列化について説明する．確率的アルゴリズムの並列化は，各プロセッサでは確率的アルゴリズムを用いて同じ問題を計算することによる．各プロセッサで異なる乱数発生器を用いることで，プロセッサごとに違った解き方をさせることになる．1 つ解を見つけるのが目的なので，あるプロセッサで解が見つかったらそのことを他のプロセッサに知らせ計算を停止させる．

並列確率アルゴリズムは，初期条件を各プロセッサにコピーした後は，いずれかのプロセッサで解が見つかるまでまったく通信する必要がない．また，それぞれのプロセッサは，仕事の割り当てを待ったり，仕事をやりつくすことはないので，常に稼働している．しかし，同じ探索パスを重複して選んでしまうかもしれないという欠点がある．そこで，確率的アルゴリズムの利害得失をみるために，確率的アルゴリズムの計算量について考える．

2.1 並列確率アルゴリズムの計算時間

確率的アルゴリズムの計算量を議論するとき，同じ入力に対しても計算する度にその時間は異なる可能性があるので，期待値としての計算時間を考えなくてはいけない．以下では，いくつかの仮定をおいて確率的アルゴリズムの期待計算時間を計算してみる．

1 つの探索パスに解が存在する確率を p とする．1 プロセッサで 1 解を得るまでの試行回数 (解を得るまでに失敗した回数 +1) の期待値は $1/p$ であり，1 試行に要する時間 (t) がパスによらず一定であるとすれば，1 解を求めるのに要する期待時間量 T_1 は t/p である．n プロセッサを用いたとき少なくとも 1 つのプロセッサで解が得られる確率 $p_n = 1-(1-p)^n$ であるから，n プロセッサで 1 解を得るまでの試行回数の期待値は

$$\frac{1}{1-(1-p)^n}$$

で，1 解を求めるのに要する期待計算時間量 T_n は

$$\frac{t}{1-(1-p)^n}$$

となる．すると台数効果は，

$$\frac{T_1}{T_n} = \frac{1-(1-p)^n}{p}$$

となる．よって

$$\lim_{p\to 0}\frac{T_1}{T_n} = n.$$

これは，問題が非常に難しく探索空間が大きい場合，n 台のプロセッサで重複なく探索できることを表している．また，

$$\lim_{p\to 1}\frac{T_1}{T_n} = 1$$

で，問題が簡単で解がたくさん存在する場合はすぐに解が見つかってしまうので台数効果は上がらない．

さらに

$$\frac{T_n}{T_{2n}} = \frac{1-(1-p)^{2n}}{1-(1-p)^n} = 1+(1-p)^n$$

でプロセッサ数を2倍にしてもスピードアップは $1+(1-p)^n$ 倍にしかならず，プロセッサを増やすと選択の重複が多くなることを示している．

2.2 マルチPSIによる実験

さて，我々は並列確率アルゴリズムの有効性を確認するため，Doyle の Truth Maintenace System [4] を取り上げた．TMS は，信念間の制約を justification と呼ばれるルールで表し，justification に矛盾しない信念の状態を導く処理系である．TMS は仮説推論や非単調推論などに用いられ，多くの研究がなされている．過去に提案された TMS の並列アルゴリズム [5, 6] はアーキテクチャとして密結合並列計算機を暗に仮定しており，通信量が多く疎結合並列計算機での実現には向いていない．

我々は以前提案した TMS の非決定的アルゴリズム [7] を確率的アルゴリズムを用いてマルチPSI上に実現した．TMS でクィーン問題を記述し16台のプロセッサを使って実験を行った．その台数効果の理論値と実験値を図に示す．なお，実験には8および11クィーン問題を用い，期待値を調べるため一定回数求解を繰り返した．また，p を理論的に求めるのは容易でないので，理論値の計算には実験的に求めた値を用いた．

11クィーン問題の場合，理論値と実験値はほぼ一致している．8クィーン問題では，理論値としての台数効果も16台で5倍程度であるが，実験値ではそれを下回っている．そこで，解が得られるまでの平均試行回数を調べたところ，8クィーンの場合10～16プロセッサを用いたとき，1.03～1.17であった．これは，10台以上プロセッサを用いた場合いずれかのプロセッサで失敗なしに解が得られていることを意味し，10台以上プロセッサを増やしても計算時間は短くならない．

3 確率的アルゴリズムの並列化 (その2)

前節で述べた並列確率アルゴリズムでは各プロセッサはそれぞれ独立に探索空間全体を対象として計算を行っている。このため，すでに他のプロセッサで調べられた探索パスを繰り返し選んでしまう可能性があり，前節の並列化では無駄な計算が多くなっていると予想される．そこで，前節と同じ探索問題の単解を求める場合について，従来の静的負荷分散法と確率的アルゴリズムを組み合わせた計算方法を提案し，前節の並列確率アルゴリズムと比較する．

それは，与えられた問題が幾つかの部分探索空間に分けられる場合に，それぞれの部分探索空間にプロセッサを割り当て，プロセッサは割り当てられた探索空間だけを確率的に計算する方法(以下では，分割方式とよぶ)である．

例えていうなら，10部屋ある家のどこかに落とした針を10人で探すことを考えよう．前節の方法(非分割方式)は10人が10人とも家全体を探すことに対応し，分割方式はそれぞれの人が別々の部屋を1部屋ずつ探すことに対応する．分割方式と非分割方式の違いは，プロセッサが1部屋だけを探すか家全体を探すかだけで，その他の部分に違いはない．両者とも，失敗したら初期状態からやりなおし，あるプロセッサで解が見つかったら他のプロセッサを停止させ，単解を求めている．分割方式では，解の存在する確率の高い部分探索空間を割り当てられたプロセッサが早く解

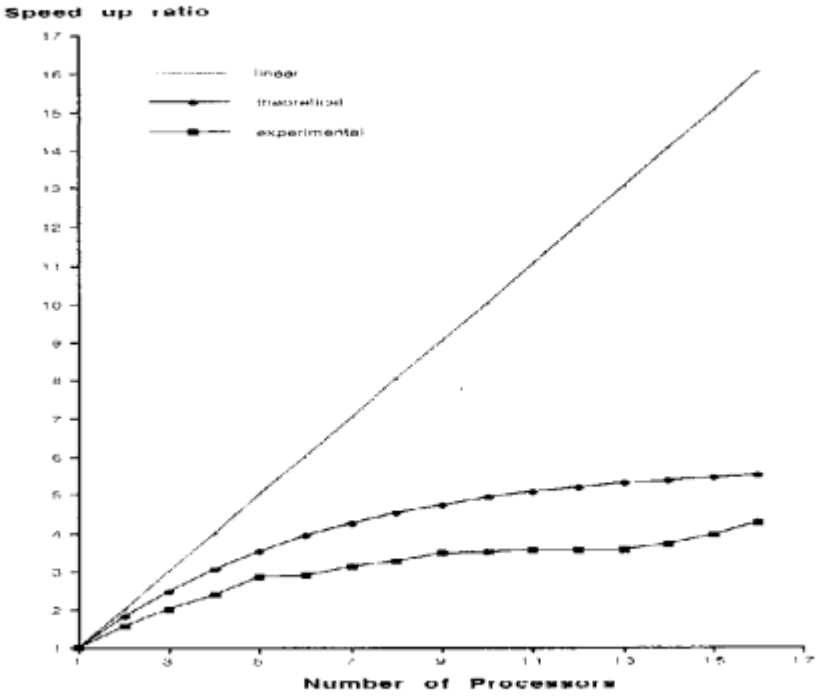


図 1: 8queen 問題の台数効果

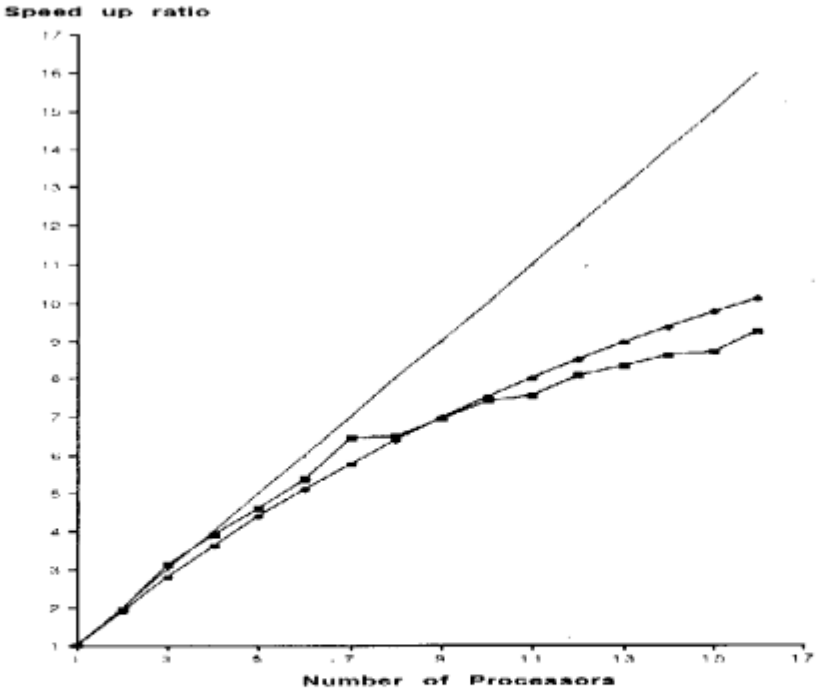


図 2: 11queen 問題の台数効果

を見つけることができるので，全体としての計算時間が短くなることを期待した。しかし，以下に示すように分割非分割のいずれがよいとも言えないことがわかった。

3.1 分割，非分割方式の計算時間の比較

以下では，前節のアルゴリズム (非分割方式) と分割方式の期待計算時間の比較を行う．解くべき問題は n 個の部分探索空間 $S_1, \ldots, S_n$ に分割できるとし，それぞれの部分探索空間における 1 探索パスに解が存在する確率を $q_1, \ldots, q_n$ とする．

非分割方式で，n プロセッサを用いたとき少なくとも 1 つのプロセッサが解を見つける確率は $Pr_1 = 1-(1-p)^n$ であった．ここで，ランダムに 1 つ探索パスを選ぶとき $S_1, \ldots, S_n$ からそれぞれ $\alpha_i = \frac{1}{n} + \varepsilon_i$(ただし，$-\frac{1}{n} \leq \varepsilon_i \leq 1 - \frac{1}{n}$, $\varepsilon_1 + \cdots + \varepsilon_n = 0$) の確率で選ぶとすれば，

$$\begin{aligned} p &= \sum_{i=1}^{n} \alpha_i q_i \\ &= \frac{1}{n} \sum q_i + \sum \varepsilon_i q_i. \end{aligned}$$

よって，

$$Pr_1 = 1 - (1 - \sum_{i=1}^{n} \alpha_i q_i)^n$$

と書け，1 解を求めるのに要する期待計算時間は

$$T_{nodiv} = \frac{t}{Pr_1}$$

となる．

分割方式では，各プロセッサ i では部分探索空間 S_i だけを探索する．分割方式で，n プロセッサを用いたとき少なくとも 1 つのプロセッサが解を見つける確率は

$$Pr_2 = 1 - \prod_{i=1}^{n} (1 - q_i)$$

となり，1 解を求めるのに要する期待計算時間は

$$T_{div} = \frac{t}{Pr_2}$$

となる．

さて，どちらの期待計算時間が早いであろうか．

$$\begin{aligned} Pr_2 - Pr_1 &= (1 - \sum_{i=1}^{n} \alpha_i q_i)^n - \prod_{i=1}^{n} (1 - q_i) \\ &= (\frac{1}{n} \sum (1 - q_i) - \sum \varepsilon_i q_i)^n - \prod (1 - q_i). \end{aligned}$$

ここで $1 - q_i \geq 0$ だから，相加平均 $\geq$ 相乗平均より

$$\frac{1}{n} \sum (1 - q_i) \geq \prod (1 - q_i)^{\frac{1}{n}}$$

PEs	分割方式	非分割方式
8	334	341
4	464	458
2	700	698
1		1157

表 1: 8queen 問題の合計計算時間 (sec)

である.
$\sum \varepsilon_i q_i \leq 0$ あるいは $0 \leq \sum \varepsilon_i q_i \leq \frac{1}{n}\sum(1-q_i) - \prod(1-q_i)^{\frac{1}{n}}$ のとき,

$$Pr_2 \geq Pr_1.$$

なぜならば

$$\frac{1}{n}\sum(1-q_i) - \sum \varepsilon_i q_i \geq \prod(1-q_i)^{\frac{1}{n}}.$$

$0 \leq \frac{1}{n}\sum(1-q_i) - \prod(1-q_i)^{\frac{1}{n}} \leq \sum \varepsilon_i q_i$ のとき,

$$Pr_1 \geq Pr_2.$$

なぜならば

$$\frac{1}{n}\sum(1-q_i) - \sum \varepsilon_i q_i \leq \prod(1-q_i)^{\frac{1}{n}}.$$

特に, $\varepsilon_i = 0$ のとき, すなわち $\sum \varepsilon_i q_i = 0$ のときは, $Pr_2 > Pr_1$ である. このとき, $\alpha_i = \frac{1}{n}$ となっていることを注意しておく.

以上の議論から, $\sum \varepsilon_i q_i$ の大小によって Pr_1 と Pr_2 の大小が決まり, その確率の大きい方が期待計算時間が短くなることがわかる. しかし分割方式の方が一概に良いとはいえない. これは $\sum \varepsilon_i q_i$ の値が大きいときは, ある部分探索空間に解が存在する確率が非常に高いことを示しており, その部分探索空間について 1 つのプロセッサだけが計算するよりも, すべてのプロセッサがその部分空間を探索する可能性のある非分割方式のほうが, 全体として解を見つける確率は高くなると考えられるからである.

3.2 分割方式の実験

我々は, 前節で実験に用いた TMS で 8 クィーン問題を記述しマルチ PSI で実験した結果, その計算時間は 300 回の合計でまったく差が出なかった (表 1).

実験では, 非分割方式の場合, プロセッサはすべて 1 回目の選択で, $\{1\times1, 1\times2, 1\times3, 1\times4, 1\times5, 1\times6, 1\times7, 1\times8\}$ のクィーンの位置の中からランダムに選ぶ. 分割方式の場合, 2 プロセッサの場合, 1 回目の選択で $\{1\times1, 1\times2, 1\times3, 1\times4\}$ からランダムに選ぶ問題と, $\{1\times5, 1\times6, 1\times7, 1\times8\}$ からランダムに選ぶ問題に分けた. 同様に 4 プロセッサの場合, 1 回目のランダムな選択をそれぞれ 2 つの候補の中から行う 4 つの問題に分けた. それぞれの問題の 1 回目の選択の候補集合は, $\{1\times1, 1\times2\}$, $\{1\times3, 1\times4\}$, $\{1\times5, 1\times6\}$, $\{1\times7, 1\times8\}$ である. 8 プロセッサの場合, それぞれ $1\times1, 1\times2, \ldots, 1\times8$ の位置にクィーンが置かれた状態から始める 8 通りの問題に分けた[1]. このとき, $\varepsilon_i = 0$ となっている.

[1] この問題の分割の仕方は, [3] の OR 並列探索における静的な負荷分散法に似ている.

この実験で分割,非分割によらず計算時間が同じであったのは,探索空間の分割の仕方から$\varepsilon_i = 0$であったのと,クィーン問題の特徴から$q_1 = \cdots = q_n$であったため,$Pr_1 = Pr_2$となったからだと考えられる.今後,ε_iとq_iについての条件を変えて実験を行う必要がある.

4 バックトラック法の確率的並列化との比較

これまでに述べた方法は,分割方式,非分割方式を問わず,失敗したら初期状態から計算をやり直していた.[8, 9]では,各プロセッサが同じ問題について選択肢をランダムに選ぶことによって計算するという点では非分割方式と同じであるが,失敗したときに初期状態に戻るのではなく,一番近いチョイスポイントにバックトラックし,まだ選ばれていない選択肢の中からランダムに選ぶ,バックトラック法の確率的並列化について述べている.

単一プロセッサについて考えたとき,バックトラック法の場合,計算が失敗したとき次の計算の候補はそれ以前の計算に依存し,選んだ部分木をサーチし尽くすまで別の部分木に進まないので,解の存在しない部分木を先に選んだとしたら無駄な計算をすることになる.我々の方法では,計算が失敗したとき次の計算の候補は探索木全体から選ばれるので,探索木を局所的に計算する可能性は低い.しかしながら,バックトラック法の場合,単一プロセッサ内では同じ探索パスが複数回選ばれることはないが,我々の方法では複数回選ばれる可能性がある.どちらに分があるかは問題によると考えられる.

また,[8, 9]の定式化の方法と我々のそれは異なっている.[8, 9]では,1プロセッサで解を得るのにかかる時間を確率変数としている.その確率変数の確率密度関数によって期待計算時間の台数効果は決まる.我々の考察の対象は試行回数であり,1試行に要する計算時間は一定であるとしていた.

5 おわりに

並列計算機で探索問題の単解を求める場合の,確率アルゴリズムによる並列化について,非分割方式と分割方式の2つの方法を提案し,期待計算時間と台数効果の理論的計算及びマルチPSIによる実験を行った.この方式は,計算の途中で通信する必要がまったくなく,通信コストの面からみると疎結合並列計算機に適しており,従来の負荷分散方式ではうまくいかなかった問題に対してもある程度の並列性を得ることができる。また,この方式は,問題や並列計算機のアーキテクチャ,プロセッサ台数によらず用いることができるという副次的な効果も持っている。

今後取り組むべきこととして,以下のことが考えられる.

1. 我々の方法では,分割方式,非分割方式ともに選択肢の選択は平等であるが,これをより可能性の高い方を選び易くするように評価関数を導入する.
2. 非分割方式の場合,探索の重複を減らすためにプロセッサ間での何らかの情報を交換する.
3. 分割方式で$\sum \varepsilon_i q_i$の大小を計算するのは一般には容易ではないので,簡便な近似的計算について検討する.

謝辞

研究の機会を与えて下さったICOT渕所長,古川次長,長谷川部長代理,相場第4研究室長代理,また,討論ご助言頂いたICOTの方々,および,ミネソタ大学V. Kumar氏に感謝する.

参考文献

[1] Masuzawa, H. et al.: "Kabuwake" Parallel Inference Mechanism and Its Evaluation, *FJCC-86*, pp. 955　962 (1986).

[2] 古市, 瀧, 市吉: 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, *KL1 Programming Workshop '90*, pp. 1 – 9 (1990).

[3] Burg, B.: Parallel Forward Checking First Part, *ICOT TR-594*, (1991).

[4] Doyle, J.: A Truth Maintenance System, *Artificial Intelligence*, **12**, pp. 231 – 272 (1979).

[5] Petrie, C. J. Jr.: A Diffusing Computation for Truth Maintenance, *Proc. ICPP-86*, pp. 691 – 695 (1986).

[6] Fulcomer, R. M., Ball W. E.: Correct Parallel Status Assignment for the Reason Maintenance System, *Proc. IJCAI-89*, pp 30 – 35 (1989).

[7] Satoh, K., Iwayama, N.: Computing Abduction by Using the TMS, *to appear in ICLP-91*,(1991).

[8] Janakiram, V. K. et al.: Randomized Parallel Algorithms for Prolog Progrms and Backtracking Applications, *Proc. ICPP-87*, pp 278 –280 (1987).

[9] Ertel, W.: Random Competition: A Simple, but Efficient Method for Parallelizing Inference Systems, *Techniche Universität München, Report FKI-143-90*,(1990).

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

戸谷智之 星田昌紀 石川幹人 新田克己
(財) 新世代コンピュータ技術開発機構

1 はじめに

本方式はタンパクのアミノ酸配列の解析手法であるマルチプルアライメントを行う一方式として考案、実装された。本稿では、まず、配列解析手法の1つとして有名なマルチプルアライメントについて述べ、ダイナミックプログラミングによるアライメント、および、KL1による並列マルチプルアライメントシステムの実装について解説する。その後、高速化を目標にして、解析、システムのモデル化を行い、プロセッサへのマッピングとパフォーマンス特性の相関を得ることができ、実験と一致したので、ここにそれを示す。

2 マルチプルアライメント

近年、タンパクのアミノ酸配列の自動分析法が開発され、それらのデータを蓄えたデータベースが急速に膨らんでいる [Bilofsky 88]。それに比べて配列情報がいかなる意味を持つかを探る遺伝子情報処理技術は未熟であるため、データベースに蓄えられたデータが十分に利用されてはいない。遺伝子情報処理のうちで最も基本的な技術は、配列間の類似性を調べる解析処理である。我々は、そのなかでも、数個の配列にわたって共通の特徴を抽出するために用いる、マルチプルアライメントという解析処理に注目し、システム化を試みた。以下に、マルチプルアライメントについて解説する。

2つの配列を、類似する部分を同じ列に揃えて並べ合わせる操作をアライメントといい、3つ以上の配列を並べる操作をマルチプルアライメントという。たとえば、次のような性質の類似したタンパクのアミノ酸配列が3つあったとする。

```
M-MULV    LLDFLHQLTHLSFSKMKALLERSHSPYYMLNRDRTLKNITETCKACAQ
HTLV      VLQLSPAELHSFTHCGQTALTLQGATTTEASNILRSCHACRG
RSV       AYPLREAKDLHTALHIGPRALSKACNISMQQAREVVQTCPHCNS
```

ここで、左側の見出しが配列の名前で、右側の文字列がアミノ酸配列である。1つの文字が1つのアミノ酸に対応する。アミノ酸は20種類あり，20種のアルファベットで識別される。例えば、最下段左からA,Y,P,L,Rは、それぞれアラニン、チロシン、プロリン、ロイシン、アルギニンを意味している。

これをマルチプルアライメントすると、次のようになる。

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

配列のところどころにギャップ"-"を入れているが、これは、その部分に対応するアミノ酸がないことを示す。この例で、C**C （*は任意のアミノ酸を表す）などの共通文字が同じ列に並んでいるのがわかる。C**Cのように複数の文字が、複数の配列で共通になっている文字の組を配列モチーフと呼び、タンパクのうちの重要な部分を指し示していると判断される。この背景には、タンパクの配列のうち、機能的、構造的に重要である部分には遺伝的変異が起きにくいという進化論的考え方 [Kimura 86] がある。

マルチプルアライメントされた結果は次のように利用できる。第一に、先に述べたように、重要な配列の部分であるモチーフを見い出して、データベースから新たな類似配列を検索する助けとできる。第二に、類似した構造を持つ配列をマルチプルアライメントした結果から、共通の部分構造に対応するアミノ酸の性質の並びがわかり、その部分構造の形態を予測する助けとできる。第三に、マルチプルアライメントから進化系統樹を描くことができ、類似配

列の各々が、どのような遺伝的過程を経てきたかを推測することができる。このようにマルチプルアライメントは、遺伝子情報処理の基本技術と位置づけることができる。

タンパク中のアミノ酸はしばしば似た性質の別のアミノ酸に置き換わるので、異なる文字でもそれらが表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して処理を行う。しかしアミノ酸の性質には親水性、疎水性、極性、酸性、塩基性、大きさなど多数あり、その類似性評価も多数の方法がある。現在最も広く使われている類似性評価尺度は、Dayhoffマトリックス [Dayhoff 78] である。この尺度は、当時知られていたアライメントをもれなく調べ、同じ列に該当アミノ酸対が並んでいることが偶然に対して何如に少ないかを数値化したものである。数値は確率の対数値になっているため、それらの足し算は複合事象の共起確率を算出したことに相当する。本方式もこの評価尺度を使用している。

Dayhoffマトリックスのような評価尺度が与えられていれば、それを用いたアライメントの評価基準により決まる「最適なマルチプルアライメント」を求めるダイナミックプログラミング法が知られている [Needleman 70]。この方法では、アライメントの評価基準をアミノ酸の全組み合わせに対して、Dayhoff値およびDayhoff値をもとに設定したギャップコストの総和をとったものとして定義している。

3　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う特徴を持つ最適化問題を解くためのアルゴリズムのひとつである。これを用いて「最適なアライメント」を求めることができる。なお、以下に出てくるコストとは、アライメントにおいては先に述べた評価基準に対応するものである。

最適化問題が、複数の段階で決定を行い、その各段階における決定が直前の段階の決定にのみ依存するという形式に定式化できるとき、DPを用いると非常に効率良く最適解（コスト最小の決定経路）を求めることができる。もし、この種の問題を最初に全ての場合を尽くして、そのうちの最小のものを選ぶという解法で解いた場合、指数オーダーの計算量がかかる。しかしDPを用いると多項式オーダーで解を得ることができる。

アミノ酸配列のアライメントは、このDPを用いて行うことができる。簡単のために配列2本のアライメントについてDPの概念的説明を行う。説明のために図1を用いる。例えば、ADHE、AHIEという2つの配列をアライメントする場合、この2つの配列を図1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2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。こうして全てのアークにコストが割り当てられる。

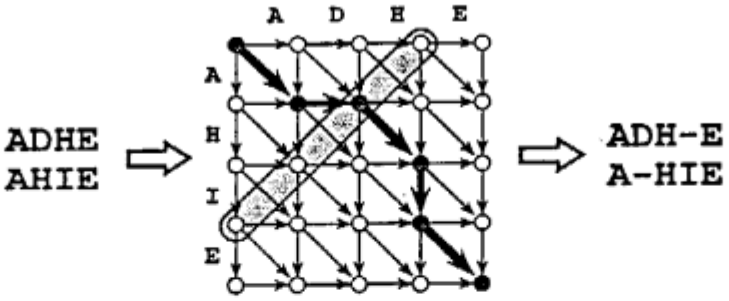


図1：2次元DPマッチングによるアライメント

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1の例では、太いアークで表された経路がコスト最小となる。この太いアークで表された経路を順に見ていくと、AとAが対応し、Dに対応するもう片方のアミノ酸はなく（つまりギャップが対応し）、HにはHが対応し ... という具合に解釈することができる。結果として図1の右側にあるアライメントが得られる。

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1の右上がり斜め線上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うことができる。具体的には、ある段階の各ノードの計算を行うためには、直前の段階の各ノード（2次元DPでは3つある）で求まった部分的最短経路のコストを参照して、今

求めたいノードに至るコストをそれぞれ計算し、このうちの最小値を求めてそれをそのノードに至るコストとすればよい。直前のどのノードを選択したかという情報も記憶しておく。この操作を最後まで繰り返せば、ネットワーク全体の最短経路を求めることができる。各ノードにおける計算は同時に実行可能な状態にあるものが存在する。このような最短経路の問題は並列に計算を行うことができる。

4　並列３次元ＤＰの意義

ダイナミックプログラミング法を用いれば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 [Murata1 85][Murata2 90] [Carrilo 88]。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う方法が試みられている [Waterman 86]。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。

我々はこの計算量の多い３次元 DP を、マルチ PSI 上で並列に動作させるプログラムを開発した。プロセッサ６４台で並列実行させたところ、パイプライン効果を含め、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、今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。

３本の最適なアライメントができても４本以上は無理なのだから、本質的に２本のときと変わらないと考えられがちであるが、そうではない。我々は (A,B,C) と (A,B,D) という２本の配列が重複する２つのアライメント結果をつなぎ合わせて、(A,B,C,D) というマルチプルアライメントを導くプログラムを研究、開発している。このプログラムにより、配列２本のアライメントをつなぎ合わせる方法に比べ、３次元ＤＰによるマルチプルアライメントの精度が飛躍的に向上することが確認されている。

すなわち、並列３次元 DP 手法は生物学的意義が大きく、マルチプルアライメントの自動化に大きな一歩を踏み出したものと評価できる。

5　ＫＬ１による並列３次元ＤＰの実装

ここでは並列化の基本的アイデア及び実装の方法を説明する。３次元ＤＰは図２のような３次元のネットワークとして表現できる。３次元 DP によるアライメントでは各段階が図２の点線で囲まれた面の領域に存在するノード群に対応する。また２次元 DP によるアライメントでは各ノードは直前の３つのノードと連結されていたが、３次元 DP では、図３のように、７つのノードと連結されている。

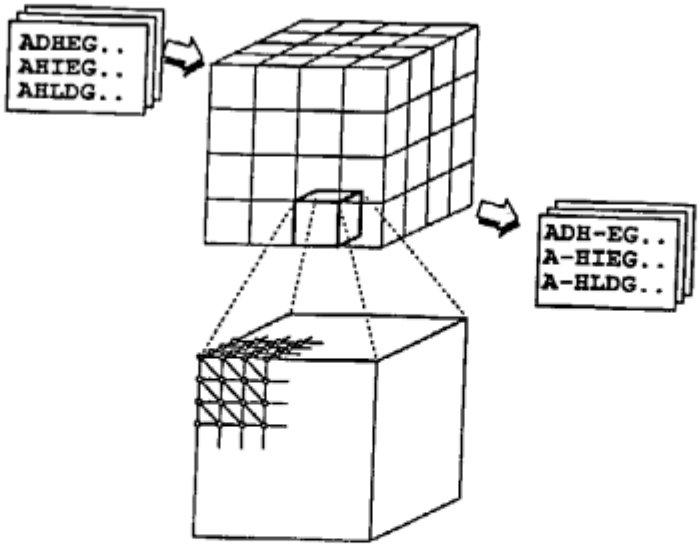


図２：３次元ＤＰによるアライメントとプロセッサへの割り当て

並列アルゴリズムを考えるとき、並列性の有無を考えるのは当然であるが、もうひとつ注意しなければならないのは、通信のオーバヘッドである。いくら並列性が高くても、プロセッサ間の通信のコストが並列性の効果を打ち消

すほど大きければ、全体として無意味なものになってしまう。幸いなことに、ここで議論しているDPは、ある段階から次の段階へ遷移するときのノード間の依存関係に局所性が存在するため、適切にプロセッサへの割り当てを行えば通信オーバヘッドを実際の計算に比べ相対的に低く抑えることができる。

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うプロセスを段階が1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うことによって全体の計算が進んでいくことになる。

次に3次元DPのプロセッサへの割り当てに関して述べる。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、並列実行を行うことが可能である。各段階の処理が、波面状にネットワークの中を進んでいくのである。

図2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が割り当てられた例が示されている。メッシュに分割された各領域は各プロセッサが対応している。各プロセッサにはノードに対応するプロセスが多数割つけられている。前にも述べたが、プロセスネットワーク自身もメッシュ構造を成しており、各ノードは近接するノードとのみ通信を行うので通信に局所性が存在する。そのため、もともとメッシュ状のプロセスネットワークを、このように相対的により大きいメッシュ状に切断しプロセッサに割り当てる方法を用いれば、通信のオーバヘッドを実際に意味のある計算に比べ相対的に小さくすることが可能である。また、この割り当て方法は、縦、横、高さ方向の切断数を容易に変えられる。

ただこの方法では、ある時点で波面を含まない部分のプロセッサは稼働しなくなってしまう。しかし、解くべき問題が複数ある場合には、それらを次々とネットワーク中に投入することが可能である。こうしたパイプライン的な並列性を導入すると、（最初と最後を除いて）プロセッサが稼働しなくなることはほとんどなくなり、プロセッサ全体の稼働率をさらに上昇させることができる。

整理すると、本並列処理方式には2種類の並列性が存在する。1つはDPの各段階におけるノード間の並列性で、各アライメントに対応する波面内の並列性ということができる。もう1つは複数のアライメントをパイプライン的に実行するときの並列性で波面間の並列性ということができる。

6 実験と解析

3次元DP法によるマルチプルアライメントは、その分野の要求としてパイプライン的に実行を行うことの意味は大きい。しかしここでは、より基本となる、プロセスネットワークをどのように分割すれば、ひとつの問題を解くのに最適なパフォーマンスが得られるかについてのみ解析を行った。そこで、処理系を分析して、処理モデルを構築し、得られたモデルが実際のパフォーマンスの特徴をよく表現していることを確認した。

6.1 モデルの構築

最もパフォーマンスの良いマッピングはどのようなものであるかを、解析を通して予測するために、モデルの構築を試みた。すでに最適経路問題についてこうした解析[Wada 90]がなされている。モデルを構築するにあたり、いくつかの仮定を行った。以下に具体的に行ったモデル作成について述べる。

まず、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。このような処理を全体の実行時間を律速する処理と呼ぶ。その処理の流れに着目できれば、処理時間は実際に処理を行っている時間とプロセス間の通信にかかっている時間とに分けることが可能である。ここで通信について仮定を設けた。プロセッサ内の通信は非常に小さいのに対して、プロセッサ間を渡るメッセージ通信は大きいので、プロセッサ間の通信のみを考慮するものとした。全体の実行を律速する処理のうち、プロセッサ間通信は、送り手側プロセスがメッセージをパケットに詰めて送る時間、データがプロセッサ間のネットワーク上を移動する時間、および受け手がパケットを解いてデータを読みとる時間から成る。ネットワーク上の移動時間は他の二つに比べて非常に小さいので無視したうえ、残りの二つが通信に要する時間は等しいと近似した。そして、この問題においては、すべてのプロセスから送られるメッセージはすべて、ほぼ均一なデータ型から成るので、この時間は一定の値（通信単位時間C）になるものと仮定した。

次に、プロセスネットワーク上のプロセスの実行がどのように伝わっていくかについて検討する。この実行にKL1処理系による実行の指向性が反映する。図3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7つのプロセスからの情報を全て受けとった時点で自分の計算を行い、次の7つのプロセスに情報を送る。送り先のプロセスを図のように、x、y、z、xy、xz、yz、xyz、とすると、プ

ログラム上では普通、この順にユニフィケーションが書かれる。すると、送り先のプロセスは実行可能ならば、ゴールスタックにこの順に積まれるため、実際の実行は逆順で行われる。図に書き込まれた数字はその順番を表している。

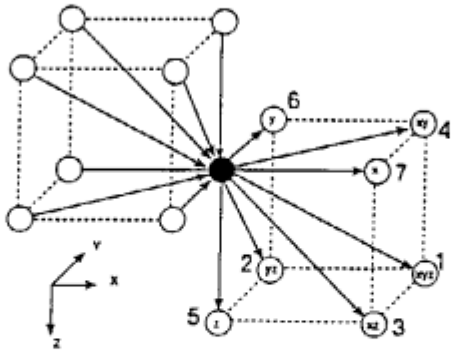


図３：プロセス間のメッセージ通信

しかし、xyz、yz、xz、xyのプロセスはさらにx、y、zのプロセスからのメッセージを待っているので、本プログラムでは、zのプロセスが最初に実行可能となる。つまり、zのプロセスに実行が移ってしまう。zは自分の処理を行った後、同様にメッセージ送信を行い、その後も同じ理由から、Z軸方向に隣接するプロセスが順次実行される。このように、１プロセッサ内では、まず、Z軸方向のプロセスが優先的に処理を行うこととなる。Z軸方向の最後のプロセスまで行き着くと、同様にメッセージを送るが、そこから先のZ軸方向は、他のプロセッサに処理がまかされる。このとき、処理を始めるためのメッセージが揃っているプロセスはxとyだけである。xとyのうち、スタックに後から積まれたyに実行が移る。yにおいても同様に、まずZ軸方向に処理は進む。Z軸方向の最後のプロセスまで行き着くと、yに対してY軸方向に隣り合うプロセスが、次に起動される。このようにZ軸方向への一連の処理の流れがY軸方向に進んでいくことになる。Y軸方向にプロセッサの最後までZ軸方向の一連の処理が終了して初めて、X軸方向に処理が移動する。つまり、これまでに述べたZ-Y平面に平行な処理の流れが、最後にX軸方向へ進むのである。各方向に３つずつプロセスがあるネットワークが１プロセッサに割り当てられる場合、プロセッサ内の実行順序を図４に例示しておく。丸がプロセスを表し、数字は実行順序を表している。

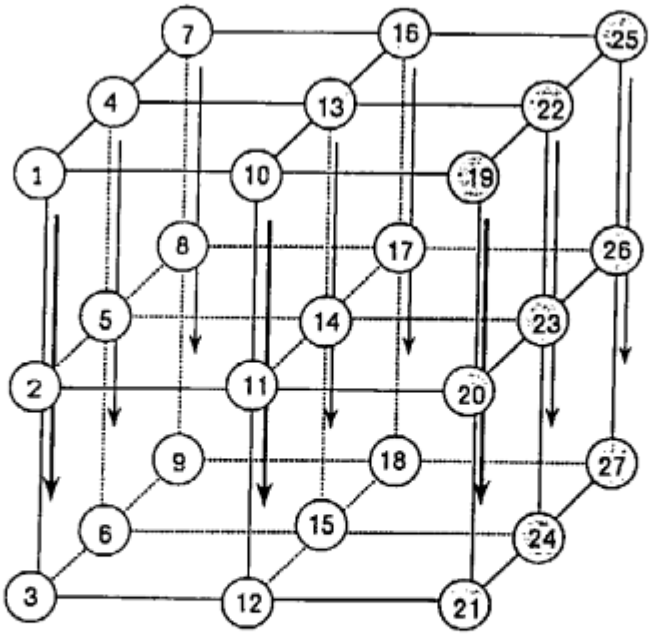


図４：プロセッサ内の実行順序

プロセスネットワーク全体でも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

ないことが分かる。図５の矢印は１プロセッサ内でのＺ軸方向への一連の処理を表している。添えられている数字は、矢印の予想される実行時刻である。同じ数字を持つ矢印は同時刻に並列に実行されると考えられる。ただし、プロセッサ間の通信遅延はこの時点では考慮していない。以下、この矢印１本に相当する処理を単位時間として、モデル化を進める。つまり、実行時間を「この矢印を実行する時間の何倍に相当するか」という観点で表現する。

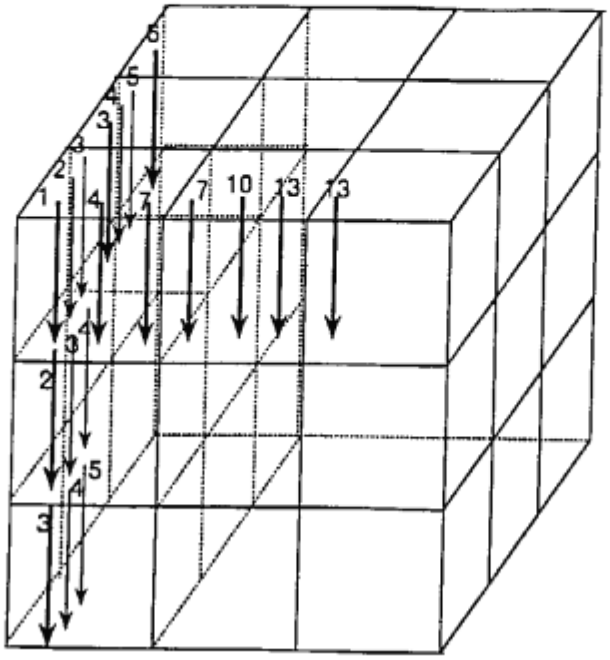


図５：全体の実行の流れ

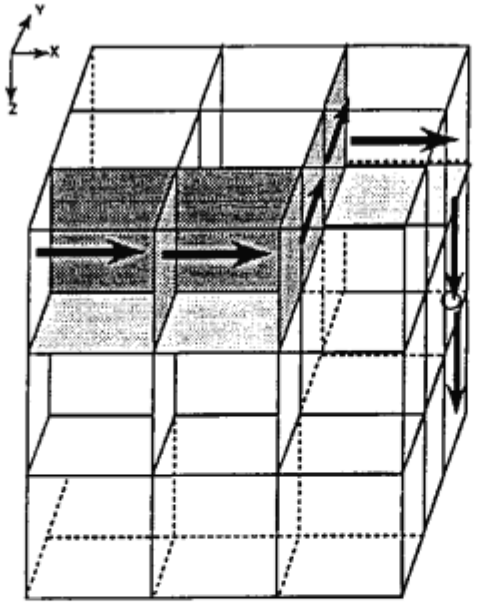


図６：律速処理と通信オーバヘッド

ここで、Nx, Ny, Nz をＸ、Ｙ、Ｚ方向の分割数、lx, ly, lz を各プロセッサに割り付けられるプロセスネットワークのＸ、Ｙ、Ｚ方向のプロセス数とする。図６に律速となる処理を矢印で示したが、その処理に要する時間、すなわち並列実行に要する時間 Tp は次式で表せる。

$$Tp = (lx \times ly) + (lx - 1) \times ly \times (Nx - 1) + (ly - 1) \times (Ny - 1) + (Nz - 1) \quad \text{(単位時間)}$$

一方、全処理を逐次実行する時間 Ts は次のようになる。

$$Ts = (lx \times Nx) \times (ly \times Ny) \times (lz \times Nz) \quad \text{(単位時間)}$$

すると、このモデルから並列実行の実計算時間 Tr は 1 プロセッサでの実行時間 $T1$ の Tp/Ts 倍となることがわかる。

$$Tr = T1 \times (\frac{Tp}{Ts})$$

実際の実行時間 T には、通信オーバヘッド To が含まれるので、

$$T = Tr + To$$

である。

我々のモデルでは、図 6 に示したハッチングの部分に相当するプロセッサ間通信だけが最終的な実行時間に寄与する。なぜならば、これまでに処理の指向性から考えたモデルによると、全体の実行時間を決める処理は、図 6 に示す矢印の処理に等しいからである。それゆえ、この律速となる処理を行うプロセッサの界面にあるプロセスの数とそれらが行うプロセッサ渡りの通信の回数により通信量は決まる。このことにより、先ほど仮定したメッセージ送信および受信に掛かる時間を通信単位時間 C として、通信オーバヘッドは計算可能となる。前と同様に、Nx, Ny, Nz を X、Y、Z 方向の分割数、lx, ly, lz を各プロセッサに割り付けられるプロセスネットワークの X、Y、Z 方向のプロセス数とすると、通信オーバヘッド To は、

$$\begin{aligned} To = 4C \times [\ & [lz \times (lx - 1) + lz \times (ly - 1) + (3/2)lz] + \\ & (Nx - 2) \times [lz \times (lx - 2) + lz \times (ly - 1) + (3/2)lz] + \\ & (Ny - 2) \times [lz \times (ly - 2) + (3/2)lz] + lz \times (ly - 1) + \\ & 2 \times [(Nz - 2) + Nx \times lx \times ly + (Ny - 2) \times ly]] \end{aligned}$$

と表される。

この式について、少し説明を加えると、最初にかかっている $4C$ は、各プロセスの 4 つのメッセージ通信がプロセッサ渡りになっていることを意味している。また、それにかかる部分は図 6 のハッチングの部分のプロセス数、つまりプロセス間通信を行うプロセス数を表している。ただし、$(3/2)lz$ というのは、6 つのメッセージをプロセッサ渡りさせるプロセス群の通信である。また、Z 軸方向には、通信も並列性で相殺されるため、図 5 の矢印の両端の 2 回の通信のみ一層分だけが累積される。

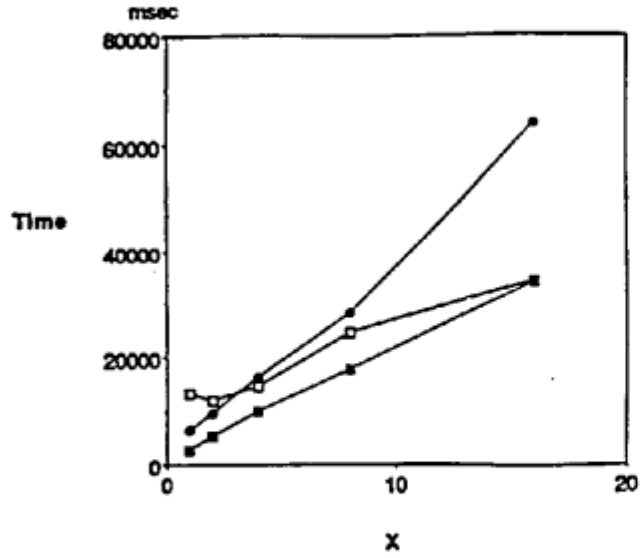
6.2 実験結果

得られたモデルに従って、実験結果との比較を行い、ある程度定量的なプログラムの特徴を捕らえることができた。その結果を以下に述べる。

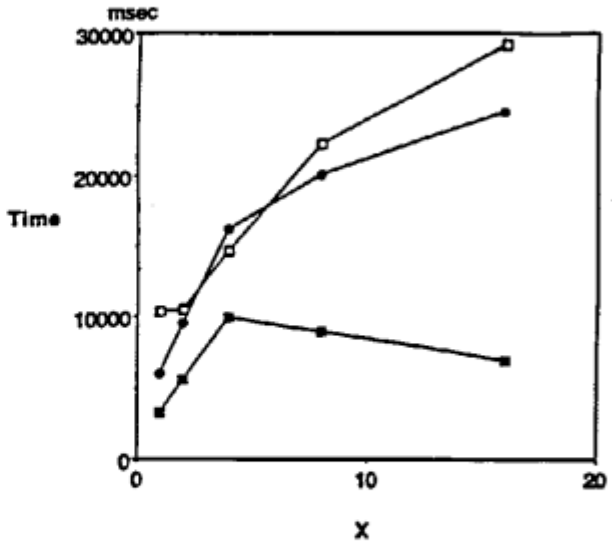
64 プロセッサで実行するとして、X 軸、Y 軸、Z 軸の分割の違いによるパフォーマンスの違いについて検討した。22 の異なった分割に対して実験を行い、実行時間 T の計測を行った。分割しない実行時間 $T1$ も計測した。それらをもとにモデルから C を求め、集計したところ、1.032 ± 0.364 msec となり、ほぼ一定とみられる値が得られた。以下、C の値は 1.0 として、解析を行った。一つの軸方向の分割を 4 として固定した時に、他の軸方向の分割を変化させたものの実測値とモデルから得た値のグラフを図 7 に示す。モデルの $C = 1$ は、C を 1.0msec として、通信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 7 から分かるように、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言える。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 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。これらの傾向は、

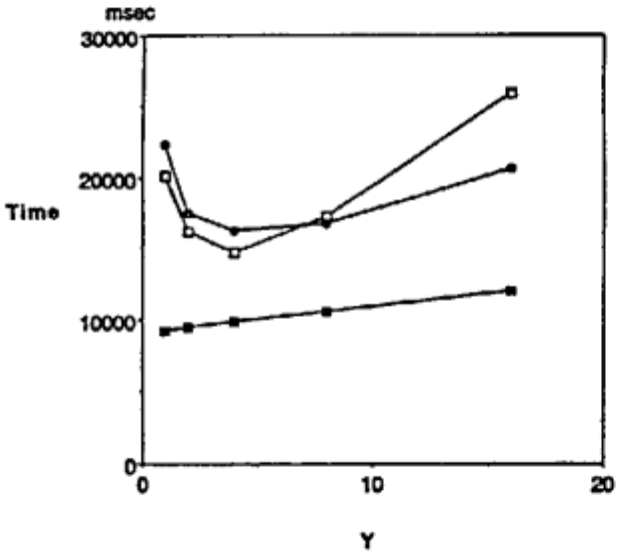
固定する値を変えた場合にも同様な結果が得られている。



Yの分割を4に固定したとき



Zの分割を4に固定したとき



Xの分割を4に固定したとき

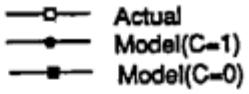


図7：X、Y、Zの分割を4に固定した時のグラフ

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時だと推測できる。

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、1×4×4の分割でマッピングを行った場合で、まさにモデルから得られた推測に一致している。ちなみに、X軸方向の分割を1にした時のグラフが図8である。また、他のデータに対しても、上記の傾向は一致している。

6.3 考察

我々がプログラム設計、開発を行っていた当初は、実行は波面状でかつ、波面はX,Y,Zの各軸方向に時間的にほぼ均等に進むと予想していた。ところが、実際の実行結果には強い指向性があり、波面の進行は軸によって大きく偏りが生じた。そのため本モデルでは、実際の実行がどのように行われ、どの部分の実行が律速段階になっているかに

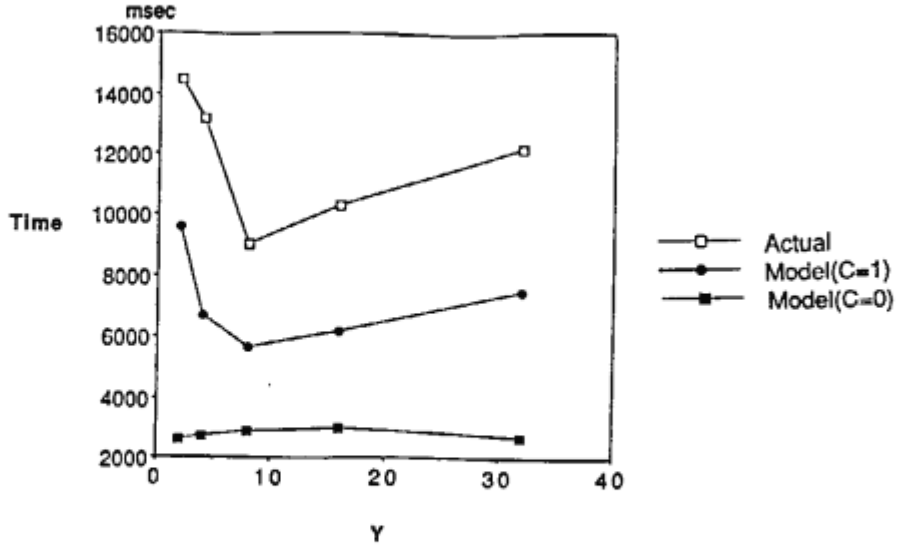


図８：Ｘの分割を１に固定したときのグラフ

ついて、踏み込んだ分析を行う必要が生じた。結果としてモデルが若干複雑になったが、プログラムの挙動をより正確に把握することができた。

本プログラムでは、１つのアライメントの実行において各ノードは必ず１回だけ実行され、無駄計算は一切生じない。それゆえ、並列化を行ったことによって生じるオーバヘッドは、通信オーバヘッドのみと考えて差し支えない。

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持つものであった。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。

実際、図７のグラフを見ると、通信オーバヘッドまで含めて定量的にモデルを考えないと、プログラムの特性を表現するのに不十分であることがよく分かる。特に、Ｘの値を固定した場合とＺの値を固定した場合は、通信オーバヘッドがないときの理想的理論値のグラフが、実測値のグラフと大きく異なっている。しかし、通信オーバヘッドまで含めた理論値のグラフは実測値のグラフと、かなり近いものになっていることが分かる。

並列処理を考察するうえでは、通信のない理想的なモデルだけを考えて、理論値と実測値の差を単に通信オーバヘッドの影響であると言うだけでは十分ではない。今後は、通信オーバヘッドまで含めたモデル構築を行うことが、ますます重要になると考えられる。

また本解析において、１つの式でモデルを表現できたことも興味深い。図７、８の理論値のグラフは、ただ１つの式から求められたものである。本プログラムの実行の進み方は強い指向性を持つと先に述べた。この実行の進み方の指向性はＸ軸方向に対して最も弱い。そのためＸ軸方向に細かく刻んでプロセッサマッピングを行うのは好ましくないということは、ある程度直観的に理解できる。しかしＸ軸方向のプロセッサマッピングを固定したときに、Ｙ軸とＺ軸の間にトレードオフの関係があるということは、実際にモデルを作って式を立てた後、理論値のグラフを書いてみて初めて分かったことである。このように本モデルを用いて、プログラムの挙動を多角的に把握することができ、プロセッサマッピングの指針を得ることができた。

7　おわりに

以上のように、適切な仮定のもとにモデルを考えれば、ＫＬ１のように高度に並列動作するプログラムでさえも、その解析は十分に可能である。今回仮定したことは、処理系などの多少の知識があれば妥当であろうと考えられるも

ののみである。そして、それをもとに解析した結果、モデル化が現実のシステムに即したものであることが確められた。こうしたモデルは、プロセッサへの並列マッピング方法の検討や、並列プログラムの挙動の理解に、極めて有用なものである。

本システムによるマルチプルアライメントは、実際にはパイプライン的に処理を行う。今回の解析では、パイプライン処理を考慮していないが、今回得られた結果とパイプライン効果の相性を検討することが、これからの課題に挙げられる。

参考文献

[Bilofsky 88] Bilofsky, H. S. and Burks, C. *"The GenBank Genetic Sequence Data Bank"* in *Nucleic Acids Research 16:5*, 1988, pp.1861-1863.

[Kimura 86] 木村：分子進化の中立説, 1986, 紀伊国屋書店

[Dayhoff 78] Dayhoff, Hunt and Hurst-Calderone *"Composition of Proteins"* in *Atlas of Protein Sequence and Structure 5:3*, Nat. Biomed. Res. Found., Washington, D. C., 1978, pp.363-373.

[Needleman 70] Needleman, S. B. and Wunsch, C. D. *"A General Method Applicable to the Search for Similarities in the Amino Acid Sequences of Two Proteins"*, in *Journal of Molecular Biology 48*, 1970, pp.443-453.

[Waterman 86] Waterman, M. S. *"Multiple Sequence Alignment by Consensus"* in *Nucleic Acids Research 14:22*, 1986, pp.9095-9102.

[Murata 1 85] Mitsuo Murata *"Simultaneous Comparison of Three Protein Sequences"* in *Proc. Natl. Acad. Sci. USA Vol. 82*, 1985, pp.3073–3077

[Murata 2 90] Mitsuo Murata *"Three-Way Needleman-Wunsch Algorithm "* in *Methods in Enzymology Volume 183*, Academic Press, 1990, pp.365–375

[Carrillo 88] Himberto Carrillo and David Lipman *"The Multiple Sequence Alignment Problem in Biology"* in *J. Appl. Math. 48*, 1988, pp.1073–1082

[Wada 90] 和田、市吉：マルチＰＳＩ上の最短経路問題の実現と評価, 1990, Proceedings of KL1 Programming Workshop '90, pp.10–17

並列処理マシン上でのシミュレーティッド・アニーリング

荒木均　舘野峰夫　加藤等　問藤隆一
松下電器産業株式会社 情報通信東京研究所

1 はじめに

ICOTの後期の委託研究において、我々は組合せ最適化問題を効率的に解くために、シミュレーティッド・アニーリング(SA)法へのヒューリスティックな知識の導入とSA法の並列化を研究している。

平成元年度の並列SA法は、状態を表現しているデータを分割して各プロセッサ・エレメント(PE)に割り当て、各PEで並列にSAを行なう方法であった。この方法を論理アーキテクチャ設計問題に適用し、データの分割数が少なければ、通常のSA法と比べて短時間で準最適な解を生成することを確認した。しかし、状態を表現しているデータの分割数には限界があり、それ以上に分割すると、準最適な解を見つけることができなかった。すなわち、分割した場合、各部で全体のコストを正確に知ることができないことが大きな原因である。

そこで、平成二年度は、各種の組合せ最適化問題に適用できるように、状態を複数のPEに一つずつ割り当て並列にSAを行なう方法を提案する。本手法は、二つの特徴を持ち、一つはコストの分布から各温度での定常状態を判断することである。もう一つは、アニーリングの過程で極小解に陥ってしまったPEだけが他のPEが持つ状態を受信し、その状態からアニーリングを開始することである。

本手法を16PEのマルチPSI上で論理アーキテクチャ設計問題を例に採り、通常のSA法と比較実験を行なった。その結果、約1/10の処理時間で同程度の解を得られることを確認した。

2 SA法

SA法[Kirkpatrick 83]は、物理現象である焼きなましを計算機上で模擬することにより、エネルギー（コスト）最小の状態を求める手法であり、各種組合せ最適化問題に適用可能な最適化アルゴリズムの一つである。そのアルゴリズムを以下に示す[木村 90]。

SA法は適当な初期状態s_0から出発し、状態$\{s_n\}_{n=0,1,2,\ldots}$を次のように順に生成し、コスト最小の状態(最適解)に近づけていく。まず、s_nに微小変形を施したs'_nを生成し、コストの差$\Delta c = c(s'_n) - c(s_n)$を計算する。$\Delta c \leq 0$ならば、$s_{n+1} = s'_n$とし、$\Delta c > 0$ならば、$p = \exp(-\Delta c/T)$として、確率$p$で$s_{n+1} = s'_n$、確率$1-p$で$s_{n+1} = s_n$とする。ここで$T$は温度パラメータと呼ばれ、変換回数$n$に依存して、高温から低温に徐々に変化させる。これをクーリング・スケジュールと呼び、温度一定で一定回数変換を繰り返し、Tを段階的に下げていくのが一般的である。これは最も単純なクーリング・スケジュールであるが、比較的良い結果をもたらすことが知られている[Kirkpatrick 83]。

温度一定の時、SAはエネルギー平衡の状態に向かう統計熱力学的な系の挙動としてとらえることができ、無限回変換を繰り返すと、どのような状態から出発しても、状態iとなる

確率が $\pi_i(T)$

$$\pi_i(T) = \frac{1}{Z(T)} \exp(-\frac{c_i}{T}), \tag{1}$$
$$Z(T) = \sum_{i \in S} \exp(-\frac{c_i}{T}).$$

である定常確率分布 $\pi(T)$ に漸近的に近づく (ただし、c_i は状態 i のコスト)。そのため、温度 T をうまく制御することにより、コストの局小的最小値を脱出して大域的最小値に収束することができる (図 1参照)。この収束過程は、大きく次の三段階にわかれると考えられ、各段階は明確な境界を持つものではなく、段階の移行は徐々に行なわれる [山下 89]。

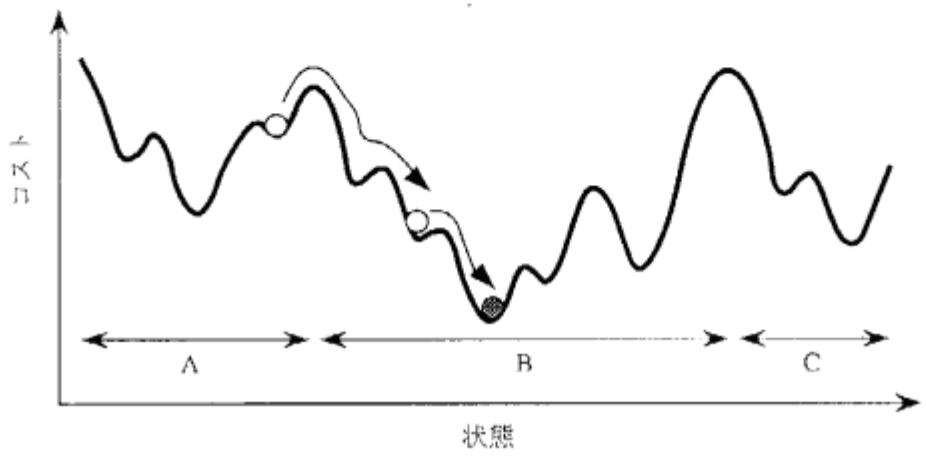


図 1: SA の収束過程

- 高温部

  十分大きな温度では、SA はランダムウォークのようにふるまう。そのため、図 1の大きなコスト障壁も簡単に越えることができ、自由に遷移することができる。したがって、高温での定常確率分布 $\pi(T)$ は一様分布に近く、変換回数を多くしてもあまり意味はない。

- 中温部

  アニーリングが進み、温度が徐々に下がってくると、やがて、大きなコスト障壁を越えることは困難になってきて、図 1の大きな谷 (例えば、A,B,C) のどこかに落ち込んでいく。これは、式 (1) で示される定常確率 $\pi_i(T)$ に差が生じ、コストの高い状態になる確率が小さくなるためである。この $\pi_i(T)$ に差が生じてくる中温部が収束状態を決定する最も重要な段階である。したがって、この中温部に最も多くの変換を施し、慎重に収束先の状態を決定することが必要である。

- 低温部

  低温部では、多くの変換がリジェクトされてしまい、状態の大きな変化は望めなくなる。すなわち、図 1の大きな谷から、飛び出すことはもはや不可能になってきて、大域的最小値へと収束していく。逆に、中温部で十分な回数変換が行なわれていないと、局所的最小値に収束してしまう。

上記のような収束過程から、SA 法は最適解を見つける可能性を持っているが、多大な計算時間を要することが問題であった。そこで、SA 法を並列化して、高速化するという研究が盛んに行なわれている。

その中でも、状態を表現しているデータを分割して各 PE に割り当て、各 PE で並列に各部の変換を行なう方法が最も良く研究されている [Rose 86][Casotto 87]。この方法は、各 PE で正確なコストを知ることができないため、分割された各部の依存関係が大きい場合には、得られる解が極小解になってしまうことが問題となっている。

また、各温度での変換の開始時に、各 PE に同じ状態を一つずつ割り当て並列に変換を繰り返し、次の温度では、全 PE の中でコストが最も小さい状態を各 PE に割り当て変換を繰り返す方法も提案されている [Aarts 89]。しかし、この方法で得られる解は、PE 間の通信を行なわずに各 PE で独立に SA を行ない、最終的に得られる各 PE の解の中で最もコストが小さいものとほとんど差がないと言われている [Aarts 89]。

これらの方法以外にも、各 PE に同じ状態を割り当て並列に変換を行ない、最初に新しい状態を受理することができた PE が他の PE にその状態を送信し、再び複数の PE が同じ状態に対して変換を繰り返す方法もある [Aarts 86]。この方法は、高温では状態の受理率が高いので、実行効率が低下してしまう。そこで、同じ状態に対して変換を行なう PE 数を、温度の低下とともに徐々に増加していくことにより、実行効率をあげている。しかし、疎結合型の並列計算機では通信コストが大きいので、この方法を適用することは処理速度の低下になると考えられる。

これらの並列方法に対して、本手法では、状態を各 PE に一つずつ割り当て並列に SA を行ない、アニーリング過程で極小解に陥ってしまった判断される PE についてのみ、他の PE が持つ状態を受信する手法を提案する。これにより、コストが小さくなりそうな状態の近傍を複数の PE で探索することができる。また、高温では定常状態に達したかどうかを各 PE のコストから判断し、変換回数を削減する方法も同時にとり入れている。

3 並列 SA 法

上で述べた SA の定性的な収束過程から、高温部では変換回数を多くする必要はないが、中低温部で変換回数が少ないと、図 1の大域的最小値には収束せず、局所的最小値に落ち込んでしまう可能性がある。この可能性は変換回数が少なければ少ないほど大きくなる。そこで、高温では、PE のコスト分布から定常状態を判断することにより変換回数を削減し、中低温では、アニーリングの過程で見つかったコストが小さい状態の近傍を複数の PE で探索する以下のような手法を考える。

まず、各 PE に状態を一つずつ割り当て並列にアニーリングを開始する。高温部ではアニーリングを始める前に、コストの大小により PE を大きいコストを持つグループと小さいコストを持つグループに二分する。そして、一定温度でのアニーリングの過程で二つのグループのコスト分布に違いがなくなれば、その温度での定常状態に達したと判断して温度を下げ、再び大きいコストと小さいコストのグループに PE を分けて、同様な処理を繰り返す。逆に、十分変換を施しても同様なコスト分布にならなければ、中低温まで温度が下がってきたと考える。中低温では、各 PE は λ 回前から現在までの変換で得られたコストの最小値を保持し、PE の中で現在最も小さいコストを持つ PE の λ 回前からの最大値より、最小値が大きければ、その PE は極小解に陥ってしまったとみなす。そのような PE は、他の PE のコストが小さい状態を受信し、それ以後は受信した状態についてアニーリングを行なう。

以上のような並列 SA 法は、高温部では二つのグループのコスト分布の違いがすぐになく

なるので、変換回数を削減することができる。また、中低温部ではコストが小さい状態に対して複数のPEがSAを行なうことが多くなり、より高い確率でコスト最小の状態を見つけることが期待できる。更に、状態の通信はコストの通信よりも通信負荷が大きいので、極小解に陥ってしまったと考えられるPEだけが状態を受信する方法は通信負荷を小さくすることができる。

以下では、高温部でコストの分布に違いがあるかどうかを判断するために用いた順位検定について述べ、その後、並列SA法の処理ついて詳しく述べる。

3.1 順位検定

順位検定[竹内 75]とは、二つの母集団のサンプルから、二つの母集団の確率分布が等しいかどうかを判定する検定方法である。例えば、二つの母集団G_x, G_yから得られたサンプルが

$$(X_1, \ldots, X_7) = (12, 27, 50, 61, 79, 91, 110)$$

$$(Y_1, \ldots, Y_7) = (10, 18, 25, 30, 39, 85, 120)$$

とする。これを大きさの順に並べると、

$$yxyyxyyxxxyxxy$$

となる。このとき、図2のようなグラフを考える。原点$(0,0)$から出発して、まず最初の値がxならば右へ一目盛進み、yならば上へ一目盛進む。次に第2の値を見て、xならば右へ、yならば上へ進む。このようにして、点$(7,7)$まで進む道を考えることができる。そして、図2の影の部分、すなわち45°とこのグラフで囲まれる部分の面積を検定統計量にしたものが順位検定である。

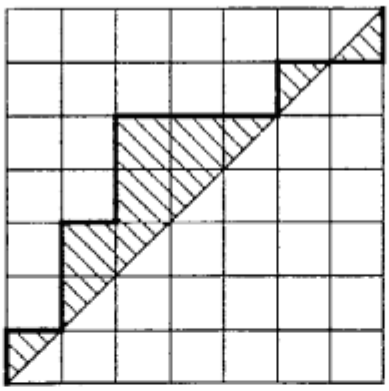


図 2: 順位検定

順位検定を用いた理由は、温度一定のSAでは、定常状態に達した時に、初期状態に関係なく同じ定常確率分布に収束するためである。すなわち、コストの大小によりPEをコストが小さいグループG_xと大きいグループG_yに分け、二つのグループのコストの分布に違いがなくなければ、定常状態に達したと考えるわけである。

3.2 処理詳細

本手法では、温度の管理、順位検定、状態送受信の決定などを行なう一つの管理 PE(以下では MPE と呼ぶ)、実際にアニーリングを行なう複数の状態生成 PE(以下では GPE と呼ぶ)が存在し、MPE に一つの PE が、GPE にその他の PE が割り当てられる。MPE の処理の流れを図 3に示す。

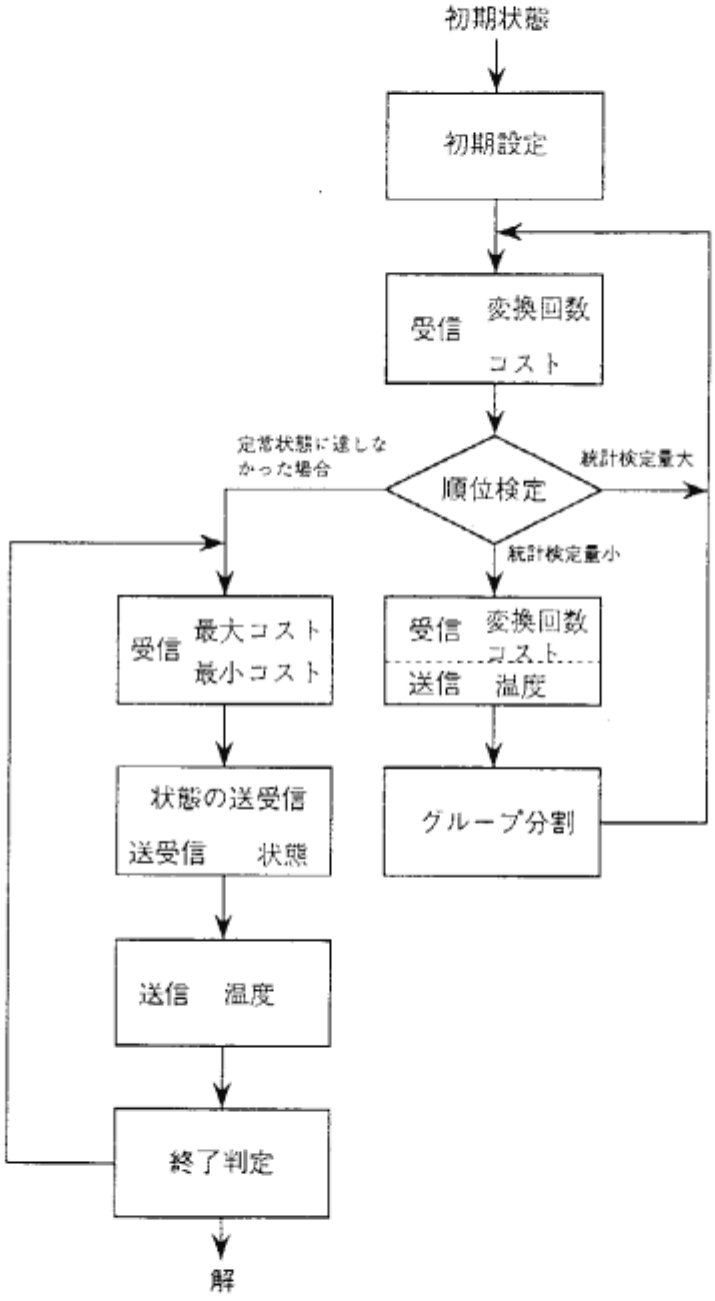


図 3: 管理 PE(MPE) の処理の流れ

初期設定では、MPE は一つの初期状態を各 GPE に割り振る。GPE は温度無限大で一定回数変換を繰り返し、異なる状態を生成する。そして、MPE にそのコストを送信すると同時に、初期温度を受信し、再び変換を開始する。このとき、MPE は送信されたコストに

より、コストが小さいグループ G_x とコストが大きいグループ G_y の二つに分け順位検定を行なう。このときの検定統計量は二つのグループのコスト分布が全く違うことを示す大きな値 r_0 となる。

次に、MPEは、一定の変換回数間隔でGPEからコストを受信し、G_x のコストと G_y のコストで順位検定を行なう。このとき、検定統計量が $\beta \cdot r_0$ 以上ならば、まだ定常状態に達していないと考え、再びコストの受信を行なう (ただし、β は $0 \leq \beta < 1$ の定数)。そうでなければ、定常状態に達したと考え、新しい温度を送信すると同時に、コストを受信する。その後、グループ分割部で、受信したコストにより再び二つのグループに分け、GPEからのコストの受信を再開する。

ただし、変換回数が K 以上になっても、定常状態に達することができなかった場合、コストが小さくなりそうもないGPEだけが現在持っている状態を捨てて、コストが小さい状態を受信する。すなわち、各GPEは λ 回前から現在までのコストの最大値と最小値を保持しており、各温度の最終状態で最も小さいコストを持つGPEの λ 回前からの最大値よりも、最小値が大きいGPEだけが、他のコストの小さい状態を持つPEから順に受信する。この送受信を行なうGPEの組はMPEが決定する。

終了判定は、前々回、前回、現在の温度で、送信されるコストの最小値が同じならば、各GPEにアニーリングを終了するように要請し、最小のコストを持つ状態を解とする。

4 結果

上記のような並列SA法をPE数が16個のマルチPSI上にインプリメントし、論理アーキテクチャ設計問題に適用した。ここで言う論理アーキテクチャ設計問題とは、PASCALなどの高級言語で書かれた動作仕様からレジスタ・トランスファー・レベルの回路を出力するものである [館野 90]。そのコスト c は、ALU、レジスタ、バスのチップ面積、処理時間を考慮した

$$c = (alu) + (register) + (bus) + (time) \tag{2}$$

で表される。

最終的に得られたコストを評価するために、最小コストに対する、最終コストと最小コストの差として、コスト誤差 e

$$e = \frac{c_{fin} - c_{opt}}{c_{opt}} \tag{3}$$

を定義した。ここで、c_{fin} は最終コスト、c_{opt} は最小コストである。一般に、c_{opt} は分からないが、平成元年度からの実験で得られた解の最小値とした。

また、通常のSAとの比較のため、次のようにパラメータを設定した。

初期温度 T_0	553°
温度減少規則	$T_{k+1} = 0.9 \cdot T_k$
終了条件	各温度での最終コストが3回連続同じとき

ただし、T_0 は受理率 $\chi_0 = 0.9$ として、次式により求めたものである [Laarhoven 87]。

$$T_0 = \frac{\overline{\Delta C^{(+)}}}{\ln(\chi_0^{-1})} \tag{4}$$

ここで、$\overline{\Delta C^{(+)}}$ はランダムウォークの平均コスト増加値である。

上記の初期温度、温度減少規則、終了条件は、通常のSAの最も一般的なパラメータ値の一つである。これらの設定以外に並列SA法のパラメータとして、

定常状態判定の閾値	$0.4 \cdot r_0$
持続回数 λ	$5 \cdot K$

とした。ここで、r_0 は各温度での開始時の順位検定値、K は変換回数の上限である。

K を変化させて実験を行ない、各実験で乱数の種を変えて5回試行した。その平均のコスト誤差と計算時間を表1に示す。表の変換回数 K は、並列SAでは各温度での変換回数の上限を、通常のSAでは各温度で K 回の変換を行なったことを意味する。

表 1: 並列SAとSAの比較

変換回数 K	コスト誤差		計算時間 (sec)	
	並列SA	SA	並列SA	SA
32	0.218	0.405	474	471
64	0.141	0.302	776	920
128	0.080	0.237	1359	1936
256	0.099	0.163	2494	4159
512	0.046	0.137	5012	8154

表1から、通常のSAで平均エラーが0.15を下回るには、各温度での512回の変換回数が必要であるが、並列SAでは64回で十分であることが分かる。そして、SAで変換回数が512回の場合は8154(sec)であり、並列SAで変換回数が64回の場合は776(sec)である。このことから、本並列SA法ではPE数が16個の場合の台数効果は約10.5(= 8154/776)であると言える。

5 まとめ

本稿では、SAの新しい並列処理方法について述べた。本手法は、PEのコストの分布から定常状態を判断することと、コストの変動範囲から極小解に陥ってしまった判断されたPEだけが状態の受信を行なうという二つの特徴を持つ。論理アーキテクチャ設計問題について本手法の有効性を確認したが、他の並列SA法との定量的な比較が不十分である。今後の課題としては、これらの比較に加えて、大規模並列計算機では、管理PEの通信負荷がかなり大きくなることが予想されるので、定常状態の判定、状態の送受信を局所的な範囲内のPE間で行なう処理方式にする予定である。

参考文献

[木村 90] 木村、瀧: 時間的に一様な並列アニーリングアルゴリズム, 信学技法, NC90-1, pp.1-8, (1990).

[舘野 90] 舘野、荒木、間藤: 論理設計エキスパートシステム (1) - アニーリング・ルールベース -, 第42回情処全大, 6K-3, (1990).

[竹内 75] 竹内: 確率分布と統計解析, 日本規格協会, (1975).

[山下 89] 山下、秋山、安西: エントロピーを用いた改良シミュレーティッドアニーリングに関する研究, 信学技法, NC89-67, pp.37-42, (1989).

[Aarts 86] E.H.L.Aarts, F.M.J.Bont, J.H.A.Habers and P.J.M. van Laarhoven, *"Parallel implementations of the statistical cooling Algorithm"*, *Integration*, journal 4, pp.209-238, (1986).

[Aarts 89] E.H.L.Aarts, J.H.M.Korst, *"Simulated Annealing and Boltzmann Machines"*, John Wiley & Sons, (1989).

[Casotto 87] A.Casotto and A.Sangiovanni-Vincentelli, *"Placement of standard cells using simulated annealing"*, *Proc. IEEE Int. Conf. on Computer-Aided Design*, Santa Clara, pp.350-353, (1987).

[Kirkpatrick 83] S.Kirkpatrick, C.D.Gelatt and M.P. Vecci, *"Optimization by simulated annealing"*, *Science*, vol.220, no.4598, pp.671-683, (1983).

[Rose 86] J.S.Rose, D.R.Blythe, W.M.Snelgrove and Z.G.Vranesic, *"Fast, high quality VLSI placement on an MIMD multiprocessor"*, *Proc. IEEE Int. Conf. on Computer-Aided Design*, Santa Clara, pp.42-45, (1986).

[Laarhoven 87] P.J.M. van Laarhoven and E.H.L.Aarts, *"Simulated Annealing : Theory and Applications"*, Kluwer Academic Publishers, (1987).

制御用エキスパートシステムのマルチＰＳＩ上での実現

鈴木淳三，小沼千穂，岩政幹人，市川哲彦，木田直道
（株）東芝　システム・ソフトウェア技術研究所

概要

従来のプラント運転制御システムは、知識ベースに格納された経験則に基づいて推論と運転制御を行うため、事前にその経験則を用意出来ないプラント状態（不測事態と呼ぶ）に対応することができない。またプラント運転制御システムには、リアルタイム性を確保するために推論の高速化も必要である。前者の問題点に対して、我々はモデルに基づく推論に着目し、不測事態において必要なプラント操作に関して、その制御知識をダイナミックに生成できる制御用エキスパートシステムの研究開発を行ってきた。本稿では後者の要求に対するＫＬ１プログラム研究開発実例として、マルチＰＳＩ上での制御用エキスパートシステムの実現方式に関して述べる。

1　はじめに

制御用エキスパートシステムは、モデルに基づく推論によりダイナミックにプラント運転制御に必要な制御知識を生成できることを特徴とするシステムである[1]。本システムは、図１に示すように、予め知識ベースに格納された制御知識に基づいてプラント運転操作に関する推論を行う正常時推論機構、モデルに基づいてプラント運転操作用の制御知識を生成する異常時推論機構から構成されている。さらに異常時推論機構は、故障仮説生成部、操作同定部、操作条件生成部、シミュレーション評価部の各推論モジュールから構成されている。また各推論モジュールにおいては図２に示すプラント構成がモデル化されている。これらの推論機構と推論モジュールの高速化に関しては、以下の観点に従った並列化を検討した。

（１）正常時推論機構の機能分割による並列化

正常時推論機構は、データ駆動型のリアルタイム制御を実現するために、大別して機能的に独立した７つのプロセスが存在する。例えば、プラントとのデータＩ／Ｏを行うプロセス、プラントデータの変化を検出するプロセスなどがある。したがってこれらのプロセスを異なるＰＥ（Processor Element）に割り付けることにより負荷分散が可能である。

（２）故障仮説生成部のモデル分割による並列化およびクラスタリングの並列化

故障仮説生成部は、センサにより得られたパラメータ値が期待される値とずれていた場合、その定性的な偏差をパラメータ間の定性的因果関係式に従って結果側から原因側へ定性伝播することにより故障仮説を生成する[2]。この定性的因果関係式を、関連する機器毎に異なるＰＥで分割管理することにより、異なる機器に関するパラメータの定性的偏差は全て並列に定性伝播することができる。さらに故障仮説生成部では、多重故障に対する故障仮説を生成するために、定性伝播により得られる故障仮説のクラスタリングを行う[3]。このクラスタリング処理は集合演算を基に実行されている[4]。この時、集合の各要素を並列に計算することのできる部分、および集合の要素の計算においてパイプライン的な計算が可能な部分の負荷分散が可能である。

（３）操作同定部の状態探索・状態検証における機器構成による並列化

操作同定部では、プラントを構成する機器の接続関係に従って、個々の機器の状態を生成したり検証したりしながら、故障仮説に対応するプラント操作を生成する[5]。この時、並列に接続された機器の状態生成を並列に実行したり、操作対象の機器と接続された複数の機器に対して並列に操作による影響を検証することが可能である。

（４）操作条件生成部の操作・条件単位の並列化

操作条件生成部は、プラント操作に対する条件を生成しＩＦ－ＴＨＥＮ形式の制御知識を生成する[6]。したがってプラント操作が複数ある場合は、各操作毎の条件生成処理は並列に実行できる。また各操作に対して５種類の条件を生成するが、これに関しても並列に実行できる。

（５）シミュレーション評価部のモデル分割とパイプライン計算による並列化

シミュレーション評価部は、プラントの動特性モデルに基づいたファジィ化定性推論によりプラントの挙動予測を行う[7][8]。この時、ファジィ演算そのものの並列化が可能である。さらに、動特性モデルをその構成によ

りいくつかのエリアに分割することにより並列に推論できる部分がある。またシミュレーション実行時に、入力データが揃った部分からシミュレーションを実行していく言わゆるパイプライン的な並列化も可能である。

以上の並列化ポイントを整理してみると、（a）プロセス間通信がない完全に独立したプロセスの並列化、（b）若干のプロセス間通信を含むプロセスの並列化、（c）パイプライン演算の可能なプロセスの並列化、に分類できる。本稿では、ある程度の並列化台数効果の認められた事例として（4）および（5）について以下に詳しく説明し、並列プロセス構成とその実装方式、実験結果と評価結果を述べる。事例（4）は並列化ポイント（a）、事例（5）は並列化ポイント（c）の確認実験となっている。また、その他の事例に関しても最後にまとめて簡単に触れ、ＫＬ１プログラミングの雑感と併せてまとめる。

2　操作条件生成部の操作・条件単位の並列化

操作条件生成部では、操作同定部で生成された各プラント操作に対する操作条件を推論し、ＩＦ－ＴＨＥＮ形式の制御知識を生成する。したがって、各操作を実行する前のプラント状態を決定した後は、図３に示すように各操作毎に条件生成を並列に実行することができる（操作レベルの並列化）。また各操作に関して５種類の一般化条件を生成するが、これらも互いに独立に生成できる（条件レベルの並列化）。したがって、操作と条件毎に条件生成プロセスを異なるＰＥに割り付けることにより推論の並列化が可能である。以下ではプロセス構成と実装方式について概略を述べた後に、並列推論の概略フローを示し、最後に実験結果と評価結果に関して述べる。

２．１　プロセス構成と実装方式

図４に並列プロセス構成とモデルおよびデータの分散方式を示す。負荷分散されるプロセスには、条件を生成する各操作に対応する条件生成プロセス（操作レベルの並列化）、これらのプロセスを管理し一連の条件生成処理を制御するスーパーバイザ・プロセスが存在する。条件レベルの並列化も行えば、条件生成プロセスの子プロセスがさらにＰＥに分散されることになる。スーパーバイザ・プロセスはＰＥ０への固定割付、条件生成プロセスとその子プロセスは残りのＰＥへサイクリックに割り付けられるよ

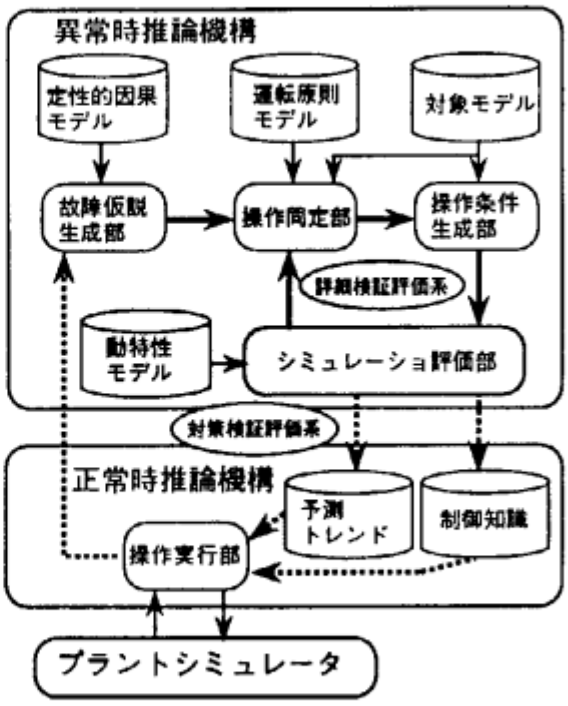


図１ 制御用ＥＳのシステム構成図

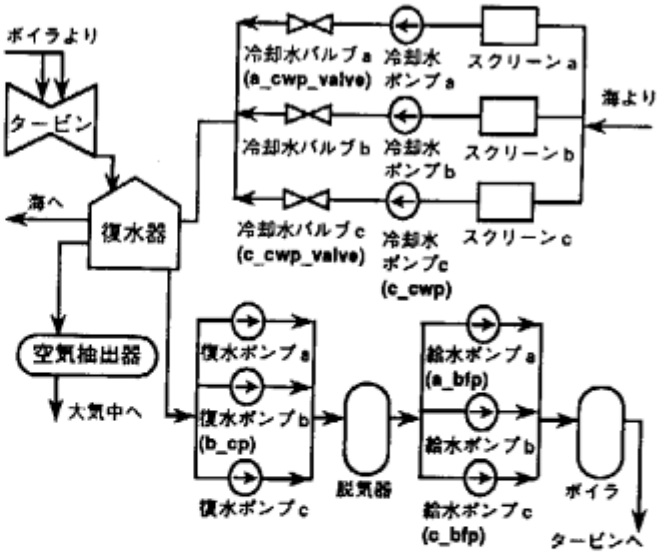


図2　モデル化された火力発電プラント

うにした。負荷分散された複数の条件生成プロセスが共通のモデルやデータにアクセスする時、ＰＥを跨いだ参照が発生するとＰＥ間通信コストのために並列効果が減少する欠点がある。この問題点に対して、メモリ効率を犠牲にして各ＰＥに共通のモデルやデータを予め確保しておく手法を採用した。この処理は条件生成

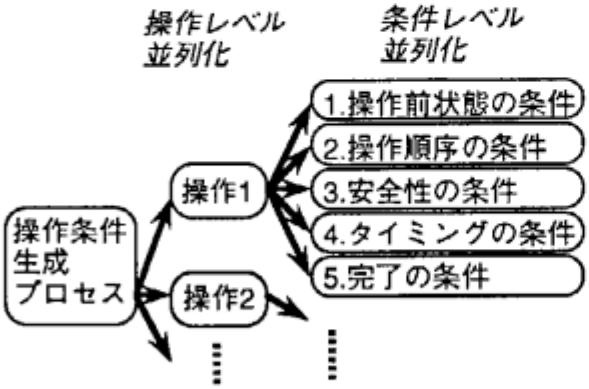


図３ 操作条件生成部の並列化イメージ

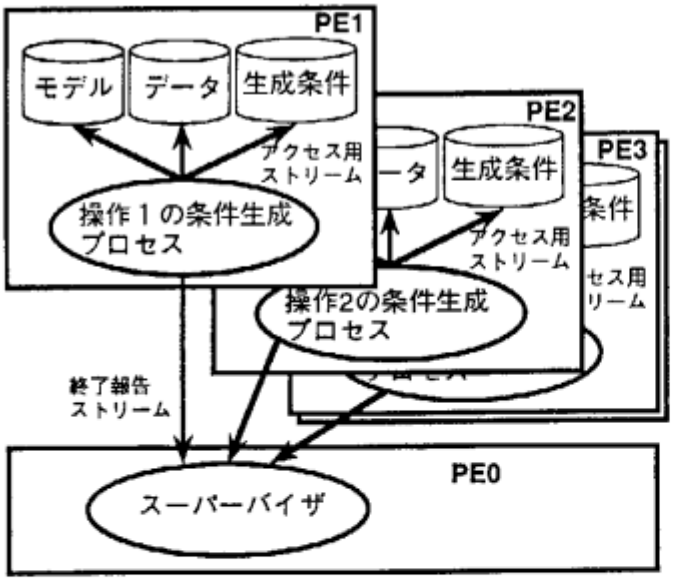


図４ 操作条件生成部の並列プロセス構成と
モデル・データ分散方式

処理に先だって実行することができるため、推論のリアルタイム性を損なうことがない。

２．２　並列推論フロー

図５に並列推論の概略フローを示す。分散データの初期化では、各プラント操作に対して、その実行前の各プラント機器の状態がモデルに設定される。条件生成においては操作レベルと条件レベルの並列化を行い、各操作に対する条件を生成する。条件生成アルゴリズムの詳細は文献[6]を参照されたい。各ＰＥに割り付られたプロセスにより生成された条件は、操作毎に収集された後、タイミングに関する条件（Ｔ条件）、操作前の状態に関する条件（Ｐ条件）、操作完了に関する条件（Ｃ条件）に分類され、最終的にＩＦ－ＴＨＥＮ型の制御知識として出力される。実験では、これらの分類処理も並列化を行った。

２．３　実験結果と評価

図２に示した対象プラントにおいて、以下に示す４つの事例に関して実験を行った。

事例１：給水ポンプ故障によるポンプ切り替え操作（a_bfp、c_bfp、b_cpに関する３つの操作）
事例２：海水温度上昇による冷却水ポンプ操作（a_cwp_valve、c_cwp_valve、c_cwpに関する３つの操作）
事例３：これら両方の操作（６つの操作）
事例４：残りの機器操作を合わせた合計１０個の操作

図６に並列効果に関する実験結果を示す。横軸は推論に利用したＰＥ台数であり、縦軸は推論総実時間である（単位はミリ秒）。実験条件としては、ＰＥ割付はサイクリック方式を採用し、負荷分散レベルは操作レベルまでとした。したがって、事例１と２に関してはスーパーバイザと条件分類処理も含めて最高１０個のＰＥを利用した実験結果までをプロットした。

実験の結果、以下の評価結果を得た。

（１）操作レベルの並列化に関しては、推論総実時間に対して、事例に応じて約２倍から３倍の並列効果が確認できた。当然のことであるが、条件生成を行う操作数が多い事例ほど並列効果が高い結果を得た。これにより操作レベルの並列化の有効性が確認できた。

（２）上記に関して、図５の条件生成の推論時間に対しては、事例に応じて約３倍から５倍の並列効果が確認できた。一方、図５の条件出力の推論時間に対しては、事例に応じて約２倍から３倍の並列効果しか達成できず、さらに条件出力に要する時間が条件生成に要する時間の４倍程度必要であることが確認できた。したがって、条件出力処理の並列効果が期待した程得られなかった点が、結果として全体の並列効果を引き下げる要因になったと思われる。

（３）条件レベルの並列化に関しては、これを行わない場合との推論時間の比較において並列効果がでないばかりでなく、かえって推論時間が増加してしまうことが分かった。これは実験でのモデルが比較的単純であるために、各条件の生成を並列に実行することによる効果よりも負荷分散によるＰＥ間通信コストの増加の方が大きいことにあると考えられる。

３　シミュレーション評価部のモデル分割とパイプライン計算による並列化

シミュレーション評価部では、生成された制御知識に基づいてプラントを操作した時のプラント挙動を予測する。本モジュールでは、動特性モデルに基づく予測推論機構としてファジィ化定性推論を採用している。これはモデルのあいまい性を考慮できるという点で有効であるが、各パラメータ値の計算を通常のファジィ演算と同様にファジィルールの総合的評価によって計算する必要がある。したがって、一般的な数値演算と比較すると推論コストが高い欠点があり、推論のリアルタイム性の要求に対して問題がある。したがって、以下に示す並列化ポイントに基づいて推論の高速化を検討した[9]。

（１）ファジィ演算の並列化
　メンバシップ関数に基づいたｍｉｎ－ｍａｘ演算を行う時に、各ファジィルールの演算は並列に実行することができる。

（２）動特性モデル分割による並列化
　プラントの動特性モデルを複数のエリアに分割することにより、各エリアにおけるパラメータ値の計算を並列に実行したりパイプライン的に計算することができる。

以下ではプロセス構成と実装方式について説明した後に、実験結果を示しその評価を行う。

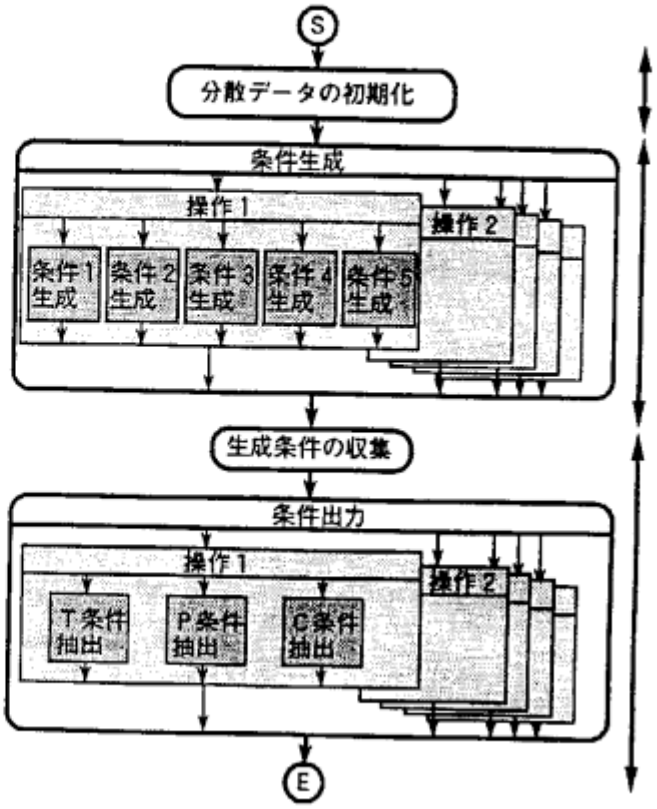


図５ 操作条件生成部の並列推論概略フロー

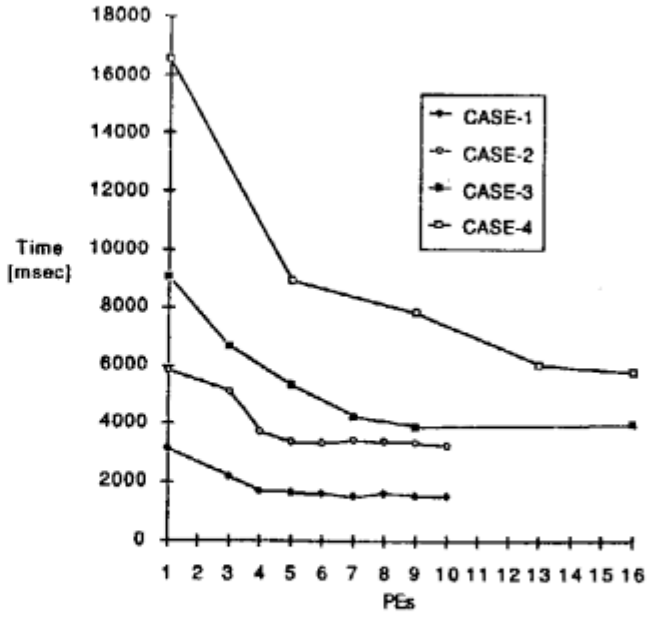


図６　操作条件生成部の操作レベルの並列化実験結果

３．１　プロセス構成と実装方式

先に示した各々の並列化ポイントに対して、図７（ａ）（ｂ）に示すような２つのシミュレーション・プロセス構成とモデル・データ分散方式を試作した。同図（ａ）はファジィ演算の並列化に関するものである。動特性モデルとファジィルールおよびパラメータ値はＰＥ０で一括管理され、シミュレーション・プロセスもＰＥ０に割り付けられる。パラメータ値の計算のためファジィ演算を実行する時、各ファジィルールに対するメンバーシップ関数上の計算プロセスが各ＰＥにダイナミックに生成され、負荷分散が行われる。一方、同図（ｂ）は動特性モデル分割による並列化に関するものである。動特性モデルの分割されたエリア毎に、エリア内のシミュレーションを実行するプロセスが各ＰＥに割り付けられる。また各エリアに関連する動特性モデル、各エリア内のパラメータ値などはエリア毎に分散管理される。さらに各エリア境界に相当するパラメータ値は、スーパバイザにより管理される。ファジィ化定性推論に基づくシミュレーション手順の詳細は、文献［７］を参照されたい。

３．２　実験結果

図８（ａ）（ｂ）に示す２つの動特性モデルを利用して実験を行った。本モデルは、（ａ）復水器内の圧力に関する動特性と、（ｂ）復水器に冷却水を供給する冷却水ポンプシステムの吐出流量に関する動特性を示している。各冷却水ポンプ（a_cwp, b_cwp, c_cwp）に関する制御目標値、各冷却水バルブ（a_cwp_valve, b_cwp_valve, c_cwp_valve）に関する制御目標値は、プラント操作に関するＩＦ－ＴＨＥＮ形制御知識のＴＨＥＮ部で規定される入力値である。またその他の定数もプラント状態に応じてセンサから得られる入力値である。

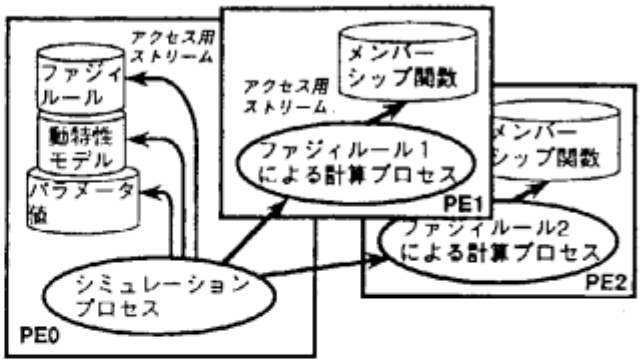
並列化ポイントに従って、実験結果を示す。

（１）ファジィ演算の並列化

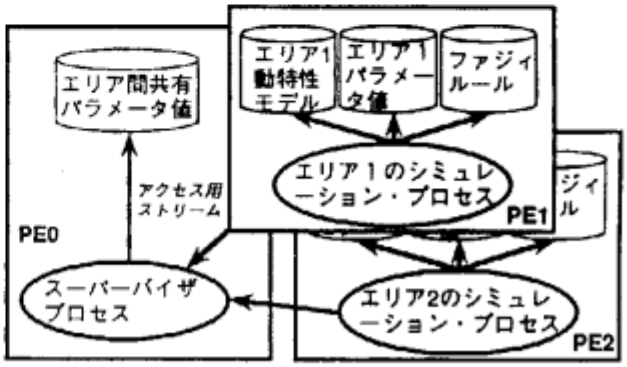
図８の矩形や丸印で示された部分がファジィ演算の対象となり、各演算に関してファジィルールが定義されている。この時、ファジィルール単位に演算プロセスをＰＥにサイクリックに負荷分散して実験を行った。その結果、負荷分散することにより総推論時間が逆に増加することが確認され、並列効果は得られなかった。

（２）動特性モデル分割による並列化

図８でハッチングを施した単位のエリア分割を一例として、いくつかのエリア分割パ

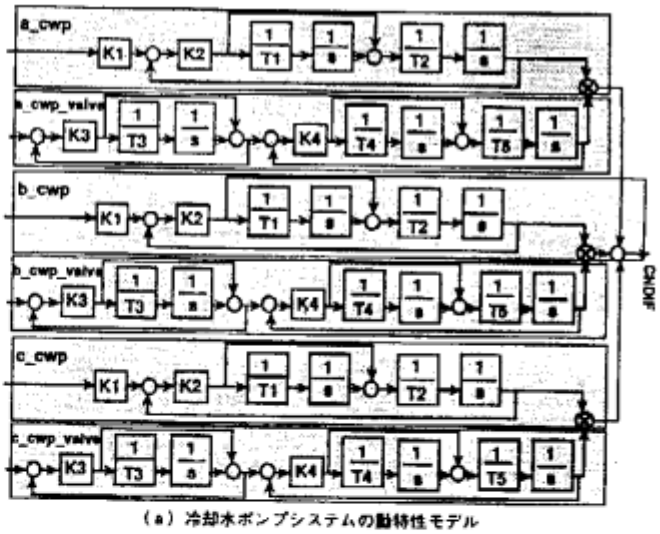


（ａ）ファジィ演算の並列化方式

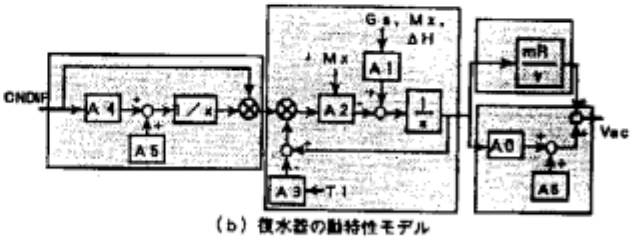


（ｂ）動特性モデル分割方式

図７ シミュレーション評価部並列化のプロセス構成とモデル・データ分散方式



（ａ）冷却水ポンプシステムの動特性モデル



（ｂ）復水器の動特性モデル

図８ 並列化実験の動特性モデル

ターンに関して実験を行い、各エリア毎にシミュレーション・プロセスを負荷分散して実験した。ＰＥへの割付はサイクリック方式を採用した。図８（ａ）（ｂ）それぞれの分割エリア数に応じてＰＥ台数を変化させ、５０ステップ分のシミュレーションを行った時の総推論時間を図９に示す。その結果、エリア分割することにより冷却水ポンプモデルの場合で約４．５倍、脱気器モデルの場合で約２．１倍の並列効果が確認できた。

３．３　実験評価

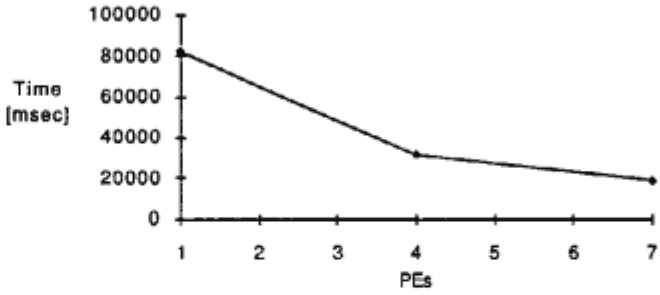
先に述べた実験結果に関して、以下の評価結果を得た。

（１）ファジィ演算の並列効果は認められなかった。これは、各ファジィルールのメンバシップ関数に基づく計算コストが非常に軽いため、負荷分散することによるＰＥ間通信コストの方が重くなり並列効果がでなかったと考えられる。したがって、本ポイントに基づく並列化はマルチＰＳＩ上では妥当な手法とは言えない。

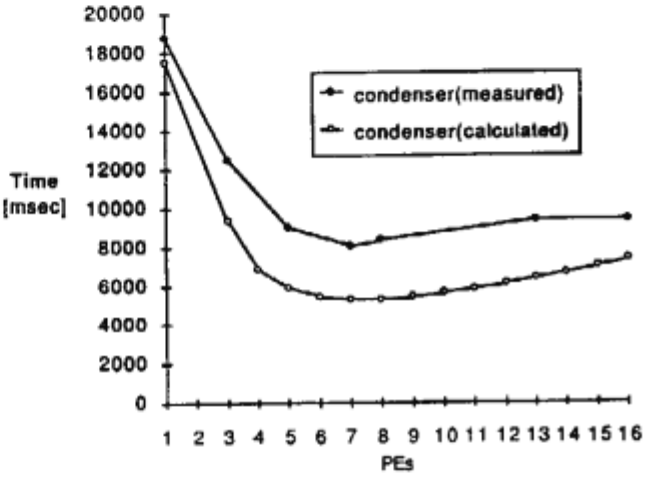
（２）動特性モデル分割において、並列に計算できるエリアに関しては負荷分散が可能である。図８（ａ）では、冷却水ポンプシステムに関する６つのエリアに関しては、それぞれの計算プロセスが異なるＰＥに割り付けられた時が最も並列効果が高くなることが確認された。

（３）同じく動特性モデル分割において、互いに入出力関係にあるようなエリアにおいてもパイプライン的な効果により並列効果が上がる部分がある。図８（ｂ）において、脱気器に関する４個のエリアでの各計算プロセスは、自己エリアの計算が終了しその結果を出力側のエリアにストリームを介して流した段階で、次の時刻の入力値に基づいた計算が可能である。このように各計算プロセスを異なるＰＥに分散することによりパイプライン的な並列効果を達成することが可能であり、これは実験においても確認されている。

（４）まとめるとシミュレーション部の推論高速化に関しては、動特性モデルの分割が有効な方法である。この時、並列に計算できるエリアのシミュレーション・プロセスを異なるＰＥに負荷分散することにより並列効果を上げることができる。さらにそれ以外の部分に関しても、パイプライン的な並列効果を達成できるように、動特性モデルを適当に分割できる可能性がある。



(a)冷却水ポンプモデルの場合



(b)脱気器モデルの場合

図９　動特性モデル分割による並列化実験結果

現在マルチＰＳＩ上で試作しているファジィ化定性推論システムにおいては、各ファジィ演算に対する推論時間が実験により確認されている。またデータ当りのＰＥ間通信時間は約２１０［ｍｓｅｃ／ｂ０ステップ］、データ当たりのスーパーバイザと各シミュレーション・プロセスにおける管理データのアクセス時間は無視できる程小さいことが同じく実験により確かめられている。詳細は省略するが、これらの事実から、本システムにおいてパイプライン的な並列効果を引き出せる動特性モデル分割基準として以下の指針が得られる。ただし、各エリアに関する計算コストは均等に分割されており、エリア境界には１つのパラメータを含み、通信コストはＰＥ間通信の回数の総和で近似できると仮定した。

$$T=T0*(S-1)+Ta+Tc \quad \cdots (1)$$

ただし、

T0：最も入力側のエリアに含まれる動特性モデルの全演算式の1ステップ分の計算コスト（単位はミリ秒／ステップ）

S　：シミュレーション・ステップ数

Ta：動特性モデル全体に含まれる全演算式の1ステップ分の計算コストの総計（単位はミリ秒）

Tc：エリア間の通信コストの総計。エリア分割数をN、データ当たり1回の通信コストTcomとすると以下の式で与えられる（単位はミリ秒）

$$Tc=2*Tcom*(N-1)*S$$

T　：モデルをN個のエリアに均等に分割して、Sステップのシミュレーションに要する総推論時間（単位はミリ秒）

（1）式で計算できるTを最小にするNの値がパイプライン的並列効果を最大にするモデル分割数であり、Ta／Nの値が均等に分割された時の各エリアにおける計算コストを示す。図8（b）の脱気器に関するモデルに対して、エリア分割の各ケースに応じて（1）式による計算値を図9（b）に合わせて表示した。実験で得られた値と計算値のずれは、実験では全てのプロセスを均等に分割することが必ずしもできなかったことが主因であると考えられる。ただし、（1）式により与えられる計算値は大体の傾向を予測するのに利用でき、エリア分割方法の決定に有効と考えられる。

4　おわりに

本稿では制御用エキスパートシステムのマルチPSI上での並列化に関して、2つのモジュールを取り上げ検討を行ってきた。これらのモジュールは、（a）プロセス間通信がない完全に独立したプロセスの並列化、または（c）パイプライン演算が可能なプロセスの並列化により、2倍から4倍程度の並列効果が確認できた大規模なKL1応用プログラム例になっている。本稿では詳しく説明しなかったが、その他のモジュールに関しても、上記（a）（c）の観点の他、（b）若干のプロセス間通信を伴うプロセスの並列化という観点においても、それぞれ並列効果が確認されている。（b）の一例としては、故障仮説生成部の並列定性伝播プロセスがある。各プロセスは定性因果モデルにおける関係式やパラメータに対応し、それぞれが定性偏差を内部状態として管理するプロセスとして実現されている。

これまでの試作を通じて、マルチPSIでのKL1応用プログラムに関して以下の知見を得ている。

（1）非常に感覚的ではあるが、プログラムの細かい処理部分の「精巧な負荷分散」を工夫しても、並列効果という点では効果が薄い。ある意味で「大胆に大雑把に負荷分散」する方が良さそうである。

（2）並列効果とは別の観点として、機能的に独立したプロセスが連携して一連の処理を行うようなシステムは、KL1プロセスのストリームによる同期メカニズムを利用するとかなり簡単にかつ美しくプログラミングが可能である。我々のアプリでは、浅い知識に基づいてプラント操作の推論と実行を行う正常時推論機構がこの例に当たる。

（3）制御用エキスパートシステムのKL1プログラミングに対しては、例題として与えた問題規模にもよるが、推論の高速化という観点では今一歩という結果であった。ただし先に述べたように、複数の推論プロセスからなるシステムのより自然なプログラミングという観点での効用は大きく、レスポンスやスループットの向上という効果もある程度得られた。今後はモデル規模の拡大に対して、さらにどの程度の推論高速化が可能なのかを理論的に検討し、実験により検証していきたい。

上に述べたように、マルチPSIおよびKL1のインパクトは強いものがあった。一方、システム実装面においては、いくつかの問題点も見られる。以下に特に目についたものを挙げて今後の改善を期待するとともに、本報告の締めくくりとしたい。

（1）KL1セルフコンパイラのレジスタ・オーバーフローの問題

KL1言語仕様上、述語の引き数が増加する傾向が強く、ベクタ等による引き数の構造化もボディ部では限界があり、また処理速度上の問題もある。さらに、KL1コードの自動生成機構を実現する場合には、この問題の存在は大きな制約となる。

（2）SIMPOSとのインタフェイスの問題

本格的なアプリケーションに対しては使い勝手のよいMMIが必須となる。MMIはSIMPOS上に構築することになるが、PIMOSとのインタフェイスが

ストリング上でしか提供されていない。したがって、リアルタイムに推論状況を表示するには速度的に問題があり、インプリメント上の大きな制約となる。

（3）ＰＩＭＯＳが良く落ちることの問題

システム開発過程において、シェルやリスナのプロセスだけでなく、ＰＩＭＯＳまで落ちることがある（ＣＳＰからのリブートができない）。エラーに対する処理系の強化を望みたい。

謝辞

本研究に関して、日頃ご助言を頂いたＩＣＯＴ第７研究室の新田室長、市吉室長代理に感謝します。また研究開発の機会を与えて頂いた当社システム・ソフトウェア技術研究所の西島所長ならびに研究第１部の河野部長に感謝します。

参考文献

[1]鈴木淳三,他:深い知識に基づく制御用エキスパートシステム－深い推論機構と詳細検証機構との融合－,第11回知識知能システムシンポジウム,pp.7－12 (1990)

[2]岩政幹人,他:制御用エキスパートシステム－定性的モデルに基づく診断機構－,第41回情全大,pp.(2－13)－(2－14) (1990)

[3]岩政幹人,他:深い知識に基づく制御用エキスパートシステム－定性的因果モデルとクラスタリングによる診断機構の開発－,第13回知能システムシンポジウム,pp.1－6 (1991)

[4]Peng,Y.,et.al :Basics of Parsimonious Covering Theory in Abductive Inference Models for Diagnostic Problem－ Solving, Springer－ Verlag, pp.49－98 (1990)

[5]鈴木淳三,他:深い知識に基づく制御用エキスパートシステム,第9回知識工学シンポジウム,pp.153－157 (1989)

[6]小沼千穂,他:深い知識に基づく制御用エキスパートシステム－操作条件生成機構の開発－,第12回知能システムシンポジウム,pp.13－18 (1990)

[7]小沼千穂,他:不測事態に対応するプラント制御用エキスパートシステム－定性推論を組み込んだ推論機構の開発－,第40回情全大,pp.298－299 (1990)

[8]Suzuki,J.,et.al :Plant Control Expert System Coping with Unforeseen Events - Model-based Reasoning Using Fuzzy Qualitative Reasoning -, Proc.of the Third International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems, pp.431-439 (1990)

[9]小沼千穂,他:制御用エキスパートシステム－詳細検証評価機構の並列化,第42回情全大,pp.(2-205)-(2-206) (1991)

適応型電子装置診断システムにおける並列処理

太田 禅† 大石 貞† 田中 淳†
田中 みどり‡ 中峯 洋一郎‡ 古関 義幸‡

†日本電気技術情報システム開発（株） ‡日本電気（株）

1 はじめに

従来のいわゆる診断エキスパートシステムには、(1) 知識ベースに記述されていない症例に対しては診断が不可能、(2) 熟練者の経験知識を獲得し、知識ベースとして整理することが困難である、といった欠点がある。筆者らはこれらの欠点を克服するシステムとして適応型電子装置診断システムの研究を進めてきた。開発した実験システムでは、モデルベース診断を軸とし、故障箇所を見つけ出すために有効な複数のテストを評価し、その時点で最適なテストを選択する機能を実現した [2]。

また、診断処理の高速化を図るため、並列化の検討を行った。具体的には、診断のためのテストの生成、選択処理を並列化し、また、負荷分散の方式について検討を進めた。

以下、2章では適応型の電子装置診断方式について示し、その並列化方式・負荷分散方式について、それぞれ3章および4章で述べる。また、5章では、実験結果について記すとともに、考察を行う。

2 適応型電子装置診断方式

ここで、対象とする装置は、複数の部品から構成され、各部品間の接続（構造）が記述できること、及び各部品の入出力関係（動作）が記述できることを前提とする。また、各構成部品は、状態をもち、その出力値は、入力値と状態値とにより決まるものと考える。ここでは仮定として、複数の部品が同時に故障することはないものとする。

本診断方式は、(1) 診断対象の構造・動作を記述した設計知識、(2) テストとサービスとの関係を記述した知識や部品の壊れやすさを記述した経験知識、(3) 症状やテスト結果をうけて故障部品を絞りこむ診断モジュール、(4) 設計知識・経験知識を利用して次におこなうべきテストを生成・選択するテスト生成選択モジュール、および (5) 診断結果から経験知識を学習する学習モジュールからなる。

図1に示すように、本システムによる診断は原因候補をテストによって消去していくことにより行う。診断の初めに経験的知識を用いて各部品の推定故障確率を計算する。次に、与えられた症状と設計知識（構造・動作に関する知識）を用いて故障の疑いのある部品のリスト（Suspect List, 以後 SL と略）を作成する。もし、$|SL| = 1$ であれば、単一故障を仮定しているので故障部品を特定できたことになり、診断は終了する。一方、$|SL| > 1$ の場合には、さらに SL 中のどの部品が壊

れているのかを調べる必要がある。そこで、システムは最も適切と考えられるテストを選択し、そのテストの実行を指示する。それに対する観測結果を用いてSLの更新を行う。このような操作を繰り返して診断を進めていく。従って、効率よく診断を進めるためには、適切なテストの生成と選択が必要である。

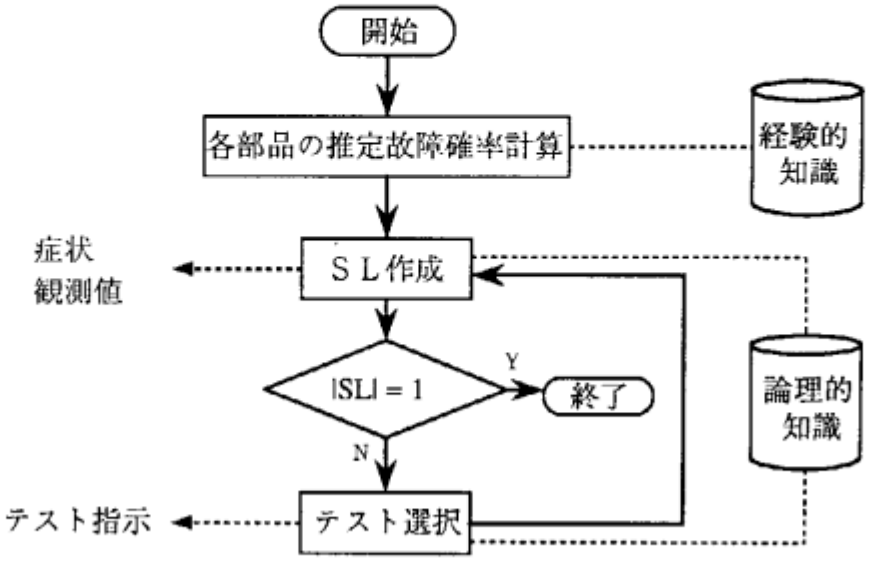


図1　診断の流れ

生成されたテストの中から、原因候補を絞り込むのに適切なテストを選択する方法について述べる。本システムは、選択の評価尺度として、テストの有効度とテストのコストを用いている。テストの有効度は、各原因候補の故障確率を基に計算した「得られる情報量の期待値」で表す。これにより、原因候補の絞り込みの際、無駄なテストをせずに効率的な診断を行える。また、テストのコストは、テストの実行に必要な時間としてとらえ、コストの高いテストが後回しになるようにする。テストの評価値は有効度をコストで割った値として定義する。

2.1　テストの有効度

一般に、テストを行うとSL内の部品（故障の疑いのある部品）が次第に絞り込まれて診断が進んでいくが、これは別の見方をすれば、テストの結果としてある情報が得られ、それによって診断が進むと考えることができる。つまり、得られた情報量が多ければ、それだけ診断が進むことになる。

こうした観点から、診断途中の各状態に対してそのエントロピーを定義し、テスト後の状態のエントロピーの期待値が最も小さくなるテストを実行するという方式をとる[1]。本システムでは、この情報量の計算に利用する故障確率を過去の経験から推定することにより、対象装置の故障分布に対応するテスト戦略を学習することを可能にした[3, 4]。

まず、診断の途中状態におけるエントロピー $E(SL)$ を以下のように定義する。その状態での原因候補集合 SL を、

$$SL = \{S_1, S_2, \ldots S_n\}$$

と表し、各要素 $S_i(i=1,n)$ の疑わしさ（故障確率）を p_i ($\sum p_i = 1,\ p_i > 0$) と表す。この状態でのエントロピーを、

$$E(SL) = -\sum_{i=1}^{n} p_i \log p_i$$

のように計算する。さらに、あるテスト T の結果 $t_j(j=1,k)$ が得られた時の SL を SL_j、それぞれのテスト結果 t_j が得られる確率を q_j としたとき、そのテストの効果 $gain(T)$ を次のように計算する。

$$gain(T) = \sum_{j=1}^{k} q_j (E(SL) - E(SL_j))$$

これは Quinlan の決定木の学習アルゴリズム ID3 [5] で用いられている gain 関数と同じ考え方によるものである。各テスト T について $gain(T)$ を求めた時に、最も大きな gain を持つテストが最も有効なテストと考える。

3 適応型電子装置診断における並列処理

一般に、並列処理を行なう上で問題となるのは、どの部分をどう並列処理にするかということと、並列化された部分の処理をいかに負荷分散させるかということである。そこで、本システムを並列化するうえで、まずはじめにシステムの診断過程から並列化の可能性を検討した。この適応型電子装置診断システムの診断過程は以下の 7 つの場面にわけられる。

1. 故障発生
2. 異常値（正常値）検出
3. 被疑部品抽出
4. テスト生成（テスト生成、テストの有効性の検証）
5. テスト評価
6. 実行テスト選択
7. テスト実行

このうち、7 のテスト実行を行なうことにより 2 の異常値 (正常値) 検出が再び行なわれることになり、被疑部品の抽出が再度検討される。以降、故障部品が確定するか実行できるテストが尽きるまでこの過程が繰り返される。この時、2 の異常値 (正常値) の検出から 7 のテスト実行にいたる過程では、ひとつ前の場面での結果が得られない限り次の場面に移行することができない。つまりこのレベルでの診断処理については並列化を行なうことはできない。しかしながら、各場面ごとにその診断内容を分析していくと、

1. 上記 4 のテスト生成において生成されるテストの有効性を検証する処理

2. 上記5のテスト評価における各テストの評価値を求める処理

において並列化が可能であることがわかる。なぜなら、この場面で扱うテストは膨大な数に上り、それらのテストの有効性の検証や評価過程における処理は各テストについて独立であるからである。加えて、この部分を並列処理化することは比較的容易であり、かつ十分効果をあげることが可能であると考えられる。さらに、テスト生成からテストの有効性の検証、そして、生成されたテストの評価に至る過程では各テストの処理をパイプライン的に行えることがうかがえる。これらの理由により、テスト生成を行なう部分から評価にいたる部分までを並列に処理することとした。

4 負荷分散

次に、負荷分散の方法について述べる。たとえ並列に処理できるとわかっても、各プロセッサの負荷が均等にならなければ使用するプロセッサの台数にみあった効果をあげることはできない。そこで、負荷分散を行なう単位について考える。本システムで扱っている診断対象装置では、動作のシミュレーションに要する計算量が入力値によらずほぼ一定である。そして、テストの生成やその評価は診断対象装置の構造と正常動作のシミュレーションをもとに行われており、テストの有効性の検証と生成されたテストの評価に内在する並列処理が可能な部分の処理の大きさは、それぞれある入力値に対する診断対象装置の動作のシミュレーション1回分の処理の大きさにほぼ比例する。このことから、テストの有効性の検証およびテストの評価にかかる処理に要する計算量もテスト内容によらずほぼ一定となる。そこで、このテストの有効性の検証とテストの評価の処理をひとつの単位とし、それらを複数のプロセッサに均等に割り振ることにより、容易に負荷の均等化を計れると考えた。以下にテスト生成および有効性の検証からテストの評価にいたる過程の並列化および負荷分散のイメージを示す。

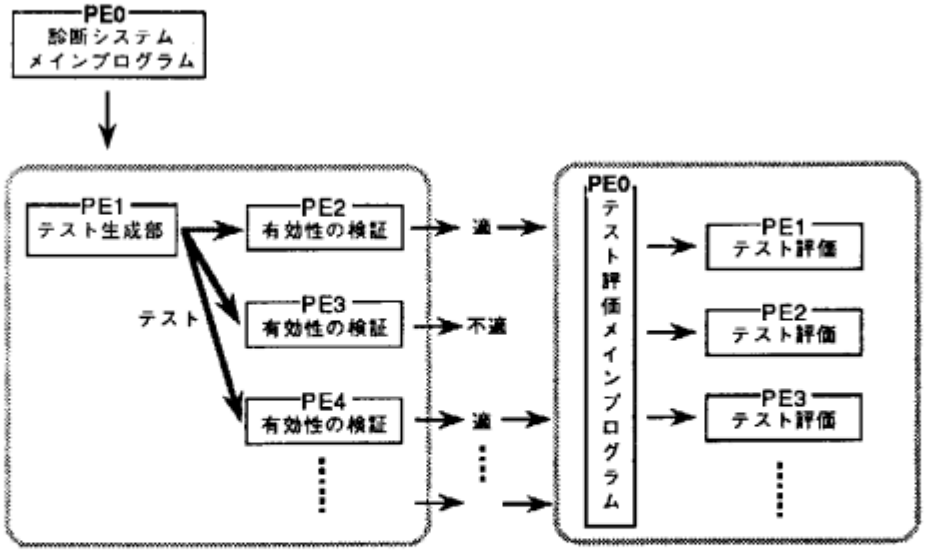


図2 並列化負荷分散イメージ

各処理のプロセッサへの割り付けを早く行うために、診断システムのメインプログラムは常にPE0におかれる。テスト生成はPE1でおこない、作成したテストから順にその他のプロセッサで、テストの有効性を検証する。有効であると確認されたテストはPE0上のプログラムによって、再度PE1から各プロセッサに割り振られ評価値を求める処理が並列に行われる。テストの有効性の検証およびテストの評価部分はテスト1個を単位として各プロセッサに渡される。

5 結果・考察

以上述べた方式に基づき適応型電子装置診断実験システムをMulti-PSI上に開発した。並列化の評価には、診断対象装置として「ビルの空調管理システム」(部品数18)と「パケット交換機」(部品数68)を使用した。実験システムの画面例を図3に示す。

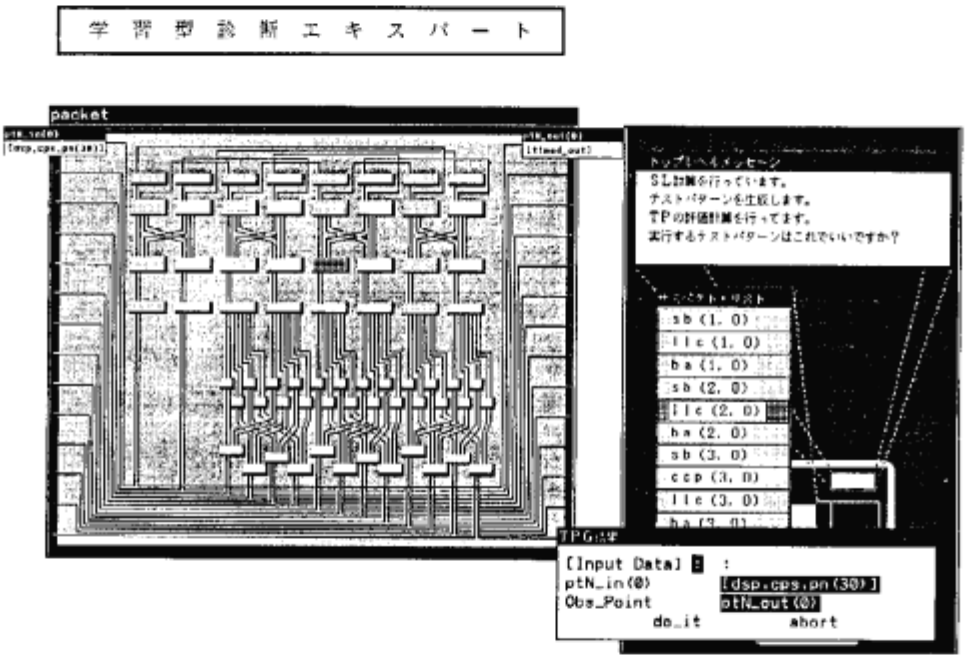


図3 画面例

はじめに、「ビルの空調管理システム」を診断対象装置とした場合について、前述した負荷分散方式により使用するプロセッサ数を変えて診断時間を測定した結果が表1である。実験に使った故障状態に対して生成されるテスト数は初期症状に対して2436個であり、有効性の検証を終えて実際にテストとして使えるのは166個である。次にこの166個のテストが評価部分に渡され、最終的には評価値の一番高いものがテストとして診断対象装置に与えられる。以降、診断が終了するまでこの処理は繰り返される。

表1　使用プロセッサ数毎の診断時間

PE数	1	2	3	4	5	6	7	8
診断時間(sec)	14.62	17.85	23.24	14.72	12.01	11.02	10.85	10.73
PE数	9	10	11	12	13	14	15	16
診断時間(sec)	10.67	10.74	10.75	10.72	10.75	10.84	10.85	10.87

＊　PIMOS2.5版を使用

＊　PE数1：　全ての処理をPE0上で行う。

PE数2：　テスト生成・評価部分をPE1上で、それ以外をPE0上で行う。

PE数3以上：図1の負荷分散イメージに従う。

表1から、使用したプロセッサ数が1台、2台の時と3台の時を比較すると、1台、2台の方が速いことがわかる。また、今回の負荷分散方式では使用するプロセッサ数が9台の時が最速であり、それ以上プロセッサ数を増やしても診断時間の向上はみられない。

使用したプロセッサ数が1台、2台の時には負荷分散の方法がそれ以外の台数の時と違い、テスト生成と評価部分が同一プロセッサ上で処理される。この場合並列化された部分に関してプロセッサ間の通信・データの転送などが起こらないために速くなったと思われる。これに対して、使用したプロセッサ数が3台以上の場合には、データの存在するプロセッサ上で処理が行われるとは限らないため、プロセッサ間の通信・データの転送などが発生する。しかし、使用したプロセッサが5台以上の時には、1,2台の時と比べても速くなっており、並列化の効果が現れた結果となっている。

次に、台数効果について述べる。使用したプロセッサ数が3台の場合の診断時間を基準としてプロセッサ数の増加による効果を調べた。各診断時間の内容を分析し、診断時間の実測値から並列化された部分の実行に要する時間の推定を行い、台数の効果を求めた結果が図4である。

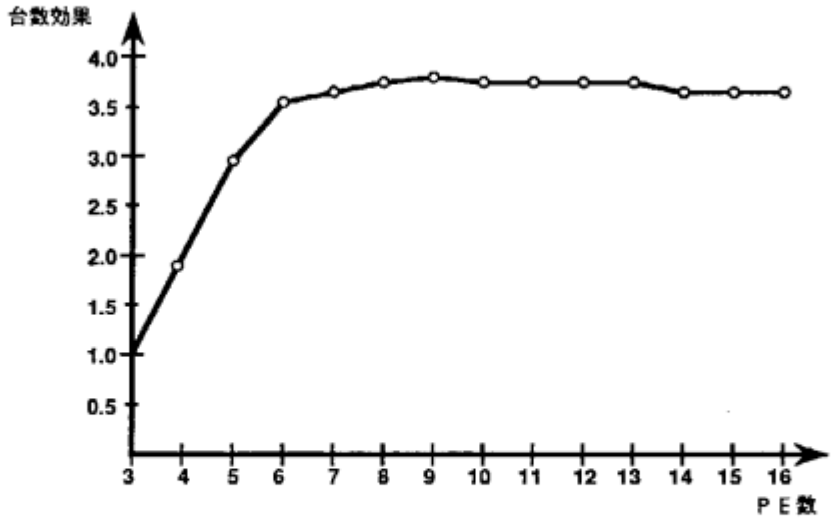


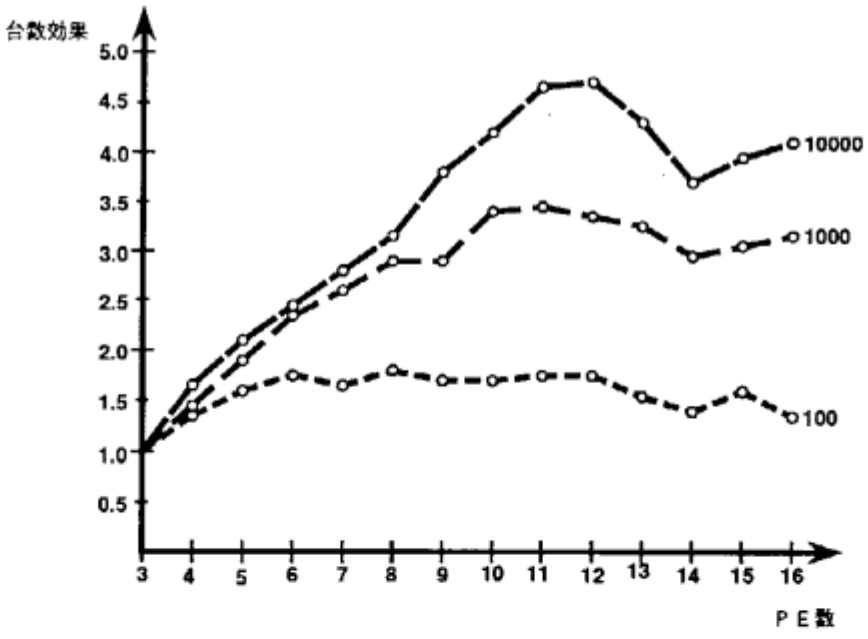
図4　台数効果

この図から、台数効果としては5台ぐらいまでが使用台数分の効果を示し、以降はほぼ平衡状態になることがわかる。

また、並列化された部分とそれ以外の部分の時間的な構成を検討した結果、診断対象装置に「ビルの空調管理システム」を扱った場合の本診断システムは、今回の並列化方式の場合最低でも約10秒（PE0上で動作するプログラムおよび他プロセッサへのデータの転送、並列化された部分の最短時間などから推定）を診断に要することがわかった。このことから、表1の「使用プロセッサ数毎の診断時間」で示された結果は、本システムで行える最短の診断時間を実現したことになる。

では、診断対象装置の規模の拡大などに伴うテスト数の増加が起こった場合、この並列化の効果はどう変化するのであろうか。現在診断対象装置として扱っている「パケット交換機」の動作のシミュレーションを使用し、テスト数の変化と台数効果について調べた結果を図3に示す。

これは、PE0にメインプログラムを置き、その他のプロセッサで「パケット交換機」の動作のシミュレーションを行ったものである。「パケット交換機」の動作のシミュレーションにかかる処理の大きさも、「ビルの空調管理システム」の場合と同様に入力値に関わらずほぼ一定であり、前述したようにテストの有効性の検証および評価部分にかかる処理の大きさが診断対象装置の動作のシミュレーションにかかる処理にほぼ比例することから、このモデルの並列化の効果を調べることにより、診断システム上で実現した際の並列化の効果を推定することができる。



＊　PIMOS 2.1版を使用

＊　PE数2台の時の処理時間を得られなかったためプロセッサ数3台の時の処理時間を1とした比率である。

図5　テスト数の変化と台数効果

図5より、テスト数が増大するにつれてより多くのプロセッサ数まで高い台数効果が得られることがわかる。この結果から診断対象装置の規模の拡大などに伴うテスト数の増大に関しては、今回採用した並列化の方法が有効であると言える。

また、今回の並列化の方法をとった場合の利点として、テストの総数や各プロセッサに与えられるテスト数を管理する必要がないことがあげられる。つまり、並列化の方法に関しては診断対象装置が変わったとしても、プログラムのメンテナンスの必要がほとんどいらないことになる。

6 おわりに

多様な症状に対応可能なモデルベースの適応型診断方式及び、その実験システムを開発した。また、同システムにおいて並列化を検討し、診断処理部の並列化を行った。さらに、負荷分散をうまく行うことで並列性が引き出せることが確認された。

謝辞

本研究は第五世代コンピュータプロジェクトの一環として行ったものである。日頃お世話になっている（財）新世代コンピュータ技術開発機構の新田室長に感謝いたします。

参考文献

[1] de Kleer, J. and Williams, B. C., "Diagnosis with behavioral modes," *Proc. IJCAI-89,* Vol. 2, pp. 1324–1330, 1989.

[2] Koseki, Y., Nakakuki, Y., and Tanaka, M., "An adaptive model-Based diagnostic system," *Proc. PRICAI'90,* Vol. 1, pp. 104–109, 1990.

[3] Nakakuki, Y., Koseki, Y., and Tanaka, M., "Inductive learning in probabilistic domain," *Proc. AAAI-90,* Vol. 2, pp. 809–814, 1990.

[4] 中辻洋一郎、古関義幸、田中みどり「確率モデルの学習方式と診断への応用」情報処理学会研究報告 *(91-AI-74)* Vol. 91 (3), pp. 19–28, 1991.

[5] Quinlan, J. R., "Induction of decision trees," *Machine Learning,* Vol. 1 (1), pp. 81–106, 1986.

KL1 上の並列プロセス指向言語 *AYA*

寿崎 かすみ 近山 隆
(財) 新世代コンピュータ技術開発機構
三田国際ビル 21 階
東京都港区三田 1 丁目 4-28
03-3456-3193, susaki@icot.or.jp

概要

プロセス指向プログラミングは Shapiro and Takeuchi [3] の論文で提唱された手法で, KL1 のような並列論理型言語で幅広く使用されている.

'プロセス' の概念を用いたモデル化はプログラムを構成するうえでのエレガントで強力な方法論である. この方法では, 再帰呼びだしにより繰り返されるゴールの実行をプロセスとみなし, そのゴールの引数をプロセスの状態とみなす. しかし, これはプログラムを解釈する方法にすぎずプログラミング言語としてサポートされていない. このため, 実際のプログラミングは煩雑になる. たとえばプロセスの状態を表す引数はすべて, 毎回述語のヘッド部および再帰呼びだしのゴールに記述しなくてはならない. また, プロセスの状態を表す引数を 1 つ増やすことは, プロセスの概念上ではささいな変更であるが, 実際のプログラムにおいてはほとんどすべての述語の引数を 1 つずつ増やすという煩雑な変更になり, この修正は新しいバグがはいる原因となる.

プログラミング言語 *AYA* はこのような問題を解決するために設計したものである. *AYA* のプログラムは KL1 のプログラムに簡単にコンパイルできるが, 状態をもったプロセスの概念を言語の基本要素として提供しているので, 前述したようなプログラミング上のささいな問題にわずらわされずに負荷分散の方法などもっと大局的な問題に集中することができる.

1 はじめに

プログラミング言語 KL1[1] は GHC(Guarded Horn Clause) に, 実用的な大規模システムを記述するための拡張を行った並列論理型言語である. 並列推論マシンのオペレーティング・システム PIMOS[2] は全面的に KL1 で記述している. この上で動作するアプリケーション・プログラムもまた KL1 で記述している.

プロセス指向プログラミングの手法は, KL1 プログラミングで広く使われているテクニックである. この方法では, 相互にメッセージを交換しながら並列に動作する協調的なプロセスとして問題をモデル化しプログラムを記述する. これによりプログラムのモジュラリティを高く保つことができる. また, 並列性の制御が容易に行える. PIMOS においては, このモデルをはじめから設計に取り入れている. たとえば, PIMOS の管理プロセス (e.g. デバイス・プロセス) は集中管理を行うことによりおこるボトルネックを避けるために分散している.

しかし, このテクニックは単なるプログラミング・テクニックにすぎず, このモデルに基づいてプログラムを直接記述するための言語上のサポートはない. このためプログラミングをする上での煩雑さがある. たとえば, プロセスの状態を表す KL1 述語の引数を毎回記述しなくてはならない. また, 新しい引数を加えるためにはすべての述語を修正しなくてはならないなどである. また, プロセスの概念をデバッガに反映しにくいという問題もある.

AYA は, これらの煩雑さや問題を軽減することを目的としている. このため *AYA* では, プロセスおよびプロセス間の通信を言語の構成要素として, そのまま記述することができるようにした. 次章以降で, 言語・文法および実装の方針について述べる.

2 言語仕様

2.1 特徴

KL1のプロセス指向プログラミングの手法では，プロセスは再帰呼びだしを行うゴールで実現していた．しかし，*AYA*ではプロセスを言語の構成要素として記述できるようにした．プロセスはまたプログラムの実行の単位でもある．*AYA*のプログラムは，クラスを単位として記述する．

*AYA*で記述した簡単なプログラムを次に示す．

```
class counter(In)
    with +in := In , +state := 0.
input in.
 :up -> @state := ~(@state + 1).
 :down -> @state := ~(@state -1).
 :show(Current) -> Current <- @state.
 :/ -> continue \\ .
end class.
```

このプログラムはカウンタのプロセスのプログラムである．このプロセスは，counterという名前のクラスで定義されている．このプロセスの起動は，プロセスがメッセージを受けとるためのストリームInを渡して行う．カウンタのプロセスは，内部状態としてそのときのカウンタの値を保持するために状態変数'state'を持つ．この変数ははじめ0に初期化されている．:up, :down, :show(Current), :/ が，このプロセスの受信することのできるメッセージである．各メッセージに対して，そのメッセージを受けとったときに行う動作が定義してある．このプロセスは:upを受けとると状態変数stateの値を1増やし，:downを受けとると1減らす．:show(Current)を受けとるとそのときの値をメッセージの引数'Current'に返す．:/ と記述してあるのは，closeメッセージである．これを受信するとプロセスは実行を終了する．受信メッセージとそれに対応した処理の定義をメソッドと呼ぶ．

このプログラムと同じものをKL1で記述した例を次に示す．

```
counter(In):- true |
        counting(In,0).
counting([up|In],State):- true |
        New := State + 1,
        counter(In,New).
counting([down|In],State):- true |
        New := State - 1,
        counter(In,New).
counting([show(Current)|In],State):- true |
        Current = State,
        counter(In,State).
counting([],State):- true | true.
```

2.2 クラス・シーン

クラスは，その内部にシーンを持つ．シーンは任意段数ネストできる．

すべてのクラスが必ず持つシーンとしてinitial sceneがある．その他のシーンはこのinitial sceneにネストしたシーンとして定義する．この，クラス・initial scene・シーンの関係を図1に示す．

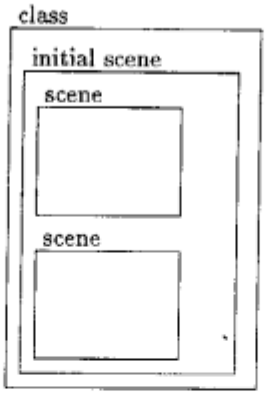


図 1: クラスとシーン

クラスおよびシーンを持つプログラムの例として, :start メッセージを受信してから :stop(Amount) メッセージを受信するまでに到着した整数を足し算し, その結果を引数 Amount に返すプログラムを示す.

```
class sigma(In)
    with +in := In, +amount := 0
    \\ wait_start.

scene wait_start.
input in.
 :start -> continue \\ adding.
 :/ -> \\ .
end scene.

scene adding.
input in.
 :stop(Amount) -> Amount <- @amount, @amount := 0
        \\ wait_start.
 :N -> integer(N) | @amount := @amount + N.
end scene.
end class.
```

クラス sigma は initial scene の中に, wait_start と adding の 2 つのシーンを持つ. このプロセスは:start メッセージを受けとるとシーン adding に移動する. :stop(Amount) が到着するまでのあいだに受けとった整数の和を求め, :stop(Amount) を受けとるとそれまでの合計を引数にかえし, 再び:start が来るのを待つ.

このように, クラスはシーンで構成する. 各シーンはそのクラスで定義したプロセスの状態 (この例では, start メッセージの待ち状態と到着する整数を足していく状態の 2 つ) に対応する. 受けとり可能なメッセージはこの例のように, シーンごとに異なって定義できる.

このようにシーンという概念を導入したことにより, 良く似たクラスを何度も生成したり終了したりするオーバーヘッドを軽減することができる.

2.3 メッセージ・ライン・ソケット

プロセス間の通信に使用するデータをメッセージと呼ぶ. メッセージとして, 具体化された任意の KL1 データを使用できる.

メッセージは送り手から受け手へラインを介して運ばれる. *AYA* のラインは KL1 の変数に対応する. プロセスは, ラインを保持するためのホルダーとしてソケットを持つ. ソケットに保持しているラインにはソケット名でアクセスする.

ソケットには入力用と出力用がある. 入力用ソケットはメッセージの受信のために使用し, 出力用ソケットはメッセージの送信に使用する. メッセージは出力用ソケットから送信し, そのソケットとラインを共有している入力用ソ

ケットで受信する．プロセスは複数の入力用ソケットを持ち，そのそれぞれに対してメソッドを定義することができる．このため，あるソケットに到着するメッセージ列の処理の途中でも他のソケットに到着したメッセージつまり別のルートからのメッセージを受け付けることができる．このことを利用して割り込み処理の記述が行える．

ラインはメッセージを1つだけ運ぶことができる．1つのソケットでメッセージ列の送受信を行うためにはメッセージを1つ送受信するたびに，送信・受信の両方が新しいラインをセットする必要がある．そこでメッセージ列のためには，リストセルに送りたいメッセージと次に使う新しいラインをつめて，1つのメッセージとして送るという方法を用いる．

この方法を用いると，前述の counter のプログラムは次のようになる．

```
class counter(In)
    with +in := In , +state := 0.
input in.
 [up |In] -> @state := ~(@state + 1), @in := In.
 [down |In] -> @state := ~(@state -1), @in := In.
 [show(Current) |In] -> Current <- @state, @in := In.
 [] -> continue \\ .
end class.
```

はじめのメッセージ，[up|In] に対する処理を見てみよう．

```
@state := ~(@state + 1), @in := In
```

ここでは，ソケット state の値を1増やし，ソケット in にリストセルの Cdr として運ばれてきた次のライン In をセットしている．一般にソケットの値の更新は，':=' で行う．

このように，リストセルにメッセージと次に使用するラインをつめて1つのメッセージとしてやりとりする方法をストリームと呼ぶ．このストリームの用法を記述するために，[up |In] を :up のように記述するシンタックスを提供している．ストリームによる通信を終了するために，close message を使用する．close message を送ることを 'ストリームを閉じる' といい，受けとることを 'ストリームを閉じられた' という．close message として [ ] を使用する．

ソケットは，クラスおよびシーン定義のはじめのところでソケット名とソケットのモード (入力か出力か) を指定して定義する．併せて初期値を指定することもできる．初期値の既定値は [ ] である．シーンはすべてのクラスが必ず持っている 'initial scene' の中に定義されているので，クラスで定義したソケットはそのクラスで定義されているすべてのシーンからアクセスできる．

2.4 ターム・変数・ユニフィケーション

具体化された KL1 のタームは，その値を持ったラインのことである．変数は，ラインを意味する．同じ変数は同じラインを表す．変数のスコープはたとえばメソッドの中である．

2つのラインを結合する処理をユニフィケーションと呼ぶ．ユニフィケーションは次のように記述する．

```
Out <- In
```

2つのライン Out と In が結合される． counter の例では，ユニフィケーションを用いてそのときの状態変数の値をメッセージ :show(Current) の引数 Current に返している．

```
Current <- @state
```

ソケットからのメッセージの送信も，ユニフィケーションを用いて記述する．ソケット out からメッセージ end を送信することを次のように記述する．

```
@out <- end
```

2.5 メソッド定義

シーンは複数の入力用ソケットを持つことができるので，受信したメッセージに対する処理はメッセージの到着したソケットの名前とメッセージの組に対して定義する．これを，メソッド定義と呼ぶ．

カウンタのプログラムの例を少し変更して，:down を受けとったときの処理が状態変数 state の値によって2通りに定義する．

```
input in.
 :down -> @state > 0 | @state := ~(@state - 1).
 :down -> @state = 0 | continue.
```

上の例の,

```
input in.
```

を基本ソケット宣言と呼ぶ. これは, メッセージの到着するソケットを宣言している.

```
:down
```

は, ソケット in にメッセージ:down が到着するというイベントのことである. つぎの

```
@state > 0
```

は条件チェックである. これをコンディションという. メッセージが基本ソケットに到着してパタンマッチに成功し, コンディションを確認して満たされていれば対応して定義されている処理を実行する. この処理をアクションという. ここでは

```
@state := ~(@state - 1)
```

および

```
continue
```

がそれぞれアクションである. 'continue' は何もしないこと (KL1 のボディの true) を意味する.

このあと必要に応じて, 他のシーンに動くことを指定できる. たとえば, sigma の例では,

```
 :start -> continue \\ adding.
```

のように記述して:start メッセージを受けとった場合は, シーン adding に移動するよう指定している. これを, 'つぎのシーンの指定' と呼ぶ.

このようにメソッドは, イベント, コンディション, アクション, 次のシーンの指定により定義される. イベントおよびコンディションは省略することができる. イベントが省略された場合は, 直接コンディションを調べる. コンディションも省略されているときはアクションがいきなり実行される. イベントは省略されてコンディションが記述されているときは, コンディションを調べ, 条件を満たしていればアクションを実行する. アクションとしてなにも行わないときは continue と記述する. 次のシーンの指定は, 他のシーンに移動するときのみ記述する. プロセスの実行を終了する場合は, 言語定義のシーン terminate へ移動する.

1 つのソケットへのアクセスが 1 つのメソッド中に複数回出現する場合は, ソケットは出現順序に従って更新される.

3 ストリームのサポート

3.1 ストリーム型メッセージ

リストセルに次に使うラインも詰めて 1 つのメッセージとして扱うものを, ストリーム型メッセージと呼ぶことにする.

ストリーム型のメッセージのリストセルを毎回記述するかわりに, リストの Car としてセットするメッセージの前に ':' をつけることにする.

```
:up
```

は, up がストリーム型のメッセージであることを示す. このシンタックスを用いるとストリーム型メッセージの列はつぎのように記述できる.

```
:up:down:show(Current)
```

ストリームを閉じる close message は, つぎのように書く.

```
:/
```

3.2 メッセージの送受信

メッセージの送受信もストリーム型メッセージのシンタックスを用いてそれぞれつぎのように記述できる.

- メッセージ送信.
  メッセージの送信は, ユニフィケーションを用いて次のように記述する.

  ```
  @out <- :up:down:show(Current)
  ```

  これは, ソケット out からメッセージ up, down, show(Current) を送信してストリームを閉じることを意味する. 変数で表されるラインに対しても同様に

  ```
  Out <- :up:down:show(Current)
  ```

  のように記述する. これは, 変数 Out で示されるラインにストリーム型のメッセージを 3 つ送ってストリームを閉じることである.

  close message のみの送信も同様に

  ```
  @out <- :/

  Out <- :/
  ```

  などと記述する.

- メッセージ受信.
  - イベントの場合.
    基本ソケットに到着したメッセージとのパタンマッチを行う. :up に 続く要素がソケットにセットされる.

    ```
    :up
    ```

    close message の場合は次のようになる.

    ```
    :/
    ```

  - コンディションの場合.
    到着したメッセージとのパタンマッチを行う. :interrupt につづく要素が ソケットにセットされる.

    ```
    @in2 = :interrupt
    ```

  close message の場合は次のようになる.

  ```
  @in2 = :/
  ```

3.3 ストリーム操作

複数本のストリームをマージして 1 本にする, 2 本のストリームをつないで 1 本にするといった操作を行うために, merge および concatenate プロセスがある.

- `merge(In,Out)`

  merge プロセスは, ストリームのマージャとして機能する. In が入力ストリーム, Out が出力ストリームである. In にベクタが渡されると, その要素が入力ストリームとして加えられる. 入力ストリームの 1 本とベクタのユニフィケーションを ' マージイン ' と呼ぶ. マージインをつぎのように書く.

  ```
  @out <= Out2
  ```

  これは, 現在ソケット out に入っているラインと {Out1,Out2} をユニフィケーションして, Out1 をソケット out の新しい値としてセットすることである.

  マージインは, 入力・出力どちらのソケットにも行える.

- concatenate(Stream1,Stream2,Stream3)

  concatenate プロセスは，ソケットに保持されているストリームともう1本のストリームを結合し，その結果をソケットの新しい値とする．この2本のストリームのどちらを前にするかにより2通りのつなぎかたがある．

  - ソケットに入っているストリームを前にする．

    concatenate(@socket,Stream,New)

    この場合をアペンドと呼び，つぎのように記述する．

    @socket <=< Stream

  - ソケットに入っているストリームを後ろにする．

    concatenate(Stream,@socket,New)

    この場合をプリペンドと呼び，つぎのように記述する．

    @socket <<= Stream

アペンドおよびプリペンド操作は，入力・出力どちらのソケットに対しても行える．

入力ソケットに対するアペンドは，次に読み込むメッセージ列を指定することである．出力ソケットにアペンドすると，そのソケットから送信したメッセージは一旦 concatenate プロセスに送られ，その後，先につながるプロセスに届けられることになる．

入力ソケットに対してプリペンドするとプリペンドされたストリーム中のメッセージが，そのソケットから次に読み込まれるメッセージとなる．出力ソケットに対するプリペンドは，ストリームを共有する入力ソケットにそのストリームを渡すことになるので，ストリーム中のメッセージを送り届けることになる．このようなことから，プリペンドによりメッセージの送信を行うことができる．プリペンドによるメッセージの送信はユニフィケーションの場合と異なり，メッセージを送信したあとストリームを閉じない．

4 実装

AYA のプログラムは，KL1 プログラムにコンパイルして実行する．コンパイル結果の KL1 プログラムはプロセス指向のプログラミング・テクニックで記述したプログラムに近いものになる．

ソケットおよびラインは KL1 述語の引数に展開する．メッセージの待ち合わせは，KL1 の同期のメカニズムで実現する．他のプロセスの初期化と次のシーンへの移動は同じボディ・ゴールとなる．

AYA では，ソケットに対して入力あるいは出力のモードを定義した．また言語定義プロセスの引数には，入力あるいは出力のモードが定まっているものもある．これらの情報を使用して，モードの定義されていない変数のモードを定め，その整合性を調べることができる．これによりバグのもととなる変数を見つけることができる．

5 今後の課題

最大の課題は，継承機構の導入である．継承機構としては，複数のクラス間でメソッド定義を共有できるようなものが望ましい．しかし *AYA* では，クラスの下にシーンという概念を導入したこと，入力として複数のソケットを扱えるようにしたことから，共有したい定義がどれであるかを特定することが難しい．また KL1 にコンパイルして実行するため，実現上の制約もある．

このような条件のもとで継承機構を導入するための1つの方法として，ある入力ソケットに対するメソッド定義を独立したテーブルに定義し，このテーブルを共有するという方式を検討している．このテーブル間での継承も行える．基本的な方式はほぼ決定しているが，このための文法などの検討は，今後行う必要がある．

このほか，実際の大規模プログラムを書くにあたってはデバッグ環境の整備も必要となる．また，文法の見直しも必要となるかもしれない．

6 おわりに

KL1 のプロセス指向プログラミングの手法をサポートするために，高水準言語 *AYA* を設計した．

KL1 ではプロセスの内部状態を表す変数の記述が煩雑であったが，これをソケットとして必要なときだけ記述することにした．また，再帰呼びだしにより永続するプロセスを基本としてこのためのゴール呼びだしを毎回記述する手間をなくした．このようなことから，*AYA* では KL1 に比べて簡明な記述が可能となった．

シーンの概念を導入したことによりメッセージごとに定義されるプロセスの動作をシーンを移ることにより変えることができるようになった．また，受信可能なメッセージそのもの，受信対象とするソケットも変えることができる．1つのシーンが複数のソケットを入力として用意できるので，あるソケットからメッセージ列を読み込んでいる間に，他のソケットに到着した，つまり他の経路からのメッセージを処理することもできる．

類似した研究として Vulcan[4], FLENG++[5], Polka[6] などがある．いずれも同じような動機にもとづいて同様のアプローチにより設計された言語である．これらとの比較において *AYA* の特徴はつぎのような点である．

- シーンの概念を導入したこと．
- 通信はラインにより行う．

参考文献

[1] K. Ueda and T. Chikayama. Design of the Kernel Language for the Parallel Inference Machine. *The Computer Journal*, 1990.

[2] T. Chikayama, H. Sato, and T. Miyazaki. Overview of the Parallel Inference Machine Operating System (PIMOS). In *Proc. of the International Conference On Fifth Generation Computing Systems 1988*, Tokyo, Japan, 1988.

[3] E.Y. Shapiro and A. Takeuchi. Object oriented programming in Concurrent Prolog. *New Generation Computing, OHMSHA Ltd. and Springer-Verlag*, 1(1):25–48, 1983.

[4] K. Kahn et al. Vulcan: Logical Concurrent Objects. In *Research Directions in Object-Oriented Programming*, Cambridge, Massachusetts, 1987. MIT Press.

[5] H. Nakamura et al. Object oriented language FLENG++ on concurrent logic programming language FLENG. In *WOOC'89 (in Japanese)*, 1989.

[6] Andrew Davison. Polka : A Parlog Object Oriented Language. Ph.d thesis, Imperial College, 1989.

A　プログラム例

AYA によるプログラムの例としてリーダーズ・ライターズ問題のプログラムを示す．この問題は共有資源アクセスの管理を扱ったもので，次のようなアルゴリズムに基づいている．

- 読みだし要求は，同時にいくつでも扱える．
- 書き込み操作が行われている間は，他の要求は一切受け付けない．
- 読みだし操作の間に書き込み要求がくると，その後あらたな読みだし要求は受け付けない．現在実行中の読みだし操作がすべて終了するのを待って書き込み操作を行う．

次のプログラムは，ファイルへの読みだし・書き込み要求を管理するプロセスである．プロセスは，readerswriters という名前のクラスとして定義されている．このプロセスは起動時に，ファイルへの要求がくるストリーム request，このプロセスからファイルデバイスへ要求を送るストリーム tofile，ファイルから管理プロセスへ処理の終了などを伝えるメッセージの流れるストリーム fromfile の3本を受けとる．

クラス readerswriters は，idling, reading, writing の3つのシーンを持つ．それぞれ，メッセージの到着待ち，読み込み処理，書き込み処理を行うシーンである．プロセスを起動するとまず idling のシーンでメッセージの到着を待つ．読み込み要求ならば reading に，書き込み要求ならば writing に移る．reading のシーンで読み込み要求を受けとるとそれはただちにファイルデバイスに送られる．書き込み要求の場合は reading の中で定義しているシーン waiting に移動し，そのとき実行されている読みだし処理がすべて終了するのを待つ．その後 writing に移って書き込み要求をファイルデバイスに送る．読みだし操作，書き込み操作が終了すると，idling に移動して次の要求を待つ．

```
module readerswriters.
public readerswriters/3.

class readerswriters(+request,-tofile,+fromfile)
```

```
  \\ idling.

  scene idling.
    input request.
      :read(Data) -> @tofile <<= :read(Data) \\ reading.
      :write(Data) -> @tofile <<= :write(Data) \\ writing.
  end scene. % idling

  scene reading
        with readers := 1.
      -> @readers = 0 | continue \\ idling.
    input request.
      :read(Data) -> @readers := ~(@readers + 1),
     @tofile <<= :read(Data).
      :write(Data) -> continue \\ waiting(Data).
    input fromfile.
      :readend -> @readers := ~(@readers - 1).

    scene waiting(+writedata).
        -> @readers = 0 | @tofile <<= :write(@writedata) \\ writing.
      input fromfile.
        :readend -> @readers := ~(@readers - 1).
    end scene. % waiting

end scene. % reading

scene writing.
  input fromfile.
    :writeend -> continue \\ idling.
end scene. %writing

end class. % readerswriters
```

並列データベース管理システムの問い合わせ処理

河村 元夫 (ICOT), 佐藤 裕幸 (三菱電機)

1 はじめに

並列推論マシン PIM 上で動作する並列データベース管理プログラム Kappa-P の研究開発をおこなっている. Kappa-P は、中期成果である分散知識ベース管理基本ソフトウェア・モジュール Kappa のデータベース管理機能を発展させたもので、並列データベース管理システムの研究開発という目的と、PIM / PIMOS の環境で効率的なデータベース管理機能を多くの知識情報処理システムに提供するという目的を持つ。

このシステムは, データモデルとして非正規関係モデルを採用している. 機能的には、拡張関係代数を中心にした言語処理系、および並列処理のための分散データベース機能が拡充されている。処理面としては、KL1 に適したプロセス指向の設計、および言語の各階層に対応した各層での並列処理が特徴である。

前回は、データベースとしての基本要素の試作評価結果について報告した。今回は、より上層部である中間言語での並列処理方式、ユーザ記述言語の中間言語への変換について述べる。

2 並列システムとしての特徴

- データベース管理システムとしての構成

  Kappa-P は、疎結合を考慮した並列処理と密結合を考慮した並列処理の組み合わせになっている.

  疎結合を考慮した並列処理とは、独立したデータベース管理システム (ローカル DBMS) が複数集まって一つのデータベース管理システムを構成し、問い合わせはそれらのローカル DBMS が協調することでおこなわれる、ということであり、密結合を考慮した並列処理とは、構成要素であるローカル DBMS の内部処理を密結合向きの並列処理でおこなうということである。

  こう切り分けた理由としては、データベース管理システムには、次のような性質があり、一つのデータベース管理システムの内部処理を、疎結合向きの並列処理でおこなう方向のアプローチは、困難と予想されたからである。

  - 操作対象のデータが二次記憶上に存在し、しかも大量である.
    複数の問い合わせ処理によるデータの一貫性を保証する必要がある.
  - 管理するデータの構造が複雑で、しかも大量である (特に主記憶データベース)

- データの配置

  非正規関係のローカルDBMSへの配置は、データベース生成時にユーザが決めることにした．巨大なデータや複数のローカルDBMSに分けることにより高い並列性が得られるようなデータのために、複数のテーブルを仮想的に一つに見せるような機構をもつ水平分割テーブルなどもサポートする．

- 問い合わせの分割

  問い合わせは、利用するテーブルの所在とそれに対する操作、分割した時に発生する通信量やローカルDBMSの負荷を考慮して分割される．

- 疎結合向き並列処理

  疎結合向き並列処理では、複数のローカルDBMSが協調することで、問い合わせ処理を行うことである。そのために次のような処理を行う。

  - 分散トランザクション

    複数のローカルDBMSによる分散処理により、複数の対象(主に二次記憶)に更新が発生するため、分散トランザクションをサポートする(プロトコルは二相コミット)．

  - ローカルDBMS間の通信

    ローカルDBMS間の通信では、非正規関係の複雑な構造のデータを流す必要がある．KL1ではデータの存在場所を意識することなくデータをアクセスすることができるが、複雑な構造のデータを大量にアクセスすることには不向きである．そのため、ローカルDBMS間の通信データを、バッファリングし一括転送向きにコード化しデータの内容ごと転送することにした．転送コマンドは、問い合わせの分割(変換)時に、問い合わせに埋め込まれる．問い合わせを解析することによりデータのうち不要な部分がわかるので、それを除くためのコマンドも同時に埋め込まれる。

- 密結合向き並列処理

  ローカルDBMSの内部では、データの所在を考慮しない処理が中心であり、一つのローカルDBMSは、主に一つのクラスタ内でのみ実行されると仮定している。

  - レコードストリームによる並列処理

    問い合わせを、演算をノード、その間の依存関係をアークとしたグラフとしてみて、演算ノードをプロセス、アークを演算対象であるレコードを流すストリームとすることで、並列処理をおこなう。中間結果を少なくするために、レコードは要求駆動でストリームに流される。

  - 基本的な演算の並列処理

    集合演算、インデックス操作などで共有メモリーを仮定した演算を行う。

- 大域情報の管理

  非正規関係の名前はそれが格納されているローカルDBMSとはまったく無関係とすることにしたため，非正規関係の名前を次のように管理することにした．

  非正規関係の名前とそれが格納されているローカルDBMSの対応が必要になるのは、問い合わせ変換時の初期だけで、重い処理ではないので、集中管理することにした。すなわち、サーバDBMSと呼ぶDBMSでシステム全体の非正規関係の名前を保持し名前の一意性検査や名前とそれが存在するローカルDBMSの対応づけをおこなう．たとえこの情報が壊れたとしても，ローカルDBMSが局所的に管理しているメタデータを集めることにより復元可能であるが，ただ一つのサーバDBMSが存在するだけだと可用性が高いとはいえない．ある程度可用性を高める目的で，複数のサーバDBMSにより大域名前情報の複製をおこなう。複製には、Votingに基づく手法を利用する。

3 問い合せ処理

3.1 言語の階層

次の三階層の言語により問い合わせを処理する．

- ユーザ記述言語

  基本的には、非正規関係のために拡張した拡張関係代数からなる式の集まりである．演繹データベースで必要な推移閉包を効率的に処理するためにループも記述できるようにしてある．

- 中間言語

  拡張関係代数の処理のための中間言語であり，拡張関係代数の処理をさらに細分化したものである．

- 原始コマンド

  PSI上のDBMS Kappa-IIのPrimitive Commandに対応するもので、一つのテーブルに対するレコードポインターに基づく処理が中心である．Kappa-IIのコマンドとの違いは、非正規関係の操作は機能的にサブセットであること，レコードストリームを導入したこと，分散トランザクションをサポートしていることなどである．

ユーザ記述言語で書かれた問い合わせは、一つのトランザクション単位でまとめて受けとられる．そして、最適化，中間言語への変換，ローカルDBMS間データ転送コマンドの埋め込み，演算順序制御コマンドの埋め込み，分割などをおこない，最終的に中間言語処理プロセスに中間言語コマンドをメッセージとして送るKL1プログラムに変換され，該当するローカルDBMSに送られる．KL1プログラムを受けとったローカルDBMSは，それに対応するトランザクションの下で実行する．すると，中間言語

処理プロセスに対し中間言語コマンドが実行すべき順序で送られてくる．中間言語処理プロセスが中間言語コマンドを処理する過程で，原始コマンドが出される．

3.2 問い合せ処理

問い合せ処理は，大きく分けると次の二つに分けられる。

- 問い合わせ変換

  ユーザ記述言語で書かれた問い合わせを，中間言語処理プロセスに中間言語コマンドをメッセージとして送るKL1プログラムに変換する部分で，インタフェースプロセスがこれをおこなう．変換の際に，テーブルの位置情報を取り出すためにサーバDBMSをアクセスする．そして、テーブルが存在するローカルDBMSからテーブルのスキーマと格納データに対する補助情報を取り出し，その情報を基に，最適化，中間言語への変換，ローカルDBMS間データ転送コマンドの埋め込み，演算順序制御コマンドの埋め込み，分割などがおこなわれる．

- 問い合わせ実行

  インタフェースプロセスが変換した中間言語処理プロセスに中間言語コマンドをメッセージとして送るKL1プログラムを受けとり、それを分散トランザクションをサポートするトランザクション プロセスの管理下で実行する．中間言語の実行では，レコード ストリームによる並列処理がなされる．テーブルの生成 / 削除時にはサーバDBMSをアクセスする．他ローカルDBMS上のテーブルへのアクセスは，変換時に埋め込まれた中間言語の転送コマンドによってなされる．すべての問い合わせ処理が終了すると分散トランザクションのプロトコル(二相コミットを採用)に従ってトランザクションを終了させるが、このときに他ローカルDBMSのトランザクション プロセスと通信を行う．

3.3 問い合わせ変換

インタフェース プロセスが，ユーザ記述言語で書かれた問い合わせを，中間言語処理プロセスに中間言語コマンドをメッセージとして送るKL1プログラムに変換する．図1が問い合わせ変換でおこなうことである。

1. サーバDBMSをアクセスしテーブルの位置情報を取り出す
2. テーブルが存在するローカルDBMSからテーブルのスキーマと格納データに対する補助情報を取り出す。この補助情報には、次のようなものがある．

   - 属性に対する索引の有無

     索引の有無で，演算に使用できるアルゴリズムが決まる．

   - 属性値の唯一性

     値がユニークであるかどうかで、演算時の属性の選択に利用．

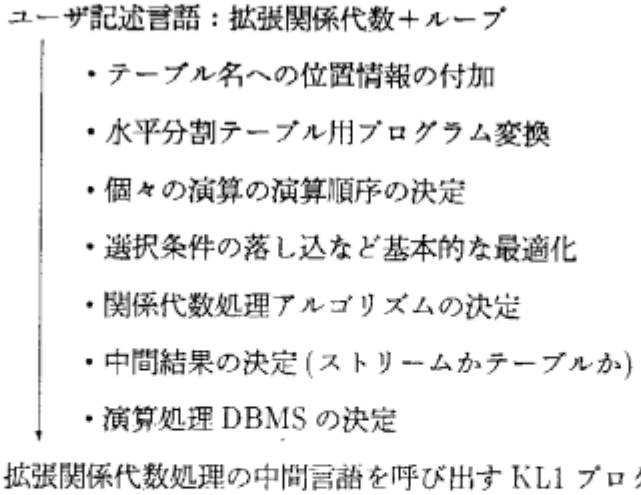


図 1: 問い合せの変換

- 属性値の種類, 属性値の数
  演算結果の量の推定に利用.
- レコードの数とその大きさ
  演算結果の量の推定に利用.

3. 水平分割テーブルをテーブルの実体に置き換える.

4. 個々の演算の演算順序を決定し、演算間に半順序関係をつける. 更新ではない演算は順序を付けずに動かすことができるが、更新操作が入ってくると、更新前/後で結果が異なるので、更新操作はそれより前に終わるべき全ての演算が終了したら開始でき、更新操作後に開始すべき演算は更新処理が終了したら開始できる. ここで、決定された順序は, 演算順序制御コマンドとして埋め込まれる.

5. テーブルの情報を基に, 選択条件の落し込などの基本的な最適化をおこない, 中間結果をレコード ストリームにするかテーブルにするか決定し, 索引の有無などから関係代数処理アルゴリズム (マージ法かハッシュ法かなど) を決め中間言語コマンドへの変換する. ここでは, 複数の候補が得られる。

6. 上で求まった複数の候補に対し、テーブルの位置情報を基に分割 (演算を実行するローカル DBMS を決める) する。そして、データ転送量を予測し最も小さいものを選択し, 通信路にデータ転送コマンドの埋め込む。
   最終的には、演算の重さを考慮し大雑把な時間軸を入れ処理時間の最小値をとるようにしたい.

7. 演算順序制御コマンドの埋め込みなどをおこない KL1 プログラムとしての体裁を整える.

図 2はユーザ記述の例で，図 3, 4はそれを変換したKL1 プログラムである．記述が繁雑になるので，一つのローカルDBMSで実行される場合のみに限定し，中間言語への変換はせず拡張関係代数のまま載せてある。実際には、これにさらに他ローカルDBMSへの通信路が引数として引き回される．

```
go(Result::result1, Temp::result2) :-
      selection(table2, ''(from = "icot"), Temp),
      difference(table1, table1, EmptyTable),
      transitive_closure(table1, EmptyTable, table1, Result),
      replace(table2, ''(a = a + 1 where a > 10)).

transitive_closure(Delta, In, R, Out) :- empty(Delta) |
      In = Out.

transitive_closure(Delta, In, R, Out) :- true |
      join(In, In, '(to = from), In1),
      projection(In1, {'1.from', '2.to'}, In2::(from, to)),
      union(In2, R, NextIn),
      difference(NextIn, In, Delta),
      transitive_closure(Delta, NextIn, R, Out).
```

図 2: 問い合わせ (ユーザ記述)

```
go(ILP, {Table2, Table1, Param_Table_for_tc}, Status) :-
      true |
      ILP = {ILP1, ILP2, ILP3},
      create(ILC1, UDC),
      Table2 = {Table2_1, Table2_2},
      Table1 = {Table1_1, Table1_2, Table1_3, Table1_4},
      ILC1 = [selection(Table2_1, ''('1' = "icot"), Temp, Status1),
              difference(Table1_1, Table1_2, EmptyTable, Status2)],
      UDC = [transitive_closure(Table1_3, EmptyTable, Table1_4, Result,
                                Param_Table_for_tc, Status3),
      ilp:wait([Status1], ILC2,
              [replace(Table2_2,''('2' = '2' + 1 where '2' > 10), Status4)]),
      ILP3 = [modify_schema(Result, result1, Status5),
              modify_schema(Temp, result2, Status6)],
      ilp:wait([Status5], Result, []),
      ilp:wait([Status6], Temp, []),
      lib:error_check([Status1,Status2,Status3,Status4,Status5,Status6],
                      go, Status).
```

図 3: 変換後のプログラム

4 まとめ

並列データベース管理システム Kappa P の並列システムとしての特徴と問い合わせ処理の並列処理について述べた．

```
create(ILP, UDCin1) :- true |
      merge({UDCin1,UDCin2}, UDC),
      top_loop(UDC, UDCin2, ILP).

top_loop([transitive_closure(Delta,In,R,Out,Param_Table_for_tc,Status)
          | UDC], UDCin, ILP) :-
      true |
      ILP = {ILP1, ILP2},
      UDCin = {UDCin1, UDCin2},
      transitive_closure(Delta,In,R,Out,Param_Table_for_tc,ILP1,UDCin1,Status),
      top_loop(UDC, UDCin2, ILP2).

top_loop([], UDCin, ILP) :- true | UDCin = [], ILP = [].

transitive_closure(Delta, In, R, Out, Param_Table_for_tc,
                   ILP, UDC, Status) :- true |
      ILP = {ILP1, ILP2},
      Delta = {Delta1, Delta2},
      ILP1 = [get_cardinality(Delta1, Cardinarity, Status1)],
      transitive_closure_01(Cardinarity, Delta2, In, R, Out, Param_Table_for_tc,
                            ILP2, UDC, Status2),
      lib:error_check([Status1,Status2], transitive_closure, Status).

transitive_closure_01(0, Delta, In, R, Out, Param_Table_for_tc,
                   ILP, UDC, Status) :- true |

      In = Out,
      Delta = [],
      R = [],
      ILP = [],
      UDC = [],
      Status = normal.

otherwise.
transitive_closure_01(_, Delta, In, R, Out, {Iformat1},
                   ILP, UDC, Status) :- true |
      ILP = {ILP1, ILP2, ILP3, ILP4, ILP5},
      In = {In_1, In_2, In_3},
      R = {R_1, R_2},
      ILP1 = [join(In_1, In_2, '('2' = '1'), In1, Status1),
              projection(In1, Iformat1, In2, Status2),
              union(In2, R_1, NextIn, Status3),
              difference(NextIn, In_3, Delta, Status4)],
      UDC = [transitive_closure(Delta, NextIn, R_2, Out, {Iformat1}, Status5)],
      lib:error_check([Status1, Status2, Status3, Status4, Status5],
                      transitive_closure_0102, Status).
```

図 4: 変換後のプログラム (続き)

メタプログラミングのための KL1 ユーティリティ

越村 三幸* 藤田 博† 長谷川 隆三
(財) 新世代コンピュータ技術開発機構

概要

KL1 においてメタプログラミングをする際に有用となる、メタプログラミングのための KL1 ユーティリティについて報告する．ユーティリティは，ユニフィケーションやパターンマッチといった，メタプログラミングに必要な機能を一通り備えている．また，十分な高速性も追求している．

1 はじめに

我々は KL1 での高速定理証明器の実現を目指している．定理証明器では，どのような表現形式を選択するのかとか，どのような戦略を選択するのか，といったことがその効率に大きく影響する．しかしいかなる選択をしたとしても，ユニフィケーションとかパターンマッチは避けられない処理となる．効率性の観点から，これらの処理は重要な処理となる．そこでまず我々は，ユニフィケーションプログラムを KL1 で書いてみることにした．

KL1 においてメタプログラミングする際の問題点については，[11] に詳しい．これを要約すると，「Prolog でのメタプログラミング技法は使えない」ということになる．Prolog でのメタプログラミングにとって，var 述語の役割は大きいが，KL1 でこれに対応する述語 unbound を健全に利用することはできないからである．これにより，オブジェクトレベルの変数をメタレベルの論理変数で表現し，var を利用しつつメタレベルとオブジェクトレベルを巧妙に識別する Prolog 的メタプログラミング技法は KL1 では不可能となる．

むしろ KL1 でのメタプログラミングにとっては，Lisp でのメタプログラミング技法が有益であることがわかる．これはオブジェクトレベルの変数をメタレベルの基底項で表現する方法である．この方法では，変数管理プログラムをメタプログラムとして記述[1]する必要がある．またこのことは必然的に，KL1 言語処理系の提供する論理変数の管理機能を，オブジェクトレベルの変数の管理機能として直接利用できないことを意味し，処理速度の低速化を招くことになる．

一般にメタプログムでは，インタープリテーションオーバヘッドによる実行速度の低速化が問題となる．KL1 におけるメタプログラムでは Prolog におけるメタプログラムよりこのオーバヘッドが大きいことは，上記のことを考慮すると容易に想像がつく．当初我々は，このオーバヘッドは数十倍から百倍程度位と予想していたが，実際には 2 ～ 3 倍であることが分かった．

我々は，オブジェクトレベルの変数の表現を，論理変数表現 / 基底項表現 / 折衷表現の三種類を考え，それぞれの表現法でユニフィケーションプログラムを記述し評価した [9, 11]．本論文ではこのう

*現在，東芝情報システム (株) 所属
†現在，三菱電機 (株) 中央研究所 所属

[1] これは Lisp の eval の a-list の操作に相当するものである．

ち最高速であった基底項表現について述べ，ユニフィーケーションプログラムの効率的な実現手法を提案する．また，この実現手法をユーティリティとしてパッケージ化したメタライブラリについて報告する．最後にメタライブラリを利用した Or- 並列 And- 逐次 Prolog インタプリタの実現例を示し，ライブラリの並列性について考察する．

2 変数の基底項表現

オブジェクトレベルの論理変数をメタレベルの基底項で表現するこの方法では，変数管理機構を記述言語でプログラミングする必要がある．これは Lisp のような記号処理言語でのメタプログラミングで通常用いられる方法である．メタプログラミングの正道とでも言うべき方法である．

変数管理の方法を二通り考えた．一つは変数管理表 (環境) をプロセス表現するもので，もう一つはベクタ表現するものである．どちらもオブジェクトの表現と環境の対によってオブジェクトを表現するもので Lisp の eval で用いられている方法に近い[2]．

2.1 プロセス方式

これは GHC によるユニフィケーションプログラム [1] を KL1 用に書き換えたものである．この方法が最も KL1 らしいプログラムといえようか．Lisp の a-list をプロセスで表現したものと思えば良い．各変数について一つずつその変数管理プロセスを生成し，それらを直列にストリームでつなげるものである．ストリーム並列性がありユニフィケーションのもつ並列性が最も良く反映されるプログラムである．

2.2 ベクタ方式

オブジェクトレベルの変数の値を環境 (ベクタメモリ) に保持する方法である．オブジェクトレベルの変数は値の保持されている場所へのポインタで表現する[3]．この表現法の欠点は，環境を持ち回って変数管理をするために制御が逐次的になることである．

KL1 ではベクタというデータ型があるが，これは要素に直接アクセスできるという点がリストに対して著しく有利である．この利点を活かし，ベクタ上に変数セルをとり，変数はそのベクタインデックスとして表現する．これにより，変数の unboundness は変数セルに書かれるタグで判定できるし，2 つの変数の同一性はベクタインデックスの同一性で簡単に判定できる．

変数セルは図 1のようにベクタ上にとられる．変数セルには 3 種類が入る．

この表現法でプログラムの制御を代えて 2 通りのプログラムを書いた．

1. 逐次制御．構造体の同士のユニフィケーションをする場合，その要素毎の unifier をフォークするが，環境をその unifier 間でバケツリレーするので結局制御は逐次になってしまうプログラム．
2. 徹底逐次制御．上記逐次制御を徹底したプログラム．現 KL1 処理系のスケジューラ [4] を考慮し，実行が中断しない限りスケジュールキューにはゴールが入らないようにした．

[2] 例えば，以下の表現と環境の対は同じオブジェクト $f(a)$ を表している．

- 表現 $= f(a)$, 環境 $= \{\}$
- 表現 $= f(X)$, 環境 $= \{X \leftarrow a\}$
- 表現 $= Y$, 環境 $= \{Y \leftarrow f(X), X \leftarrow a\}$

[3] ユニフィケーションに失敗した場合は単に環境を捨てるだけで良い．

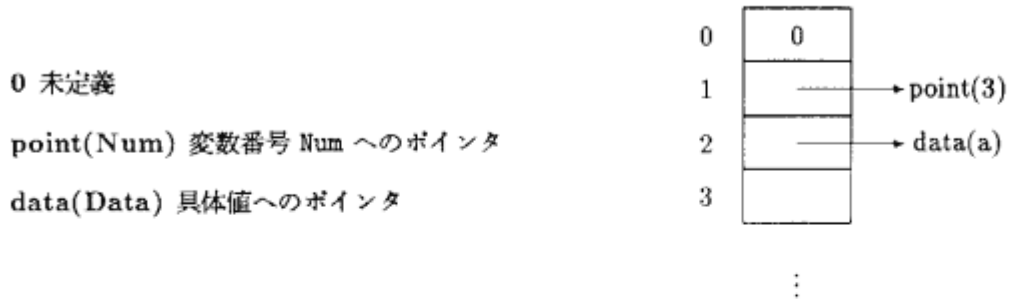


図 1: 変数環境

表 1: 1 ユニフィーケーション時間 (単位: マイクロ秒)

問題	1	2	3	4	5	6	7	8	9	10
KL1 組込み	154	2300	446	589	686	2.43	160	332	124	288
基底項表現 - プロセス方式 -	3020	70600	25400	25500	12400	379	6500	6790	8110	8250
基底項表現 - 逐次制御 -	306	4180	878	1080	1510	34.2	620	844	487	697
基底項表現　徹底逐次制御	261	3510	831	915	1170	37.4	603	873	516	737

3　各方式の評価

前節で述べた各方式について PSI-II 上の擬似マルチ PSI (マルチ PSI のシュミレータ) で測定を行った．その評価結果を示し (表 1.2) それについての考察を述べる．なお測定に用いた例題は次の通りである (A と B のユニフィケーション)．

1 A = i(i(X1,X2),X3), B = i(i(i(Y1,Y2),Y3),i(i(Y3,Y1),i(Y4,Y1)))

2 32 個の異なった変数を持つ binary tree

3 A = p(h(X1,X1),h(X2,X2),Y2,Y3,X3), B = p(X2,X3,h(Y1,Y1),h(Y2,Y2),Y3)

4 A = i(i(X,X),i(i(Y,Y),i(V,i(W,Z)))), B = i(Y,i(Z,i(i(U,U),i(i(V,V),W))))

5 A = (X0,X1,X2,X3,X4,X5,X6,f(X0),f(X6)), B = (X1,X2,X3,X4,X5,X6,a,f(X6),f(X0))

6 A = X, B = f(a)

7 A = [X,x,Y,y,Z,z,U,u,V,v,W,w], B = [x,X,y,Y,z,Z,u,U,v,V,w,W]

8 A = f(X,x,Y,y,Z,z,U,u,V,v,W,w), B = f(x,X,y,Y,z,Z,u,U,v,V,w,W)

9 A = [X0,X1,X2,X3,X4,X5,X6,X7,X8,X9], B = [0,1,2,3,4,5,6,7,8,9]

10 A = f(X0,X1,X2,X3,X4,X5,X6,X7,X8,X9), B = f(0,1,2,3,4,5,6,7,8,9)

3.1　変数管理 - プロセス方式 / ベクタ方式

プロセス方式による変数管理は並列処理言語らしいプログラミングができユニフィケーション問題の持つ並列性を自然に引き出すことができる．しかしプロセス方式はデーブル方式に比べ 1 桁以

表 2: 1 ユニフィーケーション時間比 (KL1 組込み = 1)

問題	1	2	3	4	5	6	7	8	9	10
KL1 組込み	1	1	1	1	1	1	1	1	1	1
基底項表現 - プロセス方式 -	19.6	30.7	57.0	43.3	18.1	156	40.6	20.5	65.4	28.6
基底項表現 - 逐次制御 -	1.98	1.82	1.97	1.83	2.20	14.1	3.88	2.54	3.93	2.42
基底項表現 - 徹底逐次制御 -	1.69	1.52	1.86	1.55	1.71	15.4	3.77	2.63	4.16	2.56

上[4]低速である. 標準的なユニフィケーションの並列度が 2 ~ 3 である [1] ことを考えるとプロセッサの台数効果を考慮に入れてもベクタ方式の方が勝っているといえるだろう.

なぜプロセス方式はベクタ方式に比べこんなにも低速なのであろうか? プロセス方式では変数の個数分の変数管理プロセスができる. 変数管理が必要になるとこの変数管理プロセスが起動される. 一方ベクタ方式では変数表をユニファイヤ自身が保持するためこのようなプロセスはできない. 変数管理プロセス自体の処理は軽いがこのプロセスは起動されるとすぐに休眠状態となる. このように, プロセス方式では軽いプロセスの切替えが頻繁に起こる.

現在の KL1 言語処理方式は頻繁にプロセス切替えが起こるようなプログラムに対しては有利な方式とはいえない. しかしプロセス切替えが頻繁に起こることを仮定した処理系 [8] ならばプロセス方式も一考の余地はある.

3.2 徹底逐次制御

今回作成したプログラムで最高速なものは基底項表現 徹底逐次制御 である. これは完全逐次プログラムである. つまり, サスペンドしない限りどのゴールもスケジューリングキューに enqueue されない. リストの場合の処理について - 逐次制御 - と比べることによりその差異を考えてみる.

逐次制御 .

```
unify([XH|XT],[YH|YT], T,NT) :- unify(XH,YH, T,T1), unify(XT,YT, T1,NT).
```

徹底逐次制御 .

```
unify([XH|XT],[YH|YT], Cont, Env,NEnv) :- unify(XH,YH, [{XT,YT}|Cont], Env,NEnv).
```

現在の KL1 処理方式ではボディ部にユーザゴールが複数ある場合は最初の一つを除きスケジューリングキューに入れられる. そして最初の一つは引き続き実行される[5].

- 逐次制御 - の場合リストの Tail の処理は スケジューリングキューに enqueue される. 一方 - 徹底逐次制御 - では Tail の処理は自前のスタック (Cont) に積み, Head 部分の処理を行うどのゴールも enqueue されない. その代わりユニファイされる構造体の葉に到達した時に Cont からユニファイする対象を取り出してユニフィケーションを継続するプログラムを起動しなければならない. ユニファイする構造体の葉の数を L とするとこのプログラム は L 回起動される. 一方 - 逐次制御 - では L-1 回 ゴールが enqueue される.

- 徹底逐次制御 - は - 逐次制御 - に比べ大体 10 ~ 20 % 高速である. これは完全逐次にした結果, データが常にレジスタ上にあるからである[6]. リダクション数は - 逐次制御 - に比べ L 回増えるのにも

[4] 基底項表現 - プロセス方式 - と - 逐次制御 - の比較

[5] goto で jump するものと思ってよい

[6] PIM ではこうはならない [13]. したがって,PIM では 徹底逐次制御 は遅くなるであろう.

拘わらず高速なのはそれにも増して enqueue が重いということである[7].

このプログラム自体には台数効果は全く期待できない. しかし, 台数効果が最も期待できる変数プロセス管理方式はこれより 1 桁以上低速であることをふまえると徹底逐次制御プログラムの方に軍配を挙げざるをえない.

つまり, ユニフィケーションは現 KL1 処理系では粒度が小さ過ぎる問題ということになる. ユニフィケーションはメタプログラミングでは肝となる処理ではあるが負荷分散という観点からは, もう少し粒度の大きい仕事を分散させるのが有効であることが分かる.

4 メタライブラリ

オブジェクトレベルの変数を基底項表現する場合の変数管理プログラムの記述は大変煩わしい. そこで我々はこれらのプログラムをライブラリ化した [10]. これは基底項表現 (徹底逐次制御) に基づいたプログラム群である.

4.1 機能

このライブラリが提供する機能は, ユニフィケーション (変数出現チェックあり / なし) やパターンマッチ, 環境操作やオブジェクトの入出力関連の機能などである.

1. ユニフィケーション関係
   - `meta#unify(X,Y, Env,^NEnv)` : ユニフィケーション
   - `meta#unify_oc(X,Y, Env,^NEnv)` : 出現チェック付ユニフィケーション
   - `meta#oneway_unify(Pattern,Target, Env,^NEnv)` : 一方向ユニフィケーション
   - `meta#match(Pattern,Target, Env,^NEnv)` : マッヂ
2. 環境操作関係
   - `meta#copy_term(X,^NX, Env,^NEnv)` : 環境 Env の蹴集めを X を根として行なう.
   - `meta#shallow(X, Env,^NEnv)` : X 内の変数のチェインを shallowing する.
3. オブジェクトデータベース
   - `meta#database(S)` : S はデータベースプロセスへのストリーム.
4. 入出力関係 : KL1 の Wrapped された項とメタライブラリで扱うオブジェクトレベルの項を相互に変換する.

4.2 プログラム例 - Prolog インタプリタ -

メタライブラリを使うことにより, 変数管理に煩わせられることなくメタプログラミングすることができる. 例えば, このライブラリを利用した OR- 並列 Prolog インタプリタ (図 2) では, メタプログラムは変数環境を持ち回るだけで, 変数管理の細かいところまで気を使う必要はない.

このインタプリタはライブラリの機能の内, ユニフィケーション機能とオブジェクト検索機能を使っている. ユニフィケーション機能は `meta#unify(X,Y, Env,NEnv)` を呼ぶことにより利用できる. ここで, X と Y はオブジェクトの表現で Env がユニフィケーション前の環境, NEnv がユニフィケーショ

[7] enqueue_goal + proceed は execute より重いということ.

```
solveAnd([],[], Env, Sol) :- Sol = [Env].
solveAnd([],[Gs|Gss], Env, Sol) :- solveAnd(Gs,Gss, Env, Sol).
solveAnd([G|Gs],Gss, Env, Sol) :- clauses(G,Env, Clauses), solveOr(Clauses,G,Gs,Gss, Sol).

solveOr([{C,Env}|Cs], G,Gs,Gss, Sol) :- C = (H :- B) | Sol = {Sol1,Sol2},
    meta#unify(H,G, Env,Env1), expand(Env1, B,Gs,Gss, Sol1), solveOr(Cs, G,Gs,Gss, Sol2).
solveOr([], _,_,_, Sol) :- Sol = [].

expand(fail, _,_,_, Sol) :- Sol = [].
otherwise.
expand(Env, B,Gs,Gss, Sol) :- solveAnd(B,[Gs|Gss], Env, Sol).

clauses(G,Env, CLs) :- vector(G, Size), vector_element(G,0,F) |
    Arity := Size-1, clauses1({F,Arity},Env, CLs).
clauses(G,Env, CLs) :- atom(G) | clauses1({G,0},Env, CLs).

clauses1(FA,Env, CLs) :- clauses1(FA,0,Env, CLs).

clauses1(FA,M,Env, CLs) :-
    '$PrologDataBase':get({FA,M},Env, ExpEnv), clauses1Decide(ExpEnv, FA,M,Env, CLs).

clauses1Decide({}, _,_,_,CLs) :- CLs = [].
clauses1Decide(ExpEnv, FA,M,Env, CLs) :- ExpEnv = {_,_} | CLs = [ExpEnv|CLs1],
    M1 := M+1, clauses1(FA,M1,Env, CLs1).
```

図 2: OR- 並列 Prolog インタプリタ

ン後の環境である．またオブジェクト (Prolog 節) 検索機能は '$PrologDataBase' : get/3 を呼ぶことにより利用できる．これにより述語名とアリティよりその述語定義している節を検索することができる．

インタプリタの核部分は solveAnd/4 と solveOr/5 の二つである．Prolog 節の検索は clauses/3 が行う．

solveAnd/4 ゴールの列 Goals とゴール列のスタック GoalsStack とその環境 Environment を受け取り解 Solution を計算する．ゴール列を先頭から順に解いていく (第二, 三節)．ゴール列が空になったら解が見つかったことになるのでその時の環境を返す (第一節)．

solveOr/5 ゴール (G) と節のヘッド (H) をユニファイ (meta#unify(H,G, Env,Env1)) し，成功すればその時の環境でボディ部を解く (expand/3 第二節)．

clauses/3 ゴール G の候補節のリストを CLs に返す．

5 考察

オブジェクトレベルの変数を基底項表現した場合のユニフィケーションプログラムを幾つか KL1 で記述しその性能比較を行った．それらの内最も高速だったプログラムは徹底逐次制御プログラムであり，これは Lisp でユニフィケーションプログラムを記述した場合のプログラムに非常に近い．

オブジェクトをその表現と環境の対で表現するこの手法は OR- 並列制御にはむいているが AND- 並列制御にはむかない．それはメタプログラム自身が環境を持ち回らなければならないことからきている．この手法を利用したメタプログラムの典型は，例えば，述語 meta1 と meta2 が操作するオブジェクトが AND 関係にある場合，

```
meta1(.., Env,Env1), meta2(.., Env1,Env2), ...
```

という感じになる．この場合，meta1 が何らかの処理をすることにより，環境が Env から Env1 になり，その環境下で meta2 の操作で環境が Env2 となるように処理が進む．meta2 は Env1 が決定するまで，つまり meta1 の処理がほぼ終了するまで処理を開始することができない．実質的には meta1 と meta2 の処理は逐次になってしまう．

プログラマの気持ちとしては

```
meta1(.., Env,Env1), meta2(.., Env,Env2), ...
```

と書きたいところだが，このように書くとあるオブジェクトに対し meta1 の処理をすると環境が Env から Env1 になり一方，meta2 の処理をすると環境が Env から Env2 になる，という意味になる．つまり meta1 と meta2 の操作するオブジェクトの関係は OR 関係になってしまう．この性質を利用したのが OR- 並列 Prolog インタプリタ (図 2) である．

AND 並列を引き出すには，環境管理プロセスを生成し環境の参照・更新はそのプロセスへのメッセージによることにすればよい．しかし，メッセージ通信による計算は並列言語らしいプログラムは書けるが，現 KL1 処理系では効率を損ねる．またこの方法で OR- 並列性を引き出す場合には環境管理プロセスのコピー が必要となるので，効率的な実現は困難である．

このようなことを考えると，オブジェクトの表現 - 環境方式で AND- 並列を効率良く引き出すには KL1 の機能拡張が必要であると思われる．その拡張機能は環境 (KL1 のベクタ型) に対する full test unification[8] のようなものである．

KL1 は並列言語と論理型言語双方の性質を備えてはいる．しかし論理型言語の特徴である論理変数は，並列動作するプロセス間の同期制御の実現には大きな役割を果たしているが，Prolog のように論理変数をオブジェクトレベルの変数にも用いることは一般的にはできない．これはユニフィケーションの成功 / 失敗を条件に条件分岐するプログラムが書けないからである．

このように考えると

$$KL1 = Prolog - \text{test unification} + \text{同期機構 (one-way unification)}$$

とみなすことができ，またそうみなした方が効率の良いプログラムが書ける．

一方，Shapiro は Flat Concurrent Prolog (FCP) による簡潔な Or-Parallel Prolog インタプリタを示している [5]．ここではオブジェクトレベルの変数には論理変数が用いられている．このインタプリタはガード部の full test unification を利用して Prolog のユニフィケーションを実現している．Full test-unfication により並列論理型言語と Prolog の融合が可能になっているわけである．つまり FCP は，

$$FCP = Prolog + \text{同期機構 (read-only annotation)}$$

となっており，KL1 とは対照的である．したがって，FCP は KL1 と比べるとメタプログラミングしやすいのは当然といえるだろ．しかしその代償として，FCP の (分散環境化での) 実現は KL1 と比べると格段に難しい．FCP のガード部での full test unification は論理変数に対する長時間のロック機構[9]が必要となるからである．

[8] ユニフィケーションの成功 / 失敗を条件分岐の条件に利用できること

[9] 構造体同士がユニファイ可能かどうか確かめてから，実際にその構造体内の変数に値を代入するまでのロック

このことは FCP は KL1 に比べてその言語処理系自体が重くなることを意味する．上の 2 式より大雑把にいって

$$FCP = KL1 + \text{test unification}$$

がいえるが，変数管理プログラムを KL1 で記述することは，FCP の full test unification の部分を含む機能を KL1 で記述することに相当する．変数管理プログラムを記述すれば test unification は容易に実現できるから，我々の

$$\text{メタプログラム} = KL1 + \text{変数管理プログラム}$$

によるアプローチは FCP によるメタプログラミングのアプローチに比べ，より広範囲の問題を扱える．また，両アプローチのどちらが最終的に効率の良いプログラムが書けるかは，今後の経験より明らかとなるだろう．

もちろん，静的に情報の流れが十分に分かっているような対象やそのような対象に問題が還元できる場合，KL1 は記述言語として非常に適した並列言語といえるだろう [7, 12, 2, 3]．そのような問題領域として，データベース，自然言語処理，自動プログラミング等がある．このような KL1 の言語機能を有効に生かせる問題領域を開拓していくのも今後の研究課題である．

参考文献

[1] Fujita, H.: "Parallel Unification and Meta-Interpreters in GHC", ICOT TR-468, 1989.

[2] Fujita, H., Hasegawa, R.: "A Model Generation Theorem Prover Using A Ramified-Stack Algorithm" ICOT TR-606, 1990.

[3] Hasegawa, R., Fujita, H., Fujita, M.: "A Parallel Theorem Prover in KL1 and Its Application to Program Synthesis", ICOT TR-588, 1990.

[4] Nakajima, K., Inamura, Y., Ichiyoshi, N., Rokusawa, K., Chikayama, T.: "Distributed Implementation of KL1 on the Multi-PSI/V2", In *Proc. of 6th ICLP*, 1989.

[5] Shapiro, E.Y.: "Or-Parallel Prolog in Flat Concurrent Prolog", In *Concurrent Prolog: Collected Papers* Vol.2, 1987.

[6] Ueda, K.: "Making Exhaustive Search Programs Deterministic", In *Proc. of 3rd ICLP*, LNCS 225, Springer-Verlag, 1986.

[7] Ueda, K., Chikayama, T.: "Design of the Kernel Language for the Parallel Inference Machine" , In the Computer J., Dec. 1990.

[8] Ueda, K., Morita, M.: "A New Implementation Technique for Flat GHC", In *Proc. of 7th ICLP*, 1990.

[9] 越村 三幸, 藤田 博, 長谷川 隆三: "KL1 上のユニフィケーションプログラムとその評価", ICOT TM-975, 1990.

[10] 越村 三幸, 藤田 博, 長谷川 隆三: "メタライブラリ・マニュアル", ICOT TM-976, 1990.

[11] 越村 三幸, 藤田 博, 長谷川 隆三: "KL1 におけるメタプログラミング", 記号処理 57-2,1990

[12] 竹内 彰一, 高橋 和子, 清水 広之: "並列問題解決用言語 ANDOR-II", ICOT TR-235, 1987.

[13] 平野喜芳, 後藤厚宏: "並列論理型言語 KL1 のコンパイル方式の改良", JSPP'90, 1990.

並列自然言語解析における並列協調の効果について

山崎重一郎

富士通研究所

1991年5月

概要

我々は自然言語解析のための並列処理モデルとして，形態素解析，構文解析，意味解析，文脈処理などの自然言語解析処理の全てのレベルを並列協調させる処理モデルを提案している．

この並列協調モデルが，考え方として自然であるだけでなく，実際の並列処理において効率的でもあるということを実証するために，形態素解析と構文解析の並列協調を行なうシステムをマルチPSI上に実現し，実験，評価を行なった．実験に用いた辞書の規模は，形態素解析辞書が約１００００語，単語辞書が約２０００語であり，文法規則はICOTの佐野氏が開発した約８００規則のもの[佐野91]を用いた．

実験の結果，構文解析のみの処理では３２台のプロセッサを使用したときの台数効果が６．８倍であったのに対して，形態素解析と構文解析を並列協調させたときの台数効果は９．０倍であり，並列協調により台数効果が向上することが確認できた．

また処理時間は，１台のプロセッサで実行したとき，構文解析のみに比べ形態素解析と構文解析の両方を行なったときの処理は１．５倍の時間がかかっていたが，プロセッサを３２台使うと両者はほぼ同じ時間になり，形態素解析のための時間はほとんど無視できるようになった．

１．はじめに

我々は第５世代コンピュータプロジェクトの一環として，並列推論マシンのための自然言語解析システムの研究開発を行なっている．研究の目的は，並列処理の観点から見て自然でかつ高速な並列自然言語解析の処理方法を開発することである．

我々が提案している並列処理モデルは，自然言語解析処理の全ての処理レベル，すなわち形態素解析，構文解析，意味解析，文脈処理を並列的に協調させるものである[山崎90]．

この並列協調モデルの並列処理効率に対する効果を評価するために，マルチPSI上に形態素解析と構文解析の並列協調を行なうシステムを試作し実験評価を行なった．実験に用いた辞書の規模は，形態素解析辞書が約１００００語，単語辞書が約２０００語であり，文法規則はICOTの佐野氏が開発した約８００規則のものを用いた．本稿では，この実験の結果を報告する．

２．並列協調モデル

形態素解析，構文解析，意味解析，文脈処理などの自然言語の解析の各処理はそれぞれ莫大な曖昧性を生成するが，他の処理レベルの情報を利用することによってかなりの曖昧性を解消することができる．例えば，形態素解析処理は文字の列から形態素を見つける処理であるが，ここで発生する莫大な候補は構文的な適切さを検査すれば大部分を刈り取ることができる．

しかし，一般的な自然言語解析システムでは形態素解析を行なった結果を使って構文解析を行なっているので，形態素解析処理の段階では構文解析から得られる制約条件を利用することができない．しかし，並列処理の観点から見ると，形態素解析と構文解析は段階的に行なうのではなく，部分的に実行した結果を相互に情報交換させながら並列的に実行させればよいので両者を協調させるのが自然である．

形態素解析と構文解析の関係と同様に構文解析と意味解析の関係も相互に協調すべきであり，また文脈情報の利用も処理のレベルを越えて利用可能であることが望ましい．

我々の並列自然言語解析の処理モデルは自然言語解析の全ての処理レベルの間で相互に情報交換を行ない並列協調させるというものである．

この並列協調が考え方として自然であるだけでなく，実際の並列処理において効率的な処理方法でもあることを検証することが，本実験の目的である．

3．並列協調の実現方法

我々の並列自然言語解析システムでは，形態素解析と構文解析との違いはデーターの違いに過ぎず，どちらの処理も同一の並列解析機構を用いて実行している．これによって，割り込み制御のような複雑な制御を一切持ち込むことなく，並列協調が実現できている．

ただし，文字と形態素と品詞はＫＬ１のデータタイプとして区別しており，処理の効率化に利用している．しかし，こういったデータータイプを意識した処理はプログラム中では，10クローズ程度の追加に過ぎず，基本的には全ての処理を等質的に扱っている．因みに，並列解析機構のプログラムは全体で１０００行を越えている．

このデータータイプの違いを利用した効率化は，例えば，構文構造を決めるためには，直前の単語だけではだめで，区切り位置の前方全体を検索対象にしなければならないが，文字列から形態素を検索するには直前の文字だけを参照すればよいという探索範囲の節約や，構文解析を行なうときに単一化による情報伝播を行なっているが，品詞どうしでは双方向的に情報を伝播させる必要があるが，一方が形態素の場合は情報伝播は一方向だけでよいので単一化処理をさぼることができる，というような制約伝播処理の節約などに利用している．

3．1 形態素辞書の構造

我々のシステムでは，文字列から形態素を認定する処理は辞書に存在する全ての形態素について，その右端の文字をキーとし，その左に隣接する文字を順にまとめていった木構造（TRIE構造）を形態素解析辞書として持っている．

我々の並列処理系は，このTRIE構造を論理的にはOR並列的に探索するが，実際には同一のプロセッサで処理している．TRIE構造の探索は多段ハッシュで行なってもよいが，現在はシーケンシャルに行なっている．１万語程度の辞書でも，ほとんどが６個以下ずつしか分岐していないので，現在のところはシーケンシャルで充分であると思われる．

TRIE辞書の具体例
（アトムは文字，ストリングは形態素を意味している）

```
word(険,T) :- true| T=[
    (危 < ["危険"]),
    (保 < ["保険",
            (害 < [(損 < ["損害保険"])]),
            (災 < [(火 < ["火災保険"])])])].
```

上の記述で”＜”という記号はこの記号の左側の要素が実際に左に接続したときには接続した結果が”＜”の右側の要素になることを意味している．

3．2単語辞書の構造

単語辞書では，同一の形態素でも意味や読みの違うものを別の単語としてOR並列的に扱う．なお，単語辞書の素性構造は基本的にICOTの佐野氏が開発した文法の枠組みに基づいている．また，意味的な範疇についてはEDRの概念体系を参考にした．

単語辞書の具体例

（1引数のファンクター形式は品詞を表し，{R,x}という形式の2要素のベクターは我々の並列解析機構における変数を意味している）

```
type(アクセス,R,T) :- true| T=[
  体言_0([語彙素性=[語基=[語彙=[表記=工場,読み=アクセス],範疇区分=体言語基]],
        意味素性={R,sem}<<思考内容の権利([])]),
  体言_0([語彙素性=[語基=[語彙=[表記=工場,読み=アクセス],範疇区分=体言語基]],
        意味素性={R,sem}<<方手段([])])].
```

この例は，同一の形態素であるが，意味的に異なる単語を別単語として扱っている．
また，通常は文法規則として扱うような情報も，我々のシステムでは辞書に入れている．

```
type(んだり,R,T) :- true| T=[
  (用言_4([R,r4]=[語彙素性=[系統={R,ch}<<[ウ系ナ,ウ系バ,ウ系マ]]]) < [
      連用詞_4([R,r4]=[構文素性＝[連用句関係＝and]])])].
```

これは「踏んだり蹴ったり」や「とんだりはねたり」の”んだり”の辞書記述である．
この記述も，TRIE辞書と同様に，”＜”の左側の要素が左から結合すると，結合した結果が”＜”の右側の要素になるという意味である．
この例では，”んだり”の左側に用言_4という品詞として判断された要素が結合すると，連用詞_4になることを意味している．

3．3 形態素の接続の検査

一般的な形態素解析処理では，文字列が形態素として認定されると，次に隣接する形態素どうしの接続可能性の検査を行ない，候補の絞り込みを行なう．

我々のシステムではこのような検査は構文解析の一部として行なっている．

単語辞書の例

```
type(んだり,R,T) :- true| T=[
  (用言_4([R,r4]=[語彙素性=[系統={R,ch}<<[ウ系ナ,ウ系バ,ウ系マ]]]) < [
      連用詞_4([R,r4]=[構文素性＝[連用句関係＝and]])])].
```

上の例では，左側に接続可能な要素は用言_4という品詞であるばかりでなく，系統という素性の値が型付きの変数であり，この変数への具体化が型による制約を充足されなければならない．

この型は集合表現で，ウ系ナ,ウ系バ,ウ系マという3つの要素からなる有限集合である．したがって，”

んだり”の左側にくる用言は系統素性の値がこの３つのいずれかでなければならない．

例えば，とぶ（とばない）の”と”の系統素性の値はウ系バなので，”んだり”の左側に接続することができる．

この，型制約には集合表現の他に，類概念を記述することもできる．

系統＝{R,ch}<<ウ系統([])

と書くと，系統の値として許容されるものは，ウ系アル，ウ系コ，．．．．ウ系ジルなど２５の要素を列挙したものに等しい．

因みに，今後予定している意味解析も，この型付き変数への具体化の時の型検査に基づく制約伝播によって実現するつもりである．

３．４ 文法の構造

文法に記述されるのは，純粋に品詞どうしの関係である．このような規則では品詞が持つ素性情報が単一化により双方向的に伝播する必要があるので，エントリー自体が”＜”の左側に来る構造になっている．

文法の記述例
(”*”は素性構造の直積を表す記号であり，素性構造の木から特定の部分木を切り取る作用を持つ)

```
upper(用言_2,R,T) :- true| T=[
    (用言_2([R,y2]={R,h}*(構文素性｜下位範疇化｜が格=[構文関係=subj])) < [
        (連用詞_2([構文素性=[表層格=が]]) < [
            用言_2([R,h]=[構文素性=[有標=スクランブル,
                                    下位範疇化=[が格=[構文関係=[]]]]])])])].
```

この規則は，この用言_2が，が格を主語として下位範疇化しており，左から表層格がが格の連用詞が適用されると，適用された結果できる用言_2はもうが格を下位範疇化しないことを意味している．

４．実験

自然言語解析技術として実質的に意味を持つためには，辞書，文法ともに，ある程度の規模のもので実験を行なわなければ意味がない．このために，約１万語の形態素辞書と約２０００語の単語辞書と約８００規則の構文解析文法を並列協調解析用に開発した．実験に用いた文法はICOT第６研究室の佐野氏が開発した本格的な日本語処理文法である．

４．１実験条件

実験に用いたハードウェアは３２ＰＥ構成のマルチPSI．PIMOSは2.6版である．

実験に用いた例文は以下のものであり，文字数が２４文字，形態素数が１４であり構文解析による解の個数は２００である．

例文：「その製品を買ったので生産の拡大の計画をあきらめた」

4．2 解析時間

実験の結果，構文解析のみの処理では３２台のプロセッサを使用したときの台数効果が６．８倍であったのに対して，形態素解析と構文解析を並列協調させたときの台数効果は３２プロセッサで９．０倍であり，並列協調により台数効果を高めることができた．

また処理時間は，１台のプロセッサで実行したとき，構文解析のみに比べ形態素解析と構文解析の両方を行なったときの処理は１．５倍の時間がかかっていたが，プロセッサを３２台使うと両者はほぼ同じ時間になり，形態素解析のための時間はほとんど無視できるようになった．

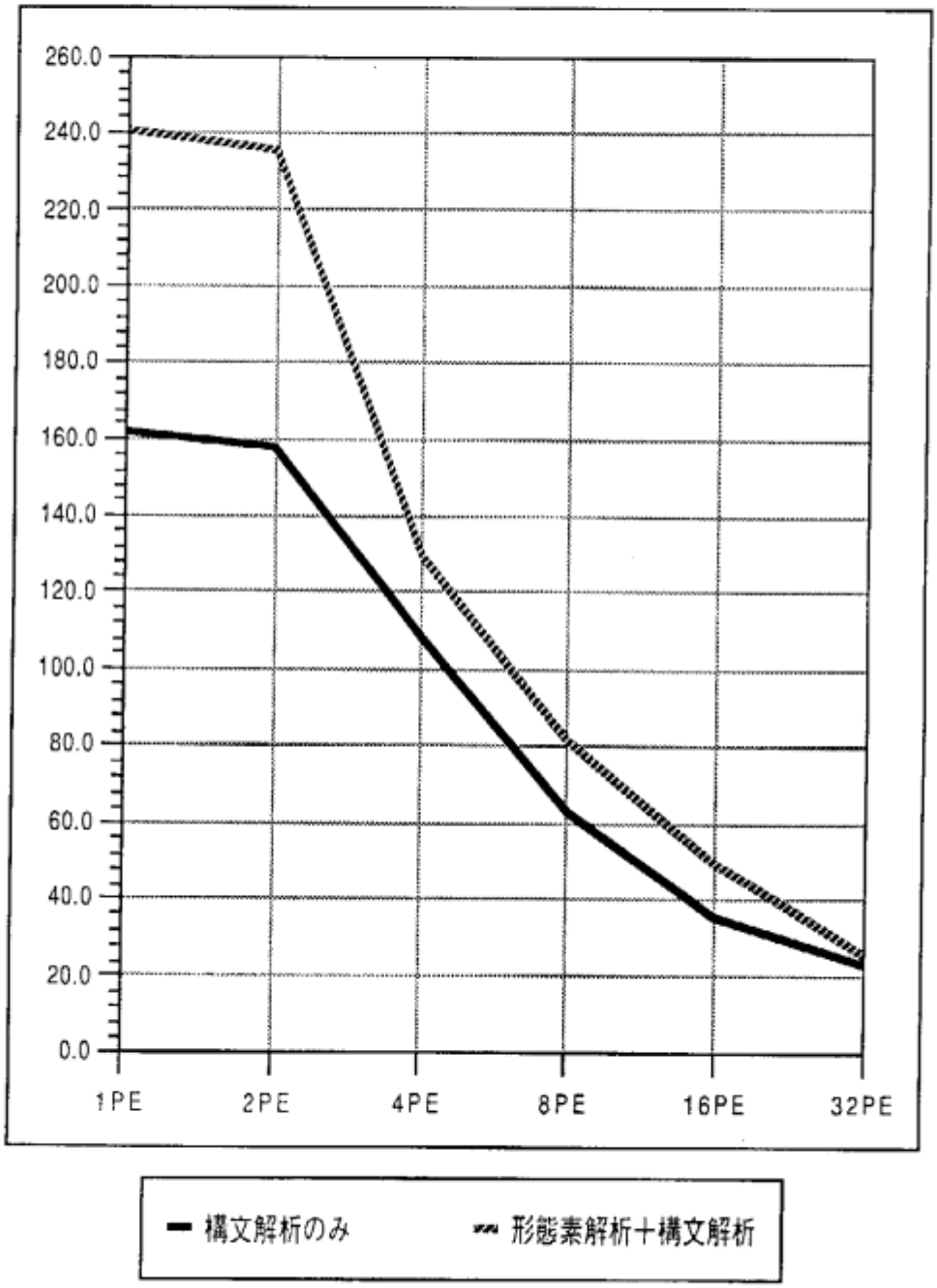


表１．プロセッサ台数と解析時間

4．今後の課題

今回の実験では，構文的曖昧性の爆発に悩まされた．これは一つの形態素が２０近い単語としての曖昧性をもつものがあり，構文解析全体の曖昧性がこの語彙的曖昧性の積になってくるからである．

今後さらに意味処理との並列協調の実験を行なう予定であるが，意味処理によって，語彙的曖昧性を中心とした構文的曖昧さを減少させ，構文解析の処理量の爆発をおさえることができれば，大規模な自然言語処理でも，並列協調の効果はさらに明確に現われるのではないかと予想している．

また，現在のシステムでは優先度制御は全く行なっておらず，単純な全解探索を行なっているが，システムの最終的な姿としては，良い解を速く出すようにしたいと考えている．このために，優先度制御と負荷分散を組み合わせた処理モデルを提案しており，これの実験を行なっていきたい．

謝辞

本研究はＩＣＯＴからの再依託研究として行なわれた．実験を進めるにあたって田中裕一室長をはじめとするＩＣＯＴ第６研究室の皆様にご協力ををいただいた，特に佐野氏には文法を提供していただいた．ここに記して感謝いたします．また，富士通研究所の長沢氏，末広氏の協力で今回の実験は実現できました．さらに，大規模化実験のための各種ツール群を作成して下さったFIPの小野寺氏，弘田氏に感謝いたします．

参考文献

[佐野91]佐野洋，福本文代：文の様相区分と文型射影への試み，「自然言語処理における統合」シンポジウム．1991.

[寿崎89]寿崎かすみ他：マルチPSIにおける並列構文解析プログラムPAXの実現および評価，並列処理シンポジウムJSPP'89,PP.343-350．1989.

[佐藤 90]佐藤裕幸．並列自然言語解析システムPAXの改良．KL1プログラミングワークショップ'90予稿集．1990.

[松本90]松本裕治．並列論理型言語における探索問題のプログラミング-Layered Stream法の拡張．KL1プログラミングワークショップ'90予稿集．1990.

[山崎他90] 山崎重一郎．並列自然言語解析システムLaPutaについて．KL1プログラミングワークショップ'90予稿集．1990.

[山崎他91] 山崎重一郎，末広香緒里，長澤郁子，杉山健司．並列自然言語解析システムLaPutaの試作評価．情報処理学会第42回全国大会講演論文集

並列一般化LRパーザの性能評価

沼崎 浩明† 池田 朋男†† 田中 穂積†
Hiroaki NUMAZAKI Tomoo IKEDA Hozumi TANAKA

†東京工業大学 工学部
Tokyo Institute of Technology
〒152 目黒区大岡山 2-12-1 TEL 03-3726-1111 (内線 4175)
††株式会社 東芝

概要

本論文では，KL1で記述した並列一般化LRパーザについて示し，その性能を並列推論マシンMulti-PSIを用いた実験に基づいて評価する．Multi-PSIは疎結合型の並列計算機であるため，負荷分散の方式の善し悪しが，パーザの性能に大きく影響する．これまでの実験により，並列パーザの負荷分散方式を考える上で，次の二点を考慮することが，性能向上の鍵となることが明らかとなっている．(1) 負荷分散を行なうプロセスが，パーズプロセスの実行を妨げないこと．(2) 仕事を終えたプロセッサを再利用する動的負荷分散方式を採用すること．(1)は一般性のある原則であり，(2)は探索空間の幅がパーザと同様に途中で増減するような問題に適用できる原則と考えて良い．ただし，(2)はプロセッサの数に余裕がある場合はこの限りではない．動的負荷分散方式は，一般に稼働していないプロセッサを検出するプロセス(以下プロセッサマネージャと呼ぶ)を作り，負荷分散を行うプロセスが，空いているプロセッサをマネージャに問い合わせることによって実現できる．この方式は，プロセッサの割り付けを予め決めておく静的負荷分散と比較して通信量が増大し，また，プロセッサマネージャの処理が全体のボトルネックとなる危険性がある点に注意が必要である．本研究ではこの点を考慮して，全てのプロセッサを消費するまでは静的負荷分散を行い，プロセッサを消費し尽くした後に動的負荷分散に移行するという融合方式を採用している．規則数180の純粋な文脈自由文法を用いて，数万個の解析木を有する文に対して得られた解析時間の台数効果(プロセッサ1台での処理時間に対して64台を用いた場合の処理速度の向上比)は，12倍という良好な値を得た．また，本研究で提案する融合方式の有用性を，静的負荷分散のみ，あるいは，動的負荷分散のみの方式と比較することにより実証している．

1 はじめに

我々は，並列論理型言語KL1を用いて並列パーザを構築し，その性能を評価している．このパーザは自然言語の並列処理を実現するために開発され，文の構文的な曖昧性を並列に解析する．そのアルゴリズムは，効率の良いことで知られているLR法に基づいている．LR法は与えられた文法から定まる全ての解析状態をLR状態遷移図として表し，これに基づく状態推移により構文解析を進める．すなわち，LRパーザは初期状態から，一語一語の入力語に応じて状態間を推移し，入力の終了時に最終状態に到達した時，文を受理する．また，その状態推移の過程で，解析木を構築する．本来，LR法は文脈自由文法のサブセットであるLR文法に対して，決定的な構文解析を行うアルゴリズムとして開発されたものであるが，これを一般の文脈自由文法に適用したアルゴリズムを一般化LR法と呼ぶ．一般化LR法の解析は，文脈自由文法の持つ曖昧性により，非決定性が生ずる．すなわち，一つの入力語に対して複数の推移先を持つ状態が生成されるということである．その状態は文の解析が曖昧になる場所を示している．我々のパーザは，そのような状態に到達した場合，可能な全ての解析を並列に実行する．曖昧性の数は入力語を読み進むにつれて指数関数的に増大するが，その数を抑制するための方法として，解析木を構築する際，規則に課せられた制約条件を評価することによって，規則の適用を制限するという方策を採用している．

本研究で提案する並列一般化LRパーザの効率性は，既にシミュレーションにより実証されている[1]．本論文では，これを実際の並列計算機Multi-PSI上で効率良く実行するための負荷分散方式を開発した．

本研究における負荷分散方式の開発の経緯は以下のように要約できる．

- 簡単な静的負荷分散方式を開発し，200程度の曖昧性のある文に対して，2倍程度の台数効果を得

ることを確認した[2].

- 上の方式を改善し，パーズプロセスの実行を妨げない静的負荷分散方式の導入により，2.5倍程度の台数効果を達成した[3].

本研究では，静的負荷分散と動的負荷分散を融合した方式を採用することにより，最大12倍の台数効果を得ている．本論文では，我々の並列一般化LRパーザとその負荷分散方式の特徴を明らかにし，実験により性能を評価する．

本論文の構成は次のとおりである．
第2章では，並列一般化LRパーザの探索問題としての特徴を明らかにする．第3章では，並列パーザの計算モデルを示す．第4章では，本研究で用いた負荷分散の方式を示す．第5章では，インプリメンテーションを示す．第6章では，64台版のMulti-PSIを用いて並列パーサの性能を評価する．また，Prolog上のGHCの処理系を用いて，シミュレーションにより得られる理想値と実際の値を比較することにより，通信コストと負荷の偏りの影響を考察する．第7章では，結論と今後の課題について述べる．

2 並列一般化LRパーザの探索問題としての特徴

文脈自由文法の構文解析は非決定性を有し，これを探索問題とみなした時，次のような特徴を持っている．

2.1 入力文と探索空間の関係

探索空間の深さと幅は入力文の長さとその性質によって異なる．

純粋な文脈自由文法に対する並列一般化LR構文解析では，探索空間の深さが入力文の長さにn比例し，探索空間の幅(プロセス数)が，$O(Cn^2)$となるアルゴリズムが提案されている[4]．しかし，本研究では意味処理との融合を行う枠組を提供するために，解析木を個別に扱う方式を採用しており，$O(C^n)$のプロセスを必要とする．ここで，CはLR状態遷移図の状態数である．

2.2 文法規則数と探索空間の関係

探索空間の深さは，一つの解析木を構築するのに適用した規則数に比例する．最悪ケースとして，一つの解析木が全ての文法規則を適用して得られる場合を考えると，探索空間の深さは文法規則数に比例する．

ある一文の解析について考えた時，探索空間の幅は上に示したようにLR状態遷移図の状態数Cにより決まる．LR状態遷移図の各状態は，その状態に推移する際に適用された規則と，その規則に割り付けられたドットの位置によって決まる[5]．従って，LR状態遷移図の状態数Cは，規則数Nと規則に割り付けることができるドットの位置の最大数(規則右辺のカテゴリの最大数+1)Mの積に比例する(ただし，状態推移の際に複数の規則が同時に適用されるケースが含まれると，その値よりも大きくなる可能性があるが，一般には，そのようなケースが全体に占める割合は少ないので無視できる)．

2.3 探索空間の一般的な形状

複数のプロセスが同一の部分木を構築すること(重複計算)を許した場合，探索空間の幅は一般に解析が進むにつれて指数関数的に増大する．ただし，解析の途中で処理を終えるプロセスが存在する場合があり，探索空間は必ずしも単調増加するとは限らない．同一の状態のプロセスを統合することにより重複計算を回避する場合，探索空間の幅は増減を繰り返しながら増大していく．後者の場合について，解析の途中の曖昧性の数を複数の文について調べた結果を，図1に示す．この図では横軸が探索の深さを示し，縦軸が幅を示している．

2.4 重複計算の回避の問題

逐次的な構文解析では，重複計算は必ず回避すべきであるが，並列処理においては，プロセッサの資源に余裕がある限り，重複計算を許しても文の解析時間は変わらない．プロセッサ間の通信コストが大きい疎結合型の並列計算機を用いた場合は，重複計算を回避するために必要な通信のオーバーヘッドによって，逆に効率が低下する可能性が大きい．

このような観点から，本研究ではパーズプロセスの数がプロセッサの台数よりも少ない間は重複計算を許し，プロセス数がプロセッサ数を上回り，一台のプロセッサで複数の解析木を構築する必要が生じた時に初めて重複計算を回避するという方式を採用している．すなわち，異なるプロセッサ間では，重複計算を許す代わりに通信を避け，そのオーバーヘッドを低減している．

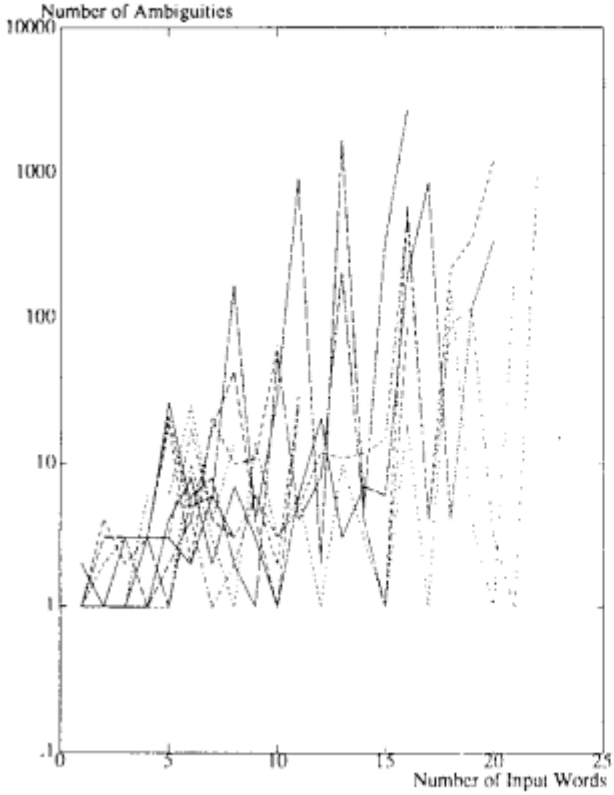


図 1: 探索空間の幅

3 並列一般化 LR パーザの計算モデル

我々の並列一般化LR パーザは以下の特徴を有する.

- 入力文を左から右へと解析する.
- 文の構文的な曖昧性を並列に計算する.
- 重複計算の回避のための同期を避ける新たなデータ構造を採用している[1](ただし, このデータ構造は密結合型のプロセッサを有する並列計算機で特に有効となる).
- 文法に与えた制約により, 漸進的な曖昧性解消を行なうことができ, 意味処理を融合する枠組を提供している.
- 全てのプロセスはデータ駆動により動作する.

この並列パーザの計算モデルを図2に示す. このモデルは, 一般化LR 構文解析の過程を 11 種類のプロセスに区分し, それらをLR パーズ表のエントリに応じて組み合わせ, さらに入力文の並びに従って接続することにより動作する. 各プロセスの間にはスタックを受渡しするストリームを張り, 解析の途中で複数のスタックが生成された場合は, それらを並列に処理する.

図3は, ある文に対するプロセスの接続関係を示している. 本論文では, この詳細な説明は省略するが, "in"の解析で実行がLR パーズ表の競合エントリに到達した時, 複数のスタックが生成され, それ以後, 並列処理が行われていることがわかる.

4 負荷分散方式

本研究では, 既に提案している静的負荷分散方式に, 動的負荷分散方式を融合した新たな負荷分散方式を導入している. 静的方式と動的方式とを融合することの利点は, 双方の長所を生かせることにある. すなわち, 生成されるプロセスの数がプロセッサ数よりも少ない間は, 通信量の少ない静的負荷分散を行い, プロセッサを全て使い尽くした後, さらに負荷分散する必要が生じた場合, 暇になったプロセッサを再利用する動的負荷分散を行う. この融合方式は, 静的方式, および動的方式のどちらか一方のみを採用した場合よりも, 高い台数効果を得ることができることが実験により確認されている.

また, 全ての探索レベルで負荷分散を行なうことも本方式の特徴である. ここでいうレベルとは, 入力語を一語読み進む動作に相当する. このようなマルチレベルの負荷分散を行う理由は, 構文解析における探索空間の幅が単調に増大しないことによる. すなわち, 探索空間の幅が, 解析の途中で増減し, その様子は入力文に応じて変化するため, 有効な負荷分散を行うためのレベルを予め設定することは不可能である. これにより, 負荷分散できる時はいつでも直ちに分散するというきめ細かな方式が必要となる.

以下では, 本研究の融合方式に採用している静的負荷分散, 動的負荷分散, 及び融合方式について説明する.

4.1 静的負荷分散方式

静的負荷分散方式は, 各パーズプロセスに使用可能なプロセッサ番号の集合を割り当てることにより実現している. 各プロセスは, 次のレベルのプロセスを使用可能なプロセッサに割り付けた後, 使用可能なプロセッサが残った場合は, それを均等に分割して, 既に分散した次のレベルのプロセスに与える.

この方式の特徴は, 次のレベルの処理に進めるプロセスは直ちに別のプロセッサに投げるという「先投げ方式」を採用している点にある.「先投げ方式」を採用した理由は, 負荷分散のプロセスがパーズプロセスの実行を妨げることなく行うという原則を実現

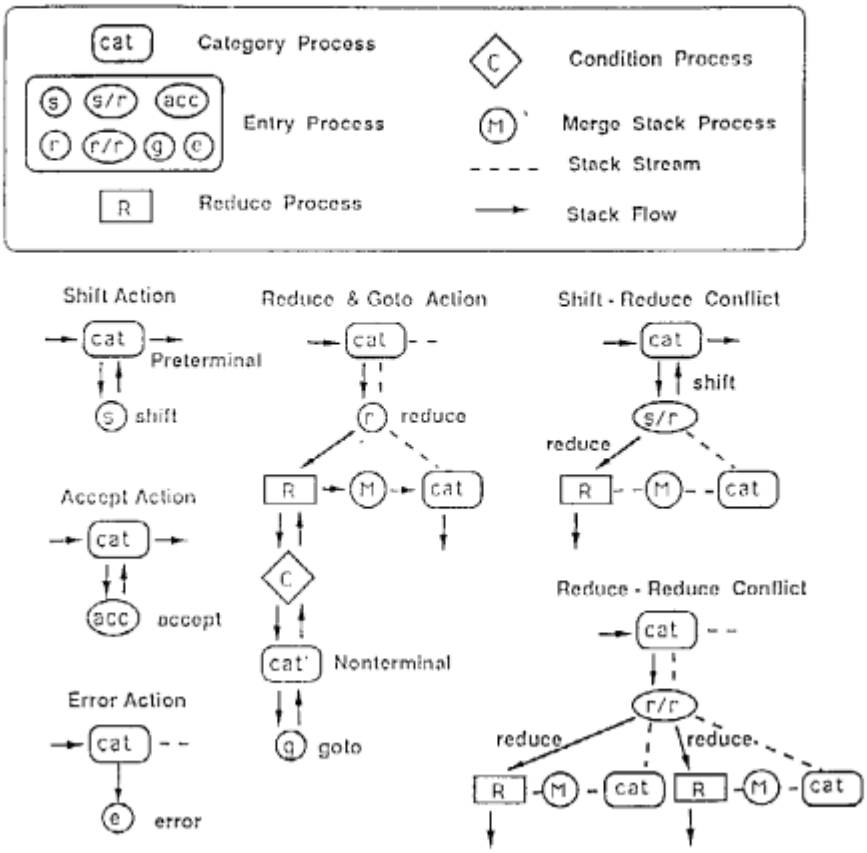


図 2: 並列一般化LR パーザの計算モデル

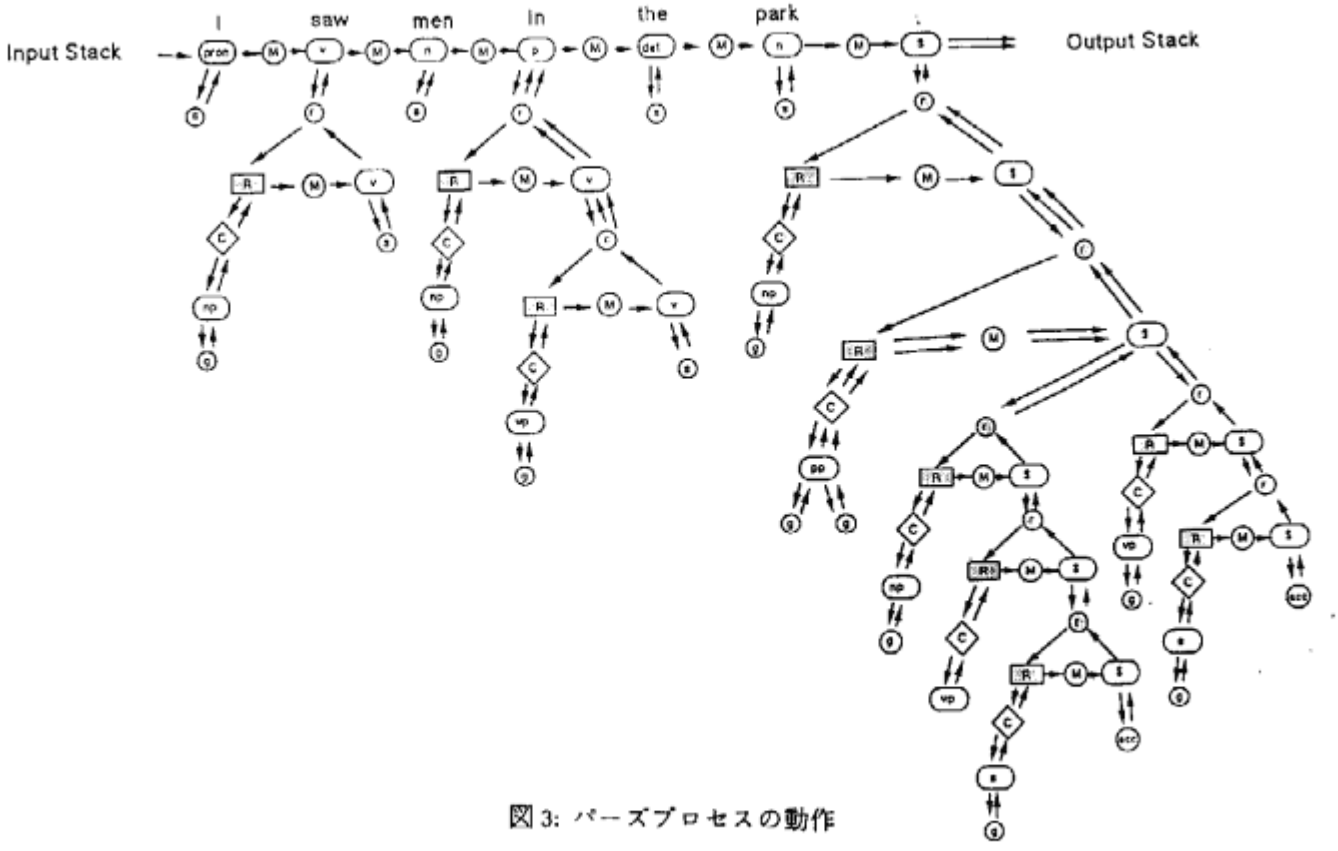


図 3: パーズプロセスの動作

するためである．先投げをを行わず，次のレベルのプロセスが全て出揃ってから負荷分散を行った場合，先に生成されたプロセスは，が無駄な中断を強いられることになり，これが全体の効率を低下させる．

この静的負荷分散方式の実現上の特徴は，プロセッサ番号の集合を{先頭番号，要素数，公差}の三つ組で定めれれる数列として表現し，通信量の低減を図っている点にある．これにより，プロセッサ番号の分割も簡単な数値計算で行うことができる[3]．

4.2 動的負荷分散方式

本研究で導入する動的負荷分散方式の特徴は，実際に処理を行うプロセス自身が，その終了時に，自分の保持していたプロセッサ番号の集合をプロセッサマネージャに渡す点にある．この方式は「1プロセッサ・1プロセス」の原則の上に成り立っている．すなわち，本方式では，一つのプロセッサ上に他のプロセッサから投げられたプロセスが複数存在することはない．

動的負荷分散を実現するプロセッサマネージャとパーザは，次のようなアルゴリズムに従って処理を進める．

- 実行の開始時のプロセッサマネージャが持つ空きプロセッサのリストは[ ]とする．
- 処理を終えたパーズプロセスは，自分が保持していたプロセッサの集合をマネージャに送る．マネージャは，それをリストに蓄える．
- プロセッサを使い尽くしたパーズプロセスが，さらに負荷分散を行おうとする際，マネージャに空いているプロセスが存在するかどうかを問い合わせる．
- この時，マネージャは空きプロセッサのリストが[ ]ならば，パーズプロセスに"none"という情報を返す．これに対し，パーズプロセスは負荷分散をあきらめる．
- リストが空でなければ，そこから一つのプロセッサ番号を取り出し，パーズプロセスに返す．パーズプロセスは，そのプロセッサに次のレベルのプロセスを投げる．

本方式と，古市らの動的負荷分散方式[6]とは，暇なプロセッサが見つからない場合の処理が異なる．本方式では，暇なプロセッサがない場合，プロセッサマネージャは，プロセッサの獲得要求に対して，空きプロセッサがないことを報告するが，古市らの方式では暇なプロセッサが見つかるまで待つ．

本方式を採用した理由は，重複計算の回避の問題と関わっている．すなわち，本方式の前提として，異なるプロセッサ上のプロセスで生ずる重複計算は，通信量の低減のために許すということがあげられる．この前提に従って負荷分散を考えた場合，重複計算を行う複数のプロセスは，一つのプロセスに統合してこれを一台のプロセッサで実行した場合と，統合せずに複数のプロセッサで別々に重複した計算を行った場合とは，どちらも解析時間に差はないが，プロセッサが空くまで待った後で負荷分散し，重複した計算を行った場合，その待ち時間の分だけ処理時間が長くなる．このような理由から，本研究では，「負荷分散できない時は負荷分散することをあきらめる」という方式を採用している．

4.3 融合方式

上に示した通り，本方式は最初に静的負荷分散方式で処理を進め，パーズプロセスが自分に割り当てられたプロセッサを使い尽くした後，さらに負荷分散を行う場合は，プロセッサマネージャに空きプロセッサを問い合るという動的方式に移行する．この方式の利点は，動的負荷分散のみの方式と比較して，通信量が少なく効率が良いことである．

5 インプリメンテーション

以下のプログラムは，一般化LRパーザを，一般の探索問題の一つとしてとらえ，KL1のプログラムとして記述したものである．

```
parse([],In,Out):- ture|
   Out=In.
parse([Word|Sentence],In,Out):- ture|
   parse_one_word(Word,In,Result),
   distribute(Sentence,Result,Out).
```

ここで，プロセス parse は，入力ストリーム In にスタックを受けとり，プロセス parse_one_word を呼んで入力された各スタックに対して一語分の解析を進め，その結果を Result に得た後，負荷分散のプロセス distribute に渡す．このプロセスは，ストリームに入力されたスタックが複数存在する時，各スタックに対して一つずつ parse_one_word を呼び出し，それらを異なるプロセッサに投げることによって負荷分散を実現する．

本研究では，プロセッサマネージャを一つだけ用いた動的負荷分散を行っている．従って，プロセッサマネージャの仕事が全体のボトルネックになる危険性もある．複数のマネージャを用いる方式の開発は今後の課題である．

6 性能評価

6.1 実験

本研究で提案する負荷分散方式の有用性を示すため，規則数180の純粋な文脈自由文法を用い，解析時間とその台数効果を測定した．さらに，Prolog上の処理系を用いて得られた理想値との比率を求めた．

解析時間の測定に用いた文は以下の13文である．

1. There are three on the table now.
2. The structural relations are holding among constituents.
3. He explained the example and he gave the rule.
4. This is a film that is developed in the research.
5. Its procedures allow phrases to inherit attributes from their constituents and sentences to get attributes.
6. This paper presents an explanatory overview of a large and complex grammar in a computer.
7. For every expression it analyzed, diagram provides an annotated description of the structural relations holding among its constituents.
8. This paper presents an explanatory overview of a large and complex grammar in a computer for interpreting english dialogue.
9. The man called 'John' presents an overview of a grammar and sentences, that is used in a computer system.
10. The annotations provide important information for other parts of the system that interpret the expression in the context of a dialogue.
11. This paper presents an explanatory overview of a large and complex grammar that is used in a computer for interpreting english dialogue.
12. This paper presents an explanatory overview of a large and complex grammar, that is used in a computer system , for interpreting dialogues.
13. Its procedures allow phrases to inherit attributes from their constituents and sentences to get attributes from the larger phrases which are constituents of the context.

文	語数	木	時間(sec)	reduction	台数効果
1	7	3	.131	798	1.1 (5.6)
2	7	14	.206	4908	1.9 (18)
3	9	66	.569	24980	3.3 (90)
4	10	26	.550	27138	3.1 (52)
5	15	2737	18.1	1448787	8.9 (981)
6	15	90	.765	48615	5.2 (98)
7	19	356	6.86	805434	8.1 (459)
8	19	1240	7.25	728699	12.3 (435)
9	20	2873	70.0	4207369	7.8
10	21	926	7.93	780492	7.25 (290)
11	21	22541	150	12180235	9.88
12	24	1608	24.8	1768683	10.41
13	25	82294	973	96014714	計測不能

表1: 実験結果

これらの文に対して，単語数，解析木の数，64台のプロセッサを用いた時の解析時間，reduction数，台数効果(理想値)を表1に示す．曖昧性の数が千以上ある文については，台数効果(プロセッサ数1台の場合と比べた時のプロセッサ数64台の時の処理速度の向上率)として，7～12倍程度の比較的良い値を得ている．

この表で，最後の文の台数効果が計測不能となっていることの理由は，1台のプロセッサを用いた場合に，Multi-PSIのメモリが不足して解析できなかったことによる．また，理想値が計測できていないものは，ワークステーションのメモリ不足により，その文が解析できなかったことによる．

使用するプロセッサ数に対する台数効果の推移を明らかにするために，文8と文12に対して，プロセッサ数と台数効果の関係をグラフにしたものを図5，図6に示す．このグラフはどちらもプロセッサ数が増加するにつれ，台数効果も向上する傾向を示しているが，台数効果はプロセッサ数に正比例せず，台数が増加するにつれて，正比例した場合の直線から離れている．これは，プロセスの粒度が小さくなる一方で，負荷分散のオーバヘッドが大きくなるためであると思われる．

また，各グラフが一様な増加を示していないのは，静的負荷分散の方式に起因すると思われる．すなわ

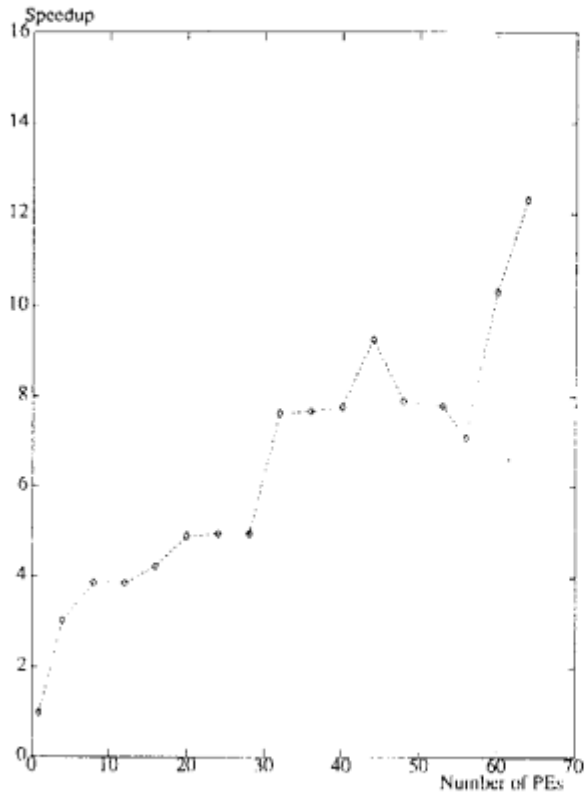


図 4: 文 8 の解析における台数効果

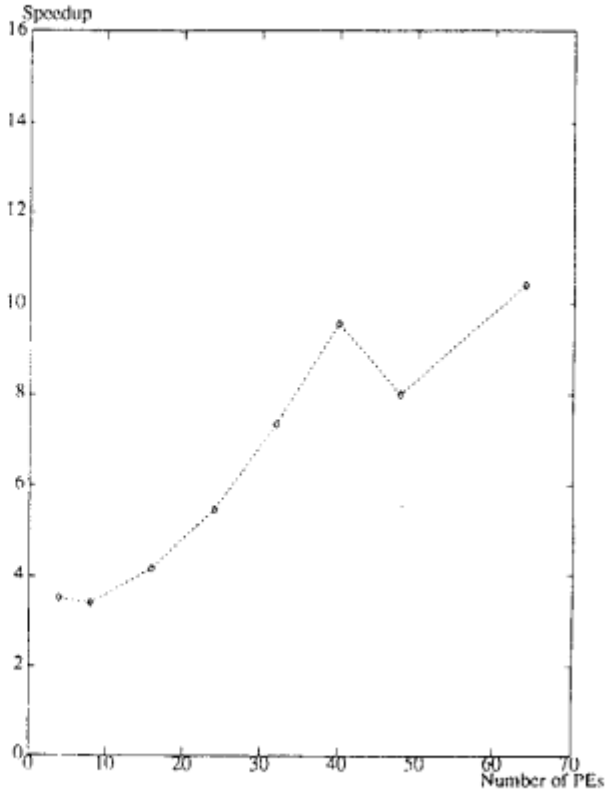


図 5: 文 12 の解析における台数効果

方式	解析時間(sec)	台数効果
静的方式	1.054	3.74
動的方式	0.888	4.44
融合方式	0.765	5.16

表 2: 負荷分散方式の違いによる性能の比較

ち，我々が採用している静的負荷分散方式では，あるプロセスが割り当てられたプロセッサを消費し尽くさない場合，そのプロセスが終了しない限り，それが確保しているプロセッサはたとえ暇であっても，他のプロセスが利用できない．従って，プロセッサの台数が増えたとしても，プロセッサの割り当て方が変わり，運悪くそのようなプロセスに多くのプロセッサを割り当ててしまった場合，台数効果が低下することもあり得る．

6.2 各負荷分散方式の評価

静的負荷分散方式，動的負荷分散方式，融合方式の三つの方式の差異を明らかにするため，64 台のプロセッサ使用時の文 7 に対する解析時間と台数効果の比較を表 2 に示す．この表の各方式を性能の良い順にあげると，融合方式，動的方式，静的方式の順になることが分かる．

6.3 負荷の偏りの分析

PIMOS の ParaGraph というツールを用いて負荷の偏りを調査した．図 7 は，各プロセッサの負荷の総量を示している．この図の横軸がプロセッサ番号，縦軸が負荷の総量である．この図から，各プロセッサの負荷にはっきりとした偏りが生じていることが分かるが，特定のプロセッサに突出した負荷がかかっていないと言う意味では，負荷分散に成功していると考えて良い．

図 8 は，時間で区切って見た場合の各プロセッサの負荷を示している．この図の横軸が時間，縦軸がプロセッサ番号，マスの中の色の濃さが負荷の重さを示している．この図から，処理の後半で負荷に大きな偏りが出ていることが明らかとなった．

7 結論

本研究では，文脈自由文法の並列構文解析を実際の並列計算機で行い，その性能を測定した．負荷分散の方式として，静的方式と動的方式とを融合した方式を採用することにより，64 台のプロセッサを用いて 12 倍という比較的良好な台数効果を得た．

本研究では，一般の探索問題に適用できる形の負荷分散方式を採用しているため，LR パーザ固有の性

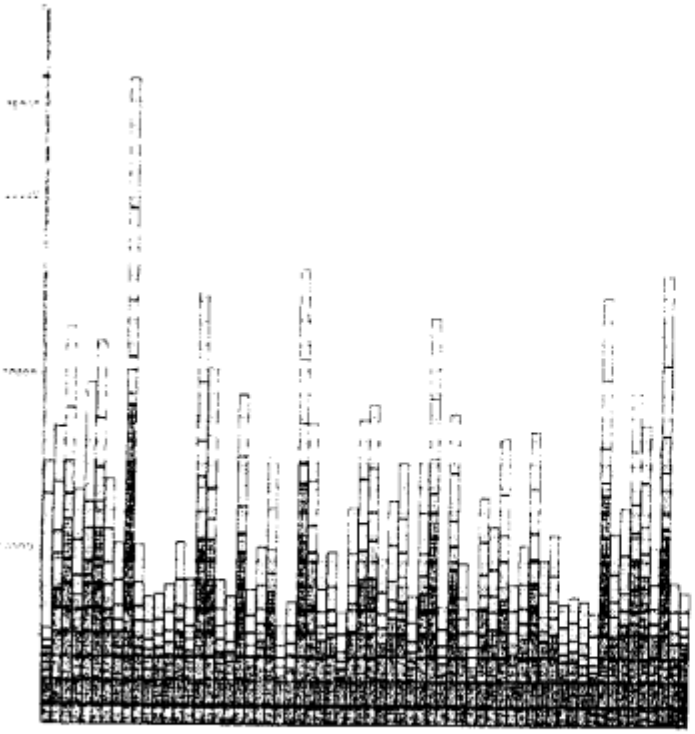


図 6: 負荷の総量の分布

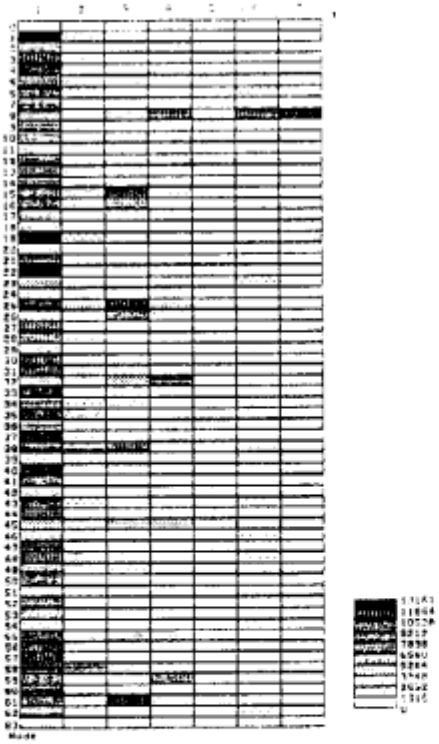


図 7: 負荷の時間的な分布

質を十分に考慮していない．すなわち，パーザが最後の入力語を読み込んだ後に生ずるレデュース動作が，並列実行可能な多くのプロセスを生ずるにもかかわらず，本方式ではこれを負荷分散していない．このため，実行の後半でかなり大きな負荷の偏りを生じていることが，プロセッサの稼働状況の分析により明らかとなった．

今後，この問題をさらに改善することにより，より高い台数効果を得る負荷分散方式を実現できる可能性がある．また，本研究では，純粋な文脈自由文法を扱っていたが，我々の並列パーザは，規則に課せられた制約条件の評価や，意味処理との融合を実現する枠組を有している．今後の研究では，それらを含めたパーザの性能評価を行う必要がある．

謝辞

本研究を進めるにあたり，日頃から御協力を頂いた田中研究室の皆様，ならびに，貴重なご意見を頂きましたICOT のKL1 タスクグループの皆様に感謝致します．

参考文献

[1] 沼崎浩明，田中穂積．論理型言語に基づく効率的な並列一般化LR 構文解析．The Logic Programming Conference, pp. 191–198, 1990 .

[2] 沼崎浩明，田中穂積．並列一般化LR パーザの負荷分散の検討．KL1 Logic Programming Workshop, pp. 123–130, 1990 .

[3] 池田朋男，沼崎浩明，田中穂積．Multi-psi を用いた並列横型探索アルゴリズムの負荷分散に関する一考察．情報処理学会第41回全国大会, pp. 123–124, 1990 .

[4] 峯　恒憲，谷口倫一郎，雨宮真人．文脈自由文法の並列構文解析．情報処理学会 自然言語処理研究会, 73(1):1–8, 1989 .

[5] J.D. Aho, A.V.and Ulman. *The Theory of Parsing, Translation, and Compiling*. Prentice-Hall, Englewood Cliffs, New Jersey, 1972.

[6] 古市　昌一，瀧　和男，市吉伸行．疎結合型並列計算機上での動的負荷分散方式とその評価．KL1 Logic Programming Workshop, pp. 1–9, 1990 .

The Concurrent Constraint Language GDCC and Its Parallel Constraint Solver

David J. Hawley
Institute for New Generation Computer Technology (ICOT)
csnet: hawley%icot.jp@relay.cs.net

Abstract

We describe the current state of development of the concurrent constraint language GDCC (*Guarded Definite Clauses with Constraints*), is a member of the cc (*Concurrent Constraint*) family of languages which supports multiple solvers and recursive queries in a committed-choice framework. GDCC models computation as the interaction of multiple cooperating agents through the exchange of information via querying and asserting the information into a central repository. Concretely, this paradigm is embedded in a guarded (conditional) reduction system, where the guards contain the queries and assertions. Control is achieved by requiring that the queries in a guard are true (entailed), and that the assertions are consistent (satisfiable), with respect to the current state of the repository.

We present and evaluate a parallel implementation of the Buchberger Algorithm, which is the basis of a constraint solver used in GDCC to handle rational polynomials. The Buchberger Algorithm is a basic technology for symbolic algebra, and several attempts at its parallelization have appeared in the recent literature, with some good results for shared memory machines. The algorithm we present is designed for the distributed-memory Multi-PSI, but nevertheless shows consistently good performance and speedups for a number of standard benchmarks from the literature.

1 Introduction

Constraints, that is, formulas describing conditions on objects in some domain, is an interesting and important programming paradigm that has a voluminous literature. In the last five years, the integration of constraint programming with Prolog has received a considerable amount of attention from both the viewpoint of applications and theory based on the theoretical foundation of Jaffar and Lassez[JaL86]. As for extending this work from the sequential to the concurrent frame, there is little published work, among which is a report of some preliminary experiments in integrating constraints into the PEPSys parallel logic system[Hen89]), and a proposal, the *Concurrent Constraint* programming language scheme, for integrating constraint programming with concurrent logic programming languages[Sar89]. The cc programming language paradigm models computation as the interaction of multiple cooperating agents through the exchange of information via querying and asserting the information into a central repository. Concretely, this paradigm is embedded in a guarded (conditional) reduction system, where the guards contain the queries and assertions. Control is achieved by requiring that the queries in a guard are true (entailed), and that the assertions are consistent (satisfiable), with respect to the current state of the repository. An attractive aspect of the cc paradigm is that it allows concurrency in the reduction system, in the constraint-solvers, and also in the interaction between the two.

This paper introduces Guarded Definite Clauses with Constraints (GDCC), an experimental cc language, which supports a user-specified set of sorts and constraint symbols in a committed-choice framework, and is intended to be a research tool for investigating issues of constraint-solving in concurrent programming languages, such as problem decomposition, use of multiple solvers and hybrid techniques, ordering of constraints, management of semi-decidable solution methods, debugging techniques, etc.

The Buchberger Algorithm for solving systems of polynomial equations forms the basis of constraint solvers for several domains in the CAL constraint logic programming system, and we will show how it can be used to build a constraint solver for GDCC. In this application, the input set of polynomials is not given at the start of the computation, but is generated concurrently by some other process, possibly depending on the intermediate sets of basis polynomials. The Buchberger Algorithm is very time/space complex which restricts its use in practice. Recently, there have been several attempts made to parallelize the Buchberger Algorithm with generally disappointing results[Pon90, Sen90], except for shared-memory machines[Vid90, ClL90]. Parallelization has been tackled at two levels: a coarse-grain parallel rewriting of the S polynomials and/or testing for subsumption and critical pairs, and a fine-grain rewriting of single S-polynomials. An interesting concurrent logic programming (data-flow) approach implemented on a transputer-based multicomputer was reported by Siegl [Sie90], with good speedups on the very small examples shown, but poor absolute performance due to the production of rules that are not completely reduced with respect to the current set of rules and to the generation of critical pairs using subsumed rules. We present a new distributed algorithm with good speedups that avoids these problems, and which can be used for both the usual *static* problem, in which the complete set of input polynomials is available at the start of the Gröbner Base calculation, and the *dynamic* problem, in which the input polynomials are sent at arbitrary intervals from some processor(s). Our presentation will focus on the dynamic variant.

The paper is organized as follows. We first informally review some of the vocabulary of the constraint logic and concurrent constraint schemes, and then we introduce the GDCC language. We then introduce the rational polynomial constraint system, discuss its suitability in the cc framework, and present a concurrent constraint solver with experimental results.

2 Constraints in Logic Programming

2.1 Constraint Logic Programming

Constraint logic programming (CLP), proposed by Jaffar and Lassez [JaL86], is an extension of logic programming in which unification is replaced by the solving of equations over some theory, and which can be then further generalized to allow non-equational relations as constraints on variable values. Similarly to Prolog, a CLP program comprises predicates and clauses, where clauses have the (abstract) syntax

Head :- Constraints, Goals.

and is executed depth-first, left-to-right, with constraints taking the place of unification. Operationally, (head) unification serves a dual purpose in logic programming, first to bind actual (goal) and formal (clause) parameters, and secondly to prune the search. Constraints take both these roles. Two major suitability requirements were given for a constraint solver to be used in a constraint logic language:

satisfaction-complete - since pruning of the proof tree is based on the constraints, the constraint solver must be able to determine the satisfiability or unsatisfiability of any constraint with respect to the accumulated constraint set.

incrementality since constraints are added dynamically during execution, the constraint solver must be able to add new constraints to the accumulated constraint set efficiently, and

Jaffar and Lassez also observe that although the mode of invocation does not affect the correctness of constraint evaluation, it does have a large effect on the efficiency of evaluation. For example, in order of increasing cost, the query `?- X=2, Y=1, X+Y=3` can be treated as a test, and `?- X+Y=3, X=1` can be handled by constraint propagation, while `?- X+Y=3, X-Y=1` must be handled as a system of simultaneous equations, by some method such as gaussian elimination. Likewise, the global ordering of constraint evaluation has a large effect on efficiency.

We would like to extend the constraint logic paradigm to a concurrent paradigm, both to take advantage of potential parallelism in the logic component and constraint solver component, and to address problems of controlling the parallelism with particular reference to the order of constraint evaluation. The next section reviews a framework for starting to deal with these issues.

2.2 Concurrent Constraint Programming Languages

Concurrent Constraint programming languages [Sar89], are a generalization to concurrency of the CLP languages. We will present a brief summary of the basic concepts of cc. The logical interpretation of CLP programs is replaced by the notion of cooperating agents, which communicate via queries and assertions into a (consistent) global database of constraints called the *store*. Saraswat remarks that the CLP scheme is too weak, since it lacks control features suitable for concurrent languages. Accordingly, in cc constraints occurring in program text are classified by whether they are querying or asserting information, into Ask and Tell constraints respectively. [1] Asking and Telling are required to be stable operations.

The following definitions are adapted freely from [SaA89, Sar89, Mah87]. We define the following sets: S is a finite set of *sorts*, including the distinguished sort HERBRAND, F a set of *function symbols*, C a set of *constraint symbols*, P a set of *predicate symbols*, and V a set of *variables*. A sort is assigned to each variable and function symbol. A finite sequence of sorts, called a *signature*, is assigned to each function, predicate and constraint symbol. We write $v : s$ if variable v has sort s, $f : s_1 s_2 \ldots s_n \rightarrow s$ if functor f has signature $s_1 s_2 \ldots s_n$ and sort s, and $p : s_1 s_2 \ldots s_n$ if predicate or constraint symbols p has signature $s_1 s_2 \ldots s_n$. We require that terms are well-sorted, according to the standard inductive definitions. An *atomic constraint* is a well-sorted term of the form $c(t_1, t_2, \ldots, t_n)$ where c is a constraint symbol, and a *constraint* is a set of atomic constraints. Let Σ be the many-sorted vocabulary $F \cup C \cup P$. A *constraint system* is a tuple (Σ, Δ, V, C), where Δ is a class of Σ structures.

We define the following meta variables: c ranges over constraints, g,h range over atoms, q ranges over clauses, and p ranges over predicates.

We now define the four relations *answers*, *accepts*, *rejects*, and *suspends*. The constraint c *answers* c_l if

$$\Delta \models (\forall x_g)(c \Rightarrow \exists x_l . c_l)$$

c *accepts* c_l if

$$\Delta \models (\exists)(c \wedge c_l)$$

and c *rejects* c_l if

$$\Delta \models (\forall x_g)(c \rightarrow \neg(\exists x_l . c_l))$$

where x_g are the variables in c, and x_l are the variables in c_l but not in c. Note that the property *answers* is strictly stronger than *accepts*, and that *accepts* and *rejects* are complementary. We say that c *suspends* c_l, if c accepts, but does not answer c_l.

A cc language program is comprised in the usual way of clauses. A clause is defined as a tuple (head,guard,tell,body), where "head" is a term with unique variables as arguments, "guard" is a tuple $< c_a, c_t >$, c_a, c_t and "tell" are a constraints, and "body" is a set of terms. The constraint c_a is said to be *ask-moded*, while c_t and "tell" are said to be *tell-moded*. We

[1] In a language such as Flat Guarded Horn Clauses[Ued86] querying and Ask-constraints correspond to guard unification and guards, while asserting and Tell-constraints correspond roughly to output unification and body unifications respectively.

abuse notation somewhat in the following definitions. Constraint s *confirms* guard $<c_a,c_t>$ if

$$s \textit{ answers } c_a \wedge s \textit{ accepts } c_t$$

constraint s *suspends* $<c_a,c_t>$ if

$$s \textit{ accepts } c_a \wedge s \textit{ accepts } c_t$$

and s *rejects* q if

$$s \textit{ rejects } c_a \vee s \textit{ rejects } c_t$$

Informally, a clause $(h, <\text{a,t}>, c, b)$ is a candidate for goal g in the presence of store s if s, $g{=}h$ *confirms* $<a,t>$ and s, $g{=}h$ *accepts* a. A goal g *commits* to candidate clause $(h, <\text{a,t}>, c, b)$, by updating the store s with $t \cup c$, and replacing g by b. A goal fails if $\forall q.q = (h,a,t,b) \wedge s \wedge g = h \Rightarrow s$ *rejects* $<\text{a,t}>$. Deciding *confirms* for multiple clauses and commitment for multiple goals can be done in parallel.

2.3 Example

The following example is adapted from [AiS88]. Given an arbitrary quadrilateral, we wish to find the ellipse that passes through the midpoints of the four sides, as illustrated in figure 1. The approach is to calculate the midpoints of the four sides first, and then to calculate the ellipse as a linear transformation of a circle. The transformation matrix is calculated based on the relationship between a unit square an the parallelogram formed by the midpoints. Here is a fragment of a GDCC program to solve this problem.

```
:- module ellipse.
:- public start/6.
{create alg Ax,Ay,Bx,By,
            Cx,Cy,Dx,Dy}
start(P1,P2,P3,P4,X,Y) :- true |
  A=p(Ax,Ay), B=p(Bx,By),
  C=p(Cx,Cy), D=p(Dx,Dy),
  mid(P1,P2,A),
  mid(P2,P3,B),
  mid(P3,P4,C),
  mid(P4,P1,D),
  calc_ellipse(A,B,C,D,X,Y).

{input alg X1,Y1,X2,Y2,X3,Y3}
mid(p(X1,Y1),p(X2,Y2),p(X3,Y3)) :-
      true |
  alg:2*X3=X1+X2,
  alg:2*Y3=Y1+Y2.

{input alg X1,Y1,X2,Y2,X3,Y3,X4,Y4}
calc_ellipse(P1,P2,P3,P4,X,Y) :-
  P1=p(X1,Y1), P2=p(X2,Y2),
  P3=p(X3,Y3), P4=p(X4,Y4),
  alg:(X1-X2)*(Y3-Y4)=(Y1-Y2)*(X3-X4),
  alg:(X1-X4)*(Y2-Y3)=(Y1-Y4)*(X2-X3) |
  calc_ellipse_okay(P1,P2,P3,P4,X,Y).
```

The entry-point is `start/6` which takes the coordinates of the quadrilateral's vertices, and the two variables on which the ellipse constraint will be imposed. For example, the query `ellipse:start(p(X1,1),p(1,5),p(X2,8),p(-9,9),X,Y)` will return constraints on `X` and `Y` such that the point (X,Y) must lie in the ellipse drawn through the midpoints of a quadrilateral whose vertices are (X1,1), (1,5), (X2,8) and (-9,9). The program proceeds by spawning four processes to calculate the midpoints of the four sides of the quadrilateral, and a process to calculate the ellipse. The guard in `calc_ellipse/6` *suspends* until, by checking that each of the two pairs of opposite sides have the same slope, it verifies that the midpoints form a parallelogram. Correctness of the method used by `calc_ellipse/6` is ensured by the guard. Additionally, the constraint-solver has a more constrained problem to deal with since the guard guarantees that the midpoint calculations have finished before new constraints are generated from `calc_ellipse/6`. This is typically much more efficient, which shows the value of *control based on information-flow* for concurrent constraint languages.

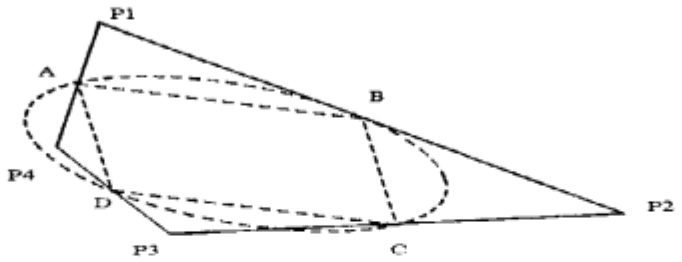


Figure 1: Example: Ellipse through midpoints

3 Solving Rational Polynomial Constraints

In [SaA89, SaS88], Buchberger Algorithm/Gröbner Base constraint solvers for the rational polynomial, boolean and finite-cofinite set domains were shown to fit reasonably well with the Constraint Logic Programming scheme, since they are incremental, and satisfaction complete. We are interested here to investigate the use of Gröbner Base techniques within the concurrent constraint programming language scheme for the rational polynomial domain, which can be formalized as the constraint system $(\Sigma = F \cup C \cup P, \triangle, V, C)$, where:

$$
\begin{aligned}
S &= \{\mathbf{A}\} \\
F &= \{\times : \mathbf{AA} \rightarrow \mathbf{A}, + : \mathbf{AA} \rightarrow \mathbf{A}\} \\
&\quad \cup \{\text{fraction} :\rightarrow \mathbf{A}\} \\
C &= \{=\} \\
P &= \{\text{string starting with a lowercase letter}\} \\
V &= \{\text{string starting with an uppercase letter}\}
\end{aligned}
$$

with the structure

$$
\begin{aligned}
D(\mathbf{A}) &= \text{the set of all algebraic numbers} \\
D(\times) &= \text{multiplication} \\
D(+) &= \text{addition} \\
D(\text{fraction}) &= \text{the rational number it denotes}
\end{aligned}
$$

and

$$\triangle = \text{axioms of complex numbers}$$

We will start this section with a summary of some well-known results.

In [Buc83], Buchberger introduced the notion of Gröbner Bases and devised an algorithm to compute the Gröbner Base of a given finite set of polynomials. This algorithm has been widely used in the field of computer algebra over the past few years.

Without loss of generality, we can assume that all polynomial equations are in the form of $p = 0$. Let $E = \{p_1 = 0, \ldots, p_n = 0\}$ be a system of polynomial equations, and I the ideal in the ring of all the polynomials generated by $\{p_1, \ldots, p_n\}$. The following close relation between the elements of I and the solutions of E is well known as the Hilbert zero point theorem [Hil90].

Theorem 3.1 *Let p be a polynomial. Every solution of E is also a solution of $p = 0$, if and only if there exists a natural number n such that p^n is an element of I.*

Corollary 3.1 *E has no solution if and only if $1 \in I$.*

Buchberger gave an algorithm to determine whether a polynomial belongs to the ideal. A rough sketch of the algorithm is as follows (see [Buc83] for a precise definition).

Let there be a certain ordering among monomials and let a system of polynomial equations be given. An equation can be considered a rewrite rule which rewrites the greatest monomial

in the equation to the polynomial consisting of the remaining monomials. For example, if the ordering is lexicographic, a polynomial equation, $Z - X + B = A$, can be considered as a rewrite rule, $Z \rightarrow X - B + A$. A rule $L_1 \rightarrow R_1$ is said to subsume rule $L_2 \rightarrow R_2$ if L_2 is a multiple of L_1. A pir of rewrite rules $L_1 \rightarrow R_1$ and $L_2 \rightarrow R_2$, of which L_1 and L_2 are not mutually prime, is termed a *critical pair*, since the least common multiple of their left-hand sides can be rewritten in two different ways. The S-polynomial of such a pair is

$$\text{S-poly}(L_1, L_2) = R_1 \frac{\text{lcm}(L_1, L_2)}{L_2} - R_2 \frac{\text{lcm}(L_1, L_2)}{L_1}$$

If further rewriting does not succeed in rewriting the S-polynomial of a critical pair to zero, the pair is said to be *divergent* and the S-polynomial is added to the system of equations. By repeating this procedure, we can eventually obtain a confluent rewriting system. The confluent rewriting system thus obtained is called a *Gröbner Base* of the original system of equations, and can be characterized by all S-polynomials rewriting to zero. The following theorem establishes the relationship between ideals and Gröbner Bases.

Theorem 3.2 *Let R be a Gröbner Base of a system of equations $\{p_1 = 0, \ldots, p_n = 0\}$, and let I be an ideal generated by $\{p_1, \ldots, p_n\}$. A polynomial, p, belongs to I if and only if p is rewritten to 0 by R.*

We could decide entailment based on 3.2, and satisfiability by using the Buchberger Algorithm to incorporate the polynomial to the Gröbner Base as per 3.1, but for our purposes there are some problems with this approach. Firstly, tentatively modifying the Gröbner Base in order to check guard satisfiability is undesirable, particularly if we wish to do so for multiple clauses simultaneously. Secondly, since the relation between the solutions and the ideal described in theorem 3.1 is incomplete, the method of theorem 3.2 is incomplete with respect to deciding entailment. For example, since Gröbner Base of $\{X^2 = 0\}$ is $\{X^2 \rightarrow 0\}$, rewriting using this Gröbner Base cannot show that $X = 0$ is entailed. There are several approaches to solve the entailment problem:

1. Use the Gröbner Base of the radical of the generated ideal, I. i.e. $\{p | p^n \subset I\}$. Although it is theoretically possible to compute, there is no efficient implementation.
2. Use the Buchberger Algorithm to add p to the Gröbner Base and then check that the resulting Gröbner Base is equivalent to the original. Unfortunately, this seems as difficult as finding the Gröbner Base of the radical.
3. Use the Buchberger Algorithm to add $p\alpha$ to the Gröbner Base, where α is a new variable. p is in the old ideal iff $1 \in$ the new ideal. This has the unfortunate side-effect of changing the Gröbner Base.
4. Find n such that p^n is rewritten to 0 by the Gröbner Base of the generated ideal. Since n is bounded[CaG88], this is a complete decision procedure. Since the bound is very large, we may prefer the incremental solution of repeatedly raising p to a small positive integer power and rewriting it by the Gröbner Base.

Since we have not run into problems with incompleteness in practice, we have chosen not to implement any of the above strategies.

3.1 Parallel Constraint Solver

There are two main sources of polynomial-level parallelism in the Buchberger Algorithm, the parallel reduction of a set of polynomials, and the parallel checking for subsumption and critical pairs of a new rule against the other rules. Since the latter is inexpensive, we must concentrate on parallelizing the coarse-grained reduction component for shared-memory architectures. However, since the convergence rate of the Buchberger Algorithm is very sensitive to the order in which polynomials are converted into rules, an implementation must be careful to select "small" polynomials early for inclusion in the developing basis.

3 SOLVING RATIONAL POLYNOMIAL CONSTRAINTS

The key idea underlying the algorithms in this paper is that of sorting a distributed set of polynomials, and we will use the "distributed enumeration sort" [Akl85] as our point of departure.

We begin by considering the "distributed enumeration sort" algorithm, which is suitable for distributed memory machines. In the sort algorithm, each processor has a complete set of the input items, and a copy of the *ownership* function which is a one-to-one function from items to processors[2]. Each processor independently compares the item it owns to all the other items in order to determine the item's rank in the sorted sequence. The method for outputting the items in sorted sequence chosen, because of its applicability to the Buchberger algorithm, is that each processor listens to the output of all the other processors, and outputs its equation when the count reaches the item's rank.

The sorting algorithm is adapted as follows. Each processor contains a complete set of basis polynomials (called *rules*) and non-basis polynomials, and a load-distribution function ω which logically partitions the polynomials by specifying which processor "owns" what polynomials. The position in the output (rule) sequence of each polynomial is calculated by its owning processor based on an associated key (for example, the leading power product) which is identical in every processor, and does not change during reduction. Each polynomial is output when it becomes the smallest one remaining. The critical-pairs and subsumptions are calculated independently by each processor, so that the processors' sets of polynomials stay synchronized. As a background task, each processor rewrites the polynomials it owns, starting with those lowest in the sorted order. Termination of the algorithm is detected independently by each engine, when the input equation stream is closed, and there are no non-basis polynomials remaining.

The dynamic problem requires more complex control, in order to prevent the arrival of input polynomials at different times at each processor from causing processors to have inconsistent views about the set of non-basis polynomials and possibly about the output (rule) sequence. Figure 2 shows the algorithm for the dynamic case. This version requires additional information about the basis and non-basis sets of each engine to be made known, eventually, to every other engine.

A serious drawback to the algorithm is that it cannot take advantage of "magic polynomials". That is, since the key which determines the output position of a polynomial is fixed before reduction begins, the key is only a rough approximation of the actual preferability of a polynomial after reduction. A possible refinement, not addressed here, is to resort the set of polynomials within each processor inside the same "output slots" owned by that processor.

Since the result for the static algorithm is straightforward, and a special case of the result for the dynamic algorithm, we will only prove correctness for the dynamic version. We would like to show that the processors have the same view of the output (rule) sequence.

Lemma 3.1 *For every $t \geq 0$, exactly one processor outputs to Channel[t].*
Proof by induction on t. Assume $t = 0$. Since B_i is updated exactly when t is incremented, we have $B_i = \mathcal{B}_i = 0$ and $P_i = K_i$. We call a processor i synchronized *if $\mathcal{S}_i = K_i$: only synchronized processors can output (line 12). By definition, $\mathcal{S}_i \subseteq \bigcap_j K_j$, and so for all synchronized processors $K_i = K = \bigcap_j K_j$. Therefore there is a unique minimum $p \in P_i$. Let $m = \omega(p)$. If processor m is synchronized, then it outputs p as soon as p has been fully rewritten, otherwise it waits until synchronization (which will eventually occur, if S is finite). In either case, t is incremented. After output, K_m will not change until $\mathcal{B}_m = B_m$ (line 9), which also freezes the value of K. We are then guaranteed that no (other) engine can output until receiving p, and incrementing t.*
Assume $t = t_1 > 0$. Now $P_i - K_i$ are the identical sets of critical pairs from the first t_1 rules. We argue similarly to the base case to obtain the required result.

Corollary 3.2 *Each processor receives the same sequence of rules.*

[2] This idea is easily generalized to a many-to-one ownership function.

```
     comment
          S = stream of polynomials.
          S_i = subset of S that engine i knows has been received by every engine.
          B_i = subset of B_i that engine i knows has been received by every engine.
          Code to maintain S_i and B_i is omitted.

(1)  do i=1, N
(2)       spawn engine(i,S,Channel) on processor i

(3)  engine(I,S,Channel)
(4)       S_i := P_i := K_i := ∅
(5)       B_i := B_i := ∅; t := 0
(6)       do forever
(7)            choose
(8)                 guard     receive X from S
(9)                           B_i = B_i
(10)                do        K_i := K_i ∪ {X}, P_i := P_i ∪ {X}
(11)                guard     (p := min(P_i)) is irreducible w.r.t. B_i
(12)                          ω(p) = i, S_i − K_i
(13)                do        output p to Channel[t+1]
(14)                          P_i := P_i ∪ {spoly(p,q) | q ∈ B_i} − {p}
(15)                          B_i := B_i ∪ {p}
(16)                guard     receive p from Channel[t++]
(17)                do        P_i := P_i ∪ {spoly(p,q) | q ∈ B_i} − {p}
(18)                          B_i := B_i ∪ {p}
(19)                guard     (L := {q | q ∈ P_i, ω(q) = i, p is reducible by B_i}) ≠ ∅
(20)                do        Rewrite L by B_i
(21)                guard     P_i = ∅, S is closed
(22)                do        output B_i to Channel[t]
(23)                          stop
(24)           endchoose
(25)      enddo
```

"The **choose** (**guard** Cond **do** Action)* **endchoose** construct specifies a non-deterministic guarded choice. Execution will suspend until at least one of the conditions obtains, and then the action corresponding to one of the guards whose condition obtains will be executed; the testing of guard conditions has no observable effect until an associated action is chosen.

[6]The algorithm for the static problem is obtained by changing all references to the stream S to the set of input polynomials P, replacing line(4) with "$P_i := P$", and deleting the framed code.

Figure 2: Algorithm for Dynamic Problem

Theorem 3.3 *For all $p, q \in S$, S-poly(p,q) rewrites to zero.*
The proof follows easily from the above corollary.

3.2 Implementation and Results

The dynamic algorithm was implemented on the Multi-PSI, a distributed-memory multiprocessor designed as a development platform for operating systems and applications based on concurrent logic programming concepts. The user-level language, KL1[UeC90, NaI89], is a data-flow language that executes at up to 128 K reductions/second on a single Multi-PSI node.

The central data structure in the implementation is a sorted list of items of work, comprising input polynomials, critical pairs, and requests to simplify rules. Priorities correspond to the key associated with each polynomial. In the current implementation for rules and input polynomials we use the largest power product as the key, and for S-polynomials we use the largest power product after canceling the largest power product of each of the two parent polynomials. The complete execution of one piece of work is broken down into stages; for example, a critical pair is first converted to a S-polynomial, rewritten, and finally normalized. Based on this breakdown, we pipeline the execution of the entire list, giving us maximum overlap between communication and local computation. Although this implementation only deletes critical pairs arising from subsumed rules, a full implementation of Buchberger's criteria for filtering useless critical pairs should also be possible.

The implementation of the $\mathcal{S}$ and $\mathcal{B}$ variables in the dynamic algorithm is based on *ACK* (acknowledgment) messages. However, the additional latency introduced applies only to the acceptance of new input polynomials, and the number of $\mathcal{B}$ related ACK messages can be decreased by updating the $\mathcal{B}$ variables less frequently. Information about processor load is piggybacked onto the ACK messages, in order to construct the ω load-distribution function dynamically (being careful to build it identically on each processor).

Finally, the calculation of the coefficients of non-basis polynomials is improved by delaying until a rule to rewrite the associated power product has been found. At that point, the coefficient expression is evaluated using divide-and-conquer, and compared to zero. This strategy results in several fold speed improvements in some examples.

Table 1: Absolute Performance of Dynamic Algorithm (sec)

Example	SAC	1 PE	2 PE	4 PE	6 PE	8 PE	12 PE	16 PE
Hairer 1	.482	2.311	1.782	1.341	1.160	1.216	1.277	1.598
Katsura 3	4.666	9.371	5.949	4.167	3.775	6.701	3.226	3.385
LTrinks	.888	49.984	30.607	22.002	14.960	14.730	12.911	12.749
BTrinks	138.425	189.016	108.381	83.362	56.906	42.765	44.313	40.590
Katsura 4	†31.077	2395.907	1303.543	783.188	531.225	463.890	390.668	302.958

† Note: The SAC figure uses a different ordering which gives better performance for this problem.

The benchmarks presented here are from the SAC system as reported in Boege et. al.[BoG86]; with the exception of Katsura 4, all examples use total degree reverse lexicographic ordering. The figures for the SAC system/IBM 3080 are given to show qualitative differences with a standard implementation.

Except for Katsura 4, the speedup curve (Figure 3) eventually becomes flat, reflecting the limits of polynomial-level parallelism in these examples. The absolute performance of the algorithm is only fair. However, reimplementing the polynomial and rational arithmetic in a standard von Neumann language should bring about a 1–2 order of magnitude performance improvement in the bulk of the computation (measured at over 90%), without affecting the parallelism. Although reimplementation would change the ratio between computation time and communication time/ latency, we conjecture a significant improvement in

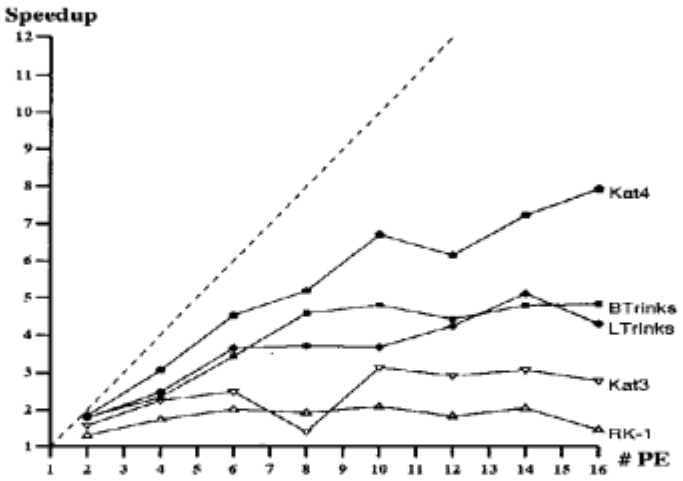


Figure 3: Speedup of Dynamic Algorithm

overall performance to the extent of passing the performance of SAC and other sequential implementations.

4 Conclusions

We have demonstrated the possibility of developing a concurrent constraint programming language GDCC using the KL1 committed-choice concurrent logic programming language. GDCC is an open system, in the sense that KL1 processes obeying a simple protocol can be used as constraint solvers for user-specified domains. We have demonstrated that Gröbner Base techniques are somewhat applicable to the concurrent context, and presented a constraint solver based on a parallelized Buchberger Algorithm for calculating the Gröbner Base on a distributed memory machine. The constraint solver exhibits substantial speedups and reasonable performance. Reimplementation of the low-level routines in a von Neumann language should substantially improve the latter. The algorithm uses broadcast messages exclusively, and it would be interesting to investigate its performance on a hardware and software platform that supports broadcasting efficiently.

References

[AiS88] A. Aiba, K. Sakai, Y. Sato, D. Hawley, and R. Hasegawa. Constraint Logic Programming Language CAL. In *International Conference on Fifth Generation Computer Systems 1988*, pages 263–276, 1988.

[Akl85] Selim. G. Akl. *Parallel Sorting Algorithms*. Notes and Reports in Computer Science and Applied Mathematics. Academic Press, 1985.

[BoG86] W. Boege, R. Gebauer, and H. Kredel. Some examples for solving systems of algebraic equations by calculating grobner bases. *J. Symbolic Computation*, 2(1):83–98, 1986.

[Buc83] B. Buchberger. Gröbner bases:An Algorithmic Method in Polynomial Ideal Theory. Technical report, CAMP-LINZ, 1983.

[CaG88] Leandro Caniglia, Andre Galligo, and Joos Heintz. Some new effectivity bounds in computational geometry. In *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes - 6th International Conference*, pages 131–151. Springer-Verlag, 1988. Lecture Notes in Computer Science 357.

REFERENCES

[ClL90] E. M. Clarke, D. E. Long, S. Michaylov, S. A. Schwab, J. P. Vidal, and S. Kimura. Parallel Symbolic Computation Algorithms. Technical Report CMU-CS-90-182, Computer Science Department, Carnegie Mellon University, October 1990.

[Hen89] P. Van Hentenryck. Parallel constraint satisfaction in logic programming: Preliminary results of chip with pepsys. In *6th International Conference on Logic Programming*, pages 165–180, 1989.

[Hil90] D. Hilbert. Über die Theorie der algebraischen Formen. *Math. Ann.*, 36:473–534, 1890.

[JaL86] Joxan Jaffar, Jean-Louis Lassez, and Michael J. Maher. A Logic Programming Language Scheme. In D. DeGroot and G. Lindstrom, editors, *Logic Programming: Functions, Relations, and Equations*. Prentice-Hall, 1986.

[Mah87] Michael J. Maher. Logic Semantics for a Class of Committed-choice Programs. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 858–876, Melbourne, May 1987.

[NaI89] Katsuto Nakajima, Yu Inamura, Nobuyuki Ichiyoshi, Kazuaki Rokusawa, and Takashi Chikayama. Distributed implementation of KL1 on the Multi-PSI/V2. In *Proceedings of ICLP'89*, pages 436–451, 1989.

[Pon90] C. G. Ponder. Evaluation of 'Performance Enhancements' in algebraic manipulation systems. In J. Della Dora and J. Fitch, editors. *Computer Algebra and Parallelism*, pages 51–74. Academic Press, 1990.

[SaA89] K. Sakai and A. Aiba. Cal: A theoretical background of constraint logic programming and its applications. *Journal of Symbolic Computation*, 8:589–603, 1989.

[Sar89] V. Saraswat. *Concurrent Constraint Programming Languages*. PhD thesis, Carnegie-Mellon University, Computer Science Department, January 1989.

[SaS88] Y. Sato and K. Sakai. Boolean Gröbner Base, February 1988. LA-Symposium in winter, RIMS, Kyoto University.

[Sen90] P. Senechaud. Implementation of a parallel algorithm to compute a Gröbner basis on Boolean polynomials. In J. Della Dora and J. Fitch, editors, *Computer Algebra and Parallelism*, pages 159–166. Academic Press, 1990.

[Sie90] K. Siegl. Gröbner Bases Computation in STRAND: A Case Study for Concurrent Symbolic Computation in Logic Programming Languages. Master's thesis, CAMP-LINZ, November 1990.

[UeC90] Kazunori Ueda and Takashi Chikayama. Design of the kernel language for the parallel inference machine. *Computer Journal*, December 1990. To appear.

[Ued86] K. Ueda. *Guarded Horn Clauses*. PhD thesis, University of Tokyo, Information Engineering, 1986.

[Vid90] J. .P. Vidal. The Computation of Gröbner bases on a shared memory multiprocessor. Technical Report CMU-CS-90-163, Computer Science Department, Carnegie Mellon University, August 1990.

ＫＬ１による１階述語論理プルーバ
- その後のＭＧＴＰ -

長谷川 隆三 藤田 正幸 藤田 博 越村 三幸
ICOT 第５研究室

1 はじめに

第１回ＫＬ１ワークショップでは、論理式中に現れる変数をＫＬ１でどう扱うかという、いわゆるメタプログラミングの問題点について言及し、ＫＬ１変数の利点を十二分に生かしたモデル生成型定理証明系（ＭＧＴＰ）の実現技法と、その簡単なインタプリタ（simple 版）の構成を示した。ここで開発したＭＧＴＰ実現技術の要点は以下の３点である。

1. 問題節を対応するＫＬ１節に変換する。
2. 対象を range-restricted な節集合に限ることにより、節のリテラルを モデルにつき合わせて充足性を判定する操作（連言照合：conjunctive matching）をＫＬ１のヘッドユニフィケーションで実現する。
3. ＫＬ１節の呼びだし時に、自動的に新変数を確保する。

今回のＫＬ１ワークショップでは、「その後のＭＧＴＰ」という副題の意に沿って、我々がこれまでに取り組んできた課題のいくつかをとりあげ、それらの解決策を示すとともに、ＫＬ１プログラミングの効用にもふれる。これらの課題は、大きく定理証明特有の課題とメタプログラミング特有の課題に分かれるが、後者については別稿「メタプログラミングのためのＫＬ１ユーティリティ」（越村他）で論じる。

2 冗長性の除去

ＭＧＴＰの効率向上を図るには、連言照合フェーズでの重複計算を防ぐことが必須である。まず考えられる方法は、Rete ネットをＫＬ１プロセスとして実現することであるが、non-Horn 節の扱いに難がある。この場合、一種の多重世界問題を考慮する必要があり、(i) ケース分割が生じる度に、これまでに作られたプロセスネット全体をコピーするか、(ii) 各分岐先で親元のプロセスネットを共有できるように、ストリーム中を流れるリテラルインスタンスに、どの枝（モデル）に属しているかを示す識別子を付与しなければならないが、いずれにしても実現コストが高くなってしまう。

これらの問題の１つの解決策として、連言照合の履歴であるリテラルインスタンスをプロセス中でなくスタック中に保持し、Rete 風アルゴリズムを実現する、ＲＡＭＳ（ramified stack）方式を開発した。本方式は、連言照合における冗長なつき合わせを防ぎ、non-Horn 節に対して共通モデルの共有を可能にするほか、リテラルインスタンスのキャッシングにより照合時間を短縮できる。しかしその反面、本方式には中間の照合結果を保持するための記憶域が膨大になるという欠点があった。

そこでもう１つの解決策として、動的に冗長性を防ぐのではなく、問題節を前処理することによって、静的に冗長性を防ぐ新たな方式を考案した。とりあえず、これをＭＥＲＣ（multi-entry repeated combination）方式と呼ぶことにする。

冗長性を排除する原理は、新たに生成されたモデル要素を δ、以前に作られたモデル要素の集合をＭとしたとき、節の前件部のリテラルの少なくとも１つは δ と照合するようにすることである。こ

のため、前件部のリテラルに対して、δとMの重複組合せを考え、この組合せ個数分のＫＬ１節を用意する（各ＫＬ１節はある重複組合せのパタンについて連言照合の掛け合わせを実行する）。

ＭＥＲＣ方式はＲＡＭＳ方式のようなキャッシング効果は持たないが、つき合わせパタンの重複は完全に防いでおり、履歴保持のための記憶域を不要にしている。照合時間を短縮するか、記憶域を削減するかのいずれを重視するかは、トレードオフの問題である。両方式の優劣をくだすデータはまだないが、一般的に言って、推論が深くなる（モデルが大きくなる）程、ＭＥＲＣ方式の方がより実際的となろう。

3　項インデキシング

ＲＡＭＳ版ＭＧＴＰでは、モデルおよびリテラルインスタンスの記憶域として線形リストを用いていたため、連言照合や包摂テスト（subsumption test）の際、モデル要素の参照に（モデルの大きさに比例する）線形時間を要した。これを改善するには、次の項インデキシングを導入することが不可欠となる。

項インデキシングの方式にはいくつかあるが、古典的な選別木（discrimination tree）による方法は平均的に良い実行効率が得られており、実現も容易である。我々はＫＬ１のベクタを用いてこれを実装した。例えば、項 f(a,b) は括弧を省略して３つのシンボル f,a,b からなるストリングとみなすことができる。このストリングを左から右へたどり、シンボルが出現する毎にアークをのばしていくと、一本の枝が形成される（アークには各シンボルが付与される）。複数の項に対してできる枝を共通 prefix で束ねると木ができるが、この木は項の集合をコンパクトに表現したものと考えることができる。木の各節点には、インデクスによって次の節点への行先が決定できるように、その出力アーク数分のエントリを持つベクタを一本ずつ切る。

選別木方式の利点の１つは、prefix order に木をたどったときの共通軌跡が共有されることである。１つの項と項集合との間で、ユニフィケーションや（前向き／後ろ向き）包摂テストを行う際、項メモリ中に保持された項集合を表わす木ををたどることによって、それらの操作の探索範囲を絞ることができる。この効果は問題に現れる定数／関数記号の数やモデルの規模（推論の深さ）が大きくなるにつれ顕著になる。

4　並列化

ＭＧＴＰの並列化を考えるにあたって、問題節の種類（Horn か non-Horn か）、とモデルの特徴（ground か non-ground か）を押えておくことが肝要である。したがって、(i) ground で non-Horn、(ii) ground で Horn、(iii) non-ground で Horn、(iv) non-ground で non-Horn、の４種の組合せについて考える。(iv) の場合には、disjunctive literals 間の共有変数をどう扱うかといった古典的な問題が存在し、うまい解決策が見当たらないこと、および自動証明の分野でとりあげられている定理は殆ど (i) ～ (iii) までであることから、ここでは (iv) の場合は除外する。

さて、(i) の場合は、ケース分割によって指数的にモデルが分岐していき、これらのモデル毎に独立に連言照合処理を行うことができるので、多大なＯＲ並列性が期待できる。Muliti-PSI のようなローカルメモリ方式のマルチプロセッサシステムでは、処理の粒度を大きくして、通信コストをできるだけ低減する必要があるが、(i) はこのシステムにあつらえ向きの問題領域と言える。「ケース分割によってプロセッサ台数分だけ探索木を展開し、それ以降は各プロセッサで逐次にモデル生成を実行する」という、最も簡単なプロセッサ割り当て方式を用いて、n-queens 問題、pigeon hole 問題等の走行実験を行った結果、（当然予想されたことではあるが）ほぼ線形の台数効果が得られた（12-queens の全解探索において、64 台で約 54 倍の性能向上）。

(ii), (iii) の場合は、ケース分割によるＯＲ並列効果が望めないので、連言照合および包摂テストを並列化することが課題となる。群の定理など、数学的な問題では、包摂テストが全計算時間の９０％以上を占める傾向にあり、この部分を並列化する効果は特に大きい。その場合の問題点は、グローバルメモリである項メモリに対するアクセス集中をいかに抑え、通信頻度をいかに低減できるかである。現在、OTTER で取られている方式を参考に、検討を進めているところである。

参考文献

[MB88] Manthey, R., and Bry, F., SATCHMO: a theorem prover implemented in Prolog, in *Proc. of CADE 88, Argonne, Illinois,* 1988.

[Fuc90] 淵一博, KL1 プログラミング雑感 - Prover の並列化の体験より - , in *Proc. of KL1 Programming Workshop '90,* pp.131-139, 1990.

[FH90] 藤田博, 長谷川隆三, KL1 による定理証明プログラム, in *Proc. of KL1 Programming Workshop '90,* pp.140-149, 1990.

[HFF90] Hasegawa, R., Fujita, H. and Fujita, M., A Parallel Theorem Prover in KL1 and Its Application to Program Sysnthesis, *Italy-Japan-Sweden Workshop*, ICOT TR-588, 1990.

[FKKFH90] Fujita, H., Koshimura, M., Kawamura, T., Fujita, M., and Hasegawa, R., A Model-Generation Theorem Prover in KL1, *Joint US-Japan Workshop,* 1990.

[HKFFK90] Hasegawa, R., Kawamura, T., Fujita, M., Fujita, H., and Koshimura, M., MGTP: A Hyper-matching Model-Generation Theorem Prover with Ramified Stacks, *Joint UK-Japanese Workshop,* 1990.

[FH90] Fujita, H. and Hasegawa, R., A Model Generation Theorem Prover in KL1 Using a Ramified-Stack Algorithm, To appear in *Proc. of ICLP'91,* , ICOT TR-606, 1991.

[Sti89] Stickel, M.E., The Path-indexing method for indexing terms, Technical Note 473, Artificial Intelligence Center, SRI International, Menlo Park, California, October 1989.

並列 ATMS

中島 誠† 太田 好彦‡ 井上 克巳‡
†(財) 日本情報処理開発協会 ‡(財) 新世代コンピュータ技術開発機構

1 はじめに

疎結合型並列計算機 Multi-PSI 上で KL1 による並列 ATMS の開発を行なっている.
仮説推論は, 常に正しい知識である事実の集合と, 常に正しいとは限らない知識である仮説の集合とに基づき, 事実の集合と矛盾しない仮説部分集合を加えて結論を導く推論形態であり, 知識システムを構築するための重要な要素技術である [Inoue 88]. しかしながら, 推論過程で加える仮説部分集合の無矛盾性を検査せねばならず, 推論速度の点で問題がある.

従来提案されている ATMS (Assumption-based Truth Maintenance System) [deKleer 86] は, 命題論理の事実集合と仮説集合を基に, データ (命題) の真偽を管理するシステムである. ATMS は新たなデータが付け加わった時に既存のデータとの整合性を保ち, 矛盾が発見された場合に仮説集合を見直してその無矛盾性を管理することを主な機能とし, これをデータ構造および, アルゴリズムの工夫により効率的に行なうことを実現している. しかしながら, 不完全な知識ベースを用いる問題解決には指数オーダの計算量が必要であると考えられることから, より一層の高速化が望まれている. [Dixon 88] では, コネクションマシン上で ATMS の並列化の試みが行なわれているが, 入力される知識 (データ) の数に対して, 指数オーダのプロセッサが必要になる. また, 疎結合並列計算機上での高速化についてもいくつかの試みが行なわれている [Harada 90] [和田 90].

これら従前の試みの中にあって本システムの特徴は, 入力されるデータの数だけのプロセス[1]を生成する並列アルゴリズムを採用している点にある. この方式により, 入力データの数に対して生成するプロセス数の爆発が防げる. さらに, これらの生成されたプロセスを複数のプロセッサへ割り当てることが, データの分配の問題に帰着されるため, 比較的簡単に行なえると考えられる.

以下本稿では, 基本的 ATMS の概要を述べ, つづいて, それをもとに並列化の実現方法を示し, さらに具体例を用いた計測値をあげて, 考察を加える.

2 ATMS

ここでは基本的な ATMS の概要を示す. ATMS の並列化の具体的な実現方法については章を改めて論じる.

ATMS[deKleer 86] は, 仮説集合と命題論理のホーン節 (理由付け (*Justification*) と呼ぶ) の集合を入力とし, アトム (データ (*Datum*) と呼ぶ) の真偽の状態 (あるいは信念の状態) を管理し, 出力する.

理由付けとして入力されるホーン節は次に示すいずれかの形式である.

$$a_1, \cdots, a_n \Rightarrow b.$$
$$a_1, \cdots, a_n \Rightarrow \bot.$$

ここに, $a_i (i = 1, \cdots, n; n \geq 0)$ 及び b は命題論理のアトム, 記号 $\bot$ は偽 (false) を表し, 記号 $\Rightarrow$ の左の各アトムを前件, 右を後件と呼ぶ.

ATMS 内部では各データは以下に示すデータ構造 (これをノードと呼ぶ) で表現され, その信念の状態が管理されている[2].

$$\gamma_{Datum} : < \text{データ}, \text{ラベル}, \text{理由付け} >.$$

[1]本稿では, プロセスという用語を解くべき問題中で並列に処理されるべきサブタスクの単位とする. これは KL1 プログラミングでいうところのプロセスよりも広い意味で用いている.

[2]データ X が真であるという仮説を表すノードを特に仮説ノード (assumption node) と呼ぶ.

現時点までに入力された仮説の集合の部分集合を環境と呼ぶ. また, 現時点まで入力された全ての理由付けの集合を J とすると, J とある環境から $\perp$(偽;false) が導かれるときその環境を矛盾環境 (*Nogood*) と呼び, J とある環境から $\perp$ が導けないときその環境を無矛盾 (consistent) な環境と呼ぶ. あるノードが信じられている (IN 状態と呼ぶ) とは, ある無矛盾な環境 E があって J と E からノードのデータが導かれることである. IN 状態のノードは, 後から追加される知識によってそのノードが信じられていない状態 (OUT 状態と呼ぶ) になる可能性を持っている. ノードのラベルはそのノードが信じられている極小の環境の集合である. ATMS は, 入力された理由付けをその後件に対応するノード (後件ノードと呼ぶ) の理由付けの項に, 前件に対応するノード (前件ノードと呼ぶ) の集合を付加して記憶している.

ATMS に新たな理由付けが与えられるとそれを基に, ラベルの更新が行なわれる. 以下は a_i (i=1,...,n:n≥0) を前件とする理由付けが与えられたときのラベルの更新手順を表したものである. また, ラベル計算の簡単な例を付録に示す.

[ラベル更新手順]

1. 後件ノードのラベルを L_{old}, 前件ノード a_i のラベルを L_i と示す. L_i に含まれる各環境の集合に関して直積をとり, L' とする.

$$L' = \{x | x = \bigcup_i E_i, E_i \in L_i\}$$

   L' は, 環境をビットベクタ[3]で表現しておくとビット演算 or により計算できる.

2. $L_{old} \bigcup L'$ から矛盾環境を取り除き, さらにその集合の極小の要素 (他の要素を包含しない要素) の集合を L_{new} とする.

3. もし L_{old}=L_{new} ならばこのノードに関するラベル計算は終了.

4. もし後件が偽であるなら, L_{new} に含まれるすべての環境について, 矛盾環境として登録し, 自分以外のノードのすべてのラベルからそれを包含する環境を除去し終了する.

5. もし後件が偽でないなら L_{new} を後件ノードの新しいラベルとする. さらに, このノードを前件ノードとする理由付けに関して 1. から実行する.

3 並列 ATMS の実現方法

本章では ATMS の並列化の実現方法について論じる.

3.1 データ構造

並列 ATMS では, 外部から与えられたデータに対応して以下のデータ構造を有するプロセスを生成し, これによってノードを表現する:

γ_{Datum}: <データ, ラベル, *Justifications*, *Consequents*, *Attention*>.

Justifications: このデータを後件とする理由付けの前件ノードへメッセージを送信する出力ストリーム (これを $J-$ ストリームと呼ぶ) の集まり.

Consequents: このデータを前件に持つ理由付けの後件ノードへメッセージを送信する出力ストリーム (これを $C-$ ストリームと呼ぶ) の集まり.

Attention: 偽に対応するノードから送られる矛盾環境の削除命令を受信する入力ストリーム (これを $A-$ ストリームと呼ぶ).

[3]仮説集合 H がその要素として n 個の仮説 $A_1, A_2, \cdots, A_n$ を含む時, この n 個の仮説を n 個のビット列 $b_1, b_2, \cdots, b_n$ と対応させた表現. 各ビットは対応する仮説がある時 1, 無い時 0 の値をとる. すると 0 個以上の仮説を組み合わせた環境 E もまたビットベクタで表現できる.

データおよびラベルは2章において定義されたものと同じである．$J-$ ストリームは理由付けに基づいてノードのラベルを計算する際に，その理由付けの前件ノードからラベルを収集するメッセージを送信するためのストリームである．さらに，$C-$ ストリームはラベルの更新が起こった場合に，後件ノードに対して，そのラベルの更新命令を伝える．*Justifications*, *Consequents* は，ノードを表すプロセスへのストリームの集まりとなっている．

また，偽に対応するノードは，極小の矛盾環境 (他の矛盾環境を包含しない矛盾環境，以下単に矛盾環境と呼ぶ) の集合 (*Nogood* データ・ベースと呼ぶ) をそのラベルとして保持し，自身以外の全てのノードへ $A-$ ストリームを通じて矛盾環境を伝達する．他のノードの *Attention* はその矛盾環境を受信するためのストリームである．

本並列 ATMS では，基本的にノードのみをプロセスとして表現しているので入力されたデータの数のプロセスが生成される．

3.2 並列化機能

本並列 ATMS では以下のような処理を並列に実行することが可能である．

1. **外部から ATMS に与えられる複数の命令の並列実行**

   ここに，命令とは，以下の3項目である．

   - 仮説の追加
   - 理由付けの追加
   - データの信念の状態の問合せ

   命令の並列実行は，各々のデータを管理するプロセスがそれぞれ独立して実行されることによって実現された機能である．以下に示すのは本並列 ATMS において実際に用いる仮説ノードの生成命令の例である．

   $$\text{create-assumption}(a,As)$$
   $$\text{create-assumption}(b,Bs)$$
   $$\text{create-assumption}(c,Cs)$$

   As,Bs,Cs はそれぞれデータ a,b,c が真であるという仮説ノードを表すプロセスへのストリームとなっている．各々の仮説ノードは互いに独立しており，同時生成が可能である．また，それぞれのノードに対する理由付けの追加，信念の状態の問い合わせは，各ストリーム (As,Bs,Cs) に対して対応する命令を送ることで実行される．したがって，複数のノードのそれぞれに対する理由付けの追加も並列に実行できる．さらに仮説の追加と理由付けの追加が混在するような場合も並列に実行できる．ただしデータの問い合わせについては，それ以前に ATMS に与えられた命令が全て処理された後行なわれる．これにより，問い合わせを行なった時点でのラベルの正当性が保証される．

2. **理由付けに伴うラベル更新の並列実行**

   ここで，ラベル更新とは具体的に以下の処理を行なうことである．

   (a) 各前件ノードのラベルの収集

   与えられた理由付けの前件ノードから，それらのラベルを集める．

   (b) 後件ノードのラベルの更新

   (a) で集めたラベルを基に後件ノードのラベルを更新する．

   (c) 更新ラベルの伝播

   このノードを理由付けの前件に持つ全てのノードに対して，更新されたラベルを伝播させる．

   (d) 矛盾する環境の削除

   もし偽に対応するノードのラベルが更新された場合には，新たに追加された矛盾環境を自分以外の全てのノードのラベルから削除する．

このうち，並列に実行できるのは(a)と(c)および(d)である．図1に示した理由付けとそれによって構築されるプロセスのネットワークを例にラベル更新の並列実行を考える．図1のプロセスがそれぞれ異なったプロセッサに割り

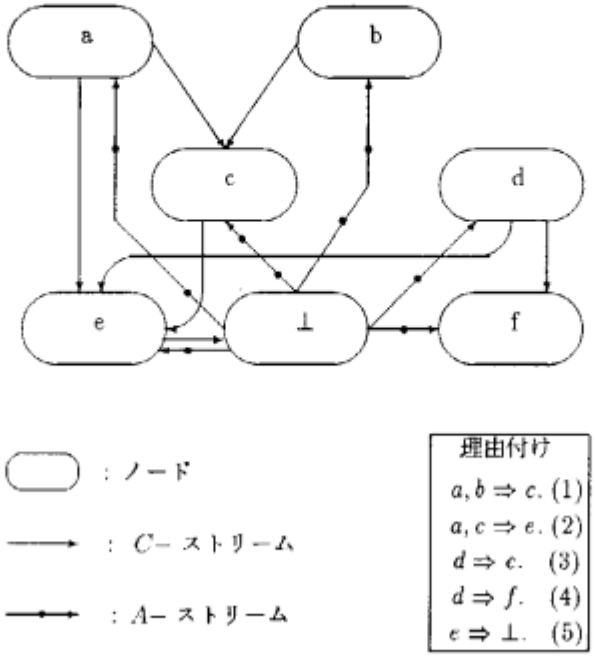


図 1: 本並列 $ATMS$ によって生成されるプロセス・ネットワークの例

当てられた時，理由付け(1,3,4)のようにお互いの後件ノードをそれぞれの前件に持たない場合は，各々の理由付けに基づくラベル計算は独立して並列に実行することができる．また，理由付け(1,2)と(2,3,5)とによって生じるラベルの伝播処理もパイプライン処理によって並列に実行することが可能となっている．

以下に並列ATMSにおけるラベルの更新手順を示す．

[並列ラベル更新手順]

ATMSにある理由付け $a_1, \cdots, a_i, \cdots, a_n \Rightarrow x$ $(1 \leq i \leq n; n \geq 0)$ が与えられた時，その理由付けの後件ノードを表すプロセス(γ_x と示す)において以下の手続きを実行する．

1. γ_x は，その前件ノード γ_{a_i} への $J-$ ストリームを有しているので，γ_{a_i} に対してそのラベル L_i の問い合わせを発する．これを受信した γ_{a_i} は自身のラベルを返す．γ_x は L_i を得ないと次の処理ができないので，この問い合わせ処理のプライオリティは他の処理よりも高くしている[4]．これにより，個々に独立したノードから，ほとんど同時にラベルの収集を行なうことができる．

2. γ_x は $L' = \{\varepsilon | \varepsilon = \cup_{i=1}^{n} E_i,\ E_i \in L_i\}$ を計算する．ここに，環境のビットベクタ表現を用いているため，集合演算 $\cup$ はビット演算 or でよい．また，γ_x はすべての前件ノードのラベルが揃わなくても，少なくとも2つのラベルが送られてくれば計算を開始することができる．

3. 現在の自身のラベルの内容を L として，$x \neq \bot$ なら，$L \cup L'$ から矛盾環境を取り除いた集合の極小の要素の集合を求めて L'' とする．$x = \bot$ なら，単に $L \cup L'$ の極小の要素の集合を求めて L'' とする．

4. $L = L''$ ならばこの処理を終わる．そうでなければ L'' を自身のラベルとする．

[4] KL1実装上は，節の優先関係指定(alternatively)によって実現している．

5. $x \neq \perp$ であるならば，この x を理由付けの前件に持つ全てのノードに対して $C-$ ストリームを介してラベルの更新命令を送る．それを受けたノードは，自身の *Justifications* に格納してある理由付けに基づいて 1. より計算を始める．$x = \perp$ ならば，新しく追加された矛盾環境を $A-$ ストリームに送信する．$A-$ ストリームを介して矛盾環境を受信したプロセスは，自分自身のラベルの要素で矛盾環境を含むものを全て削除する．

後件が相異なる複数の理由付けが与えられた時には，各々並列にそれぞれの後件ノードに対応するプロセスにおいて，上記の手順を並列に実行することができる．

4 実測結果

本章では，並列 ATMS の性能評価を行なう．

並列 ATMS の実行性能を評価するために 8×8 のチェス盤上に 8 個のクィーンが衝突しないように配置する 8 クィーン問題を取り上げた．その問題解決のために，ATMS に与える仮説集合，矛盾環境および 8 個のクィーン (以下 Q と示す) の配置に関する理由付けを図 2のように用意した．

ここで，$q(i,j)(1, \leq i,j \leq 8)$ は，チェス盤上 i 行 j 列にクィーンを置くという仮説を意味する．また矛盾環境 {q(i,j), q(m,n)} $(1 \leq i,j,m,n \leq 8)$ は以下の条件を満たすものとした．

- $i = m$ あるいは $j = n$
- $|i-j| = |m-n|$

$$(Assumptions)\begin{cases} q(1,1), & q(1,2), & q(1,3), & \cdots & q(1,8), \\ q(2,1), & q(2,2), & \cdots & \cdots & q(2,8), \\ q(3,1), & q(3,2), & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ q(8,1), & q(8,2), & q(8,3), & \cdots & q(8,8). \end{cases}$$

$$(Nogoods)\Big\{ \{q(1,1),q(1,2)\}, \quad \{q(1,1),q(2,1)\}, \quad \cdots \quad \{q(i,j),q(m,n)\}, \quad \cdots \quad \{q(8,7),q(8,8)\}. \Big.$$

$$(6\ Columns\ Justifications)\begin{cases} q(1,1) \Rightarrow pos1. & q(1,2) \Rightarrow pos1. & \cdots & q(1,8) \Rightarrow pos1. \\ q(2,1) \Rightarrow pos2. & q(2,2) \Rightarrow pos2. & \cdots & q(2,8) \Rightarrow pos2. \\ & & \cdots & \\ q(6,1) \Rightarrow pos6. & q(6,2) \Rightarrow pos6. & \cdots & q(6,8) \Rightarrow pos6. \end{cases}$$

$$(8\ Queens\ Justificatins)\begin{cases} pos1, & pos2, & pos3, & pos4, & pos5, & pos6, & q(7,1), & q(8,1) & \Rightarrow & queen1. \\ pos1. & pos2, & pos3, & pos4, & pos5, & pos6, & q(7,1), & q(8,2) & \Rightarrow & queen2. \\ & & & & & \cdots & & & & \\ pos1, & pos2, & pos3, & pos4, & pos5, & pos6, & q(7,8), & q(8,8) & \Rightarrow & queen64. \end{cases}$$

図 2: 8 クィーン問題理由付け例

まず，8×8 の盤上の 1 つの桝目に 1 つの Q をおくことを仮説とする (*Assumptions*)．次に二つの Q が互いにとり合う位置にある組合せを矛盾環境とする (*Nogoods*)．また，6 個の Q に 1 から 6 行まで行ごとに理由付けを導入する (*6 Columns Justifications*)．さらに，この 6 個の Q と 7,8 行における残り 2 つの Q の配置に対応する理由付けを追加する (*8 Queens Justificatons*)．8 クィーンの解は *queen1* から *queen64* までのラベルの和集合で示される．

ここで生成されるノードは，$q(i,j)$ $(1 \leq i,j \leq 8)$, $pos\ k$ $(1 < k \leq 6)$, $queen\ l$ $(l \leq l \leq 64)$, および $\perp$ であるから，その数は，

$$8 \times 8 + 6 + 64 + 1 = 135$$

である．このノードを以下の条件で各プロセッサに静的に割り当てる．M は使用するプロセッサ数を表し，PE は各ノードを分散させるプロセッサ番号を示す．

$$PE_{q(i,j)} = (i-1)*8+j \mod M \quad (1 \le i,j \le 8)$$
$$PE_{posk} = k \mod M \quad (1 \le k \le 6)$$
$$PE_{queenl} = l \mod M \quad (1 \le l \le 64)$$

M の値を 1 から 64 まで段階的に変更して，実行を行なった結果を図 3 に示す．この結果によると，1 台のみのプロセッサで実行を行なった場合に対して，8 台の時の実行速度は約 6.5 倍，64 台の時は約 23 倍早くなることがわかる．

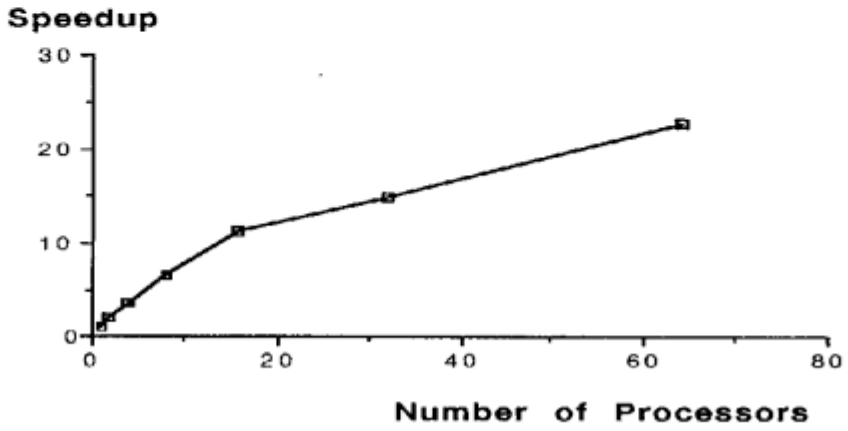


図 3: 8 クィーン測定結果

5 考察

図 2 の *8 Queens Justifications* に示した 64 個の理由付けでは，*pos1* から *pos6* までの前件ノードがすべて同じであり，*queen1* から *queen64* までのノードで重複したラベル計算をしている．これは以下の理由による．

8 クィーンの解法では，8 個の Q について 1 行から 8 行まで行ごとに理由付けを導入して (*pos1* から *pos8* の作成)，*pos1* から *pos8* を前件に持つ後件ノード γ_{queen} を作成する方法がまず考えられる[5]．しかしながら，この方法ではノード γ_{queen} に負荷が集中することになり，マルチプロセッサによる並列実行の有効性を発揮できない．このことから明白なようにノードをプロセスとして並列実行を行なうことの限界は，ラベル計算が一つのノードへ集中する場合にある．これを回避するためには *8 Queens Justifications* のように理由付けの方法に工夫を加えて，負荷を分散させる必要がある．その結果，一つのノードへの理由付けを *queen1* から *queen64* に対する 64 の理由付けに分散して並列に処理することができ，図 3 に示したような台数効果が得られる．ここで，理由付けの分解による負荷の分散は，使用できる計算機資源を考慮した適度な分散を行なわねばならないことを付記しておく．すなわち，8 個のクィーンの全ての配置を理由付けとして導入して[6]，負荷の分散を行なうことが考えられるが，この方法では 8^8 の理由付けが必要となり，メモリの不足を招いて解を求めることができなかった．

[5] 図 2 の 6 *Columns Justifications* に以下を追加し，

$$q(7,1) \Rightarrow pos7. \quad q(7,2) \Rightarrow pos7. \quad \cdots \quad q(7,8) \Rightarrow pos7.$$
$$q(8,1) \Rightarrow pos8. \quad q(8,2) \Rightarrow pos8. \quad \cdots \quad q(8,8) \Rightarrow pos8.$$

8 Queens Justifications の代わりに次の理由付けを用いる．

$$pos1,pos2,pos3,pos4,pos5,pos6,pos7,pos8 \Rightarrow queen.$$

[6] 次のような理由付けになる．

$q(1,x_1),\cdots,q(i,x_i),\cdots,q(8,x_8) \Rightarrow queen(x_1,\cdots,x_i,\cdots,x_8). \quad (1 \le x_i \le 8, 1 \le i \le 8)$

[Dixon 88] でインプリメントされたATMSは，13クィーン問題で逐次処理の70倍の高速化がなされたと報告している．しかしながら，この基本的なアルゴリズムは，無矛盾な環境下で信じられているデータを各プロセッサが保存するものである．したがって，入力された仮説の数の指数オーダのプロセッサが必要になる．そこで，極大の無矛盾な環境をプロセッサに割り当てる方法も提案されているが，これもまた，最悪の場合において指数オーダのプロセッサが必要になる．

Multi-PSI上で稼働するATMSは，[Harada 90] と [和田 90] に述べられている．前者のシステムでは，5プロセッサのときに2倍の高速化が観測されているが，1つの理由付けに伴うラベル計算の並列アルゴリズムを提案しており，各前件データのラベルの要素数の相乗積オーダのプロセスが生成されてしまう可能性があると考えられる．また，後者では，各ATMSノードをプロセスで表現し，複数の理由付けの集まりに伴うラベル計算タスクを1プロセッサに割り当てている．これによって，16プロセッサのときに4倍の高速化がなされている．しかしながら，事実と矛盾する環境の集合を保存している *Nogood* データ・ベースがネットワークの端にあるプロセス構成なので，ラベルの無矛盾性を維持する計算に時間がかかるように思われる．

[Rothberg 89] および [奥乃 90] は，共有メモリ型並列マシン上でインプリメントされたATMSについて述べている．[Rothberg 89] のATMSでは，14PE上での8クィーン問題解決において約6.5倍の高速化が観測されている．また，[奥乃 90] のATMSでは，4PEで3．5倍の高速化が観測されている．しかしながら，共有メモリ型マルチ・プロセッサでは，PE数をあまり増やせない小規模並列のアーキテクチャなので，それ以上の高速化はあまり期待できないと考えられる．

6 まとめ

並列ATMSの特徴は，ノードをプロセスによって表すことで，その生成数を現実的な数に押えることができる点にあり，同時にATMSの処理の基本となるラベル更新の処理をそれぞれのノードが独自に行なうことで，ノードを一括管理する方法に比べて並列化の効果を期待できることにある．しかしそのためには，一つのノードへ負荷が集中することを避けるような理由付けの戦略が不可欠となる．

また，ATMSは推論エンジンと連結することで，効率的な仮説推論システムを構築できることが分かっているが [太田 91]，本並列ATMSも，適当な推論エンジンとの連結によって仮説推論システムを構築することができる．この時，推論エンジンとの連結で注目することは，大量の理由付けが連続してATMSに入力されることが予想されることである．したがって，推論エンジンの側では，探索空間の枝刈りのために，少なくとも各理由付けの後件のラベルに無矛盾な環境があるかないかを高速に返す機能が要求される．開発された並列ATMSは複数の理由付けに伴うラベル計算を並列に計算する機能を備えたもので，仮説推論システムのコンポーネントとして用いる上で有利な特徴を有しているといえる．

今後の課題としては，使用できるプロセッサへ順番に割り当てている負荷分散の仕方をノードの依存関係に従ったものにするということが上げられる．

謝辞

本研究に対して貴重なコメントをいただいたICOT第5研究室の長谷川隆三室長並びに第7研究室の市吉伸行氏に謝意を表します．

参考文献

[deKleer 86] deKleer, J. : "An Assumption based TMS", *Artif. Intell.* , Vol. 28, pp. 127–162(1986).

[Dixon 88] Dixon, M. , de Kleer, J. : "Massively Parallel Assumption-based Truth Maintenance", *Proc. of the AAAI-88*, pp. 199-204(1988).

[Harada 90] Harada, T. and Mizoguch, F. : "Pararell Constraint Satisfaction by Paralleling ATMS", *Proc. of the PRICAI*, pp. 462-455(1990).

[Inoue 88] Inoue, K.: "Problem Solving with Hypothetical Reasoning",*Proc.of the International Conference on Fifth Generation Computer Systems*, Vol.3, pp.1275–1281(1988).

[太田 91] 太田好彦, 井上 克巳: "ATMS を用いた前向き仮説推論システムにおける効率的な推論方式", 人工知能学会誌, Vol.6, No.2, pp. 247--259(1991).

[奥乃 90] 奥乃 博: "網: 新しい ATMS の処理系とその共有メモリ型マルチプロセッサ上での並列処理", 人工知能学会誌, Vol. 5, No. 3, pp. 79-88(1990).

[Rothberg 89] Rothberg, E. and Gupta, A. : "Experiences Implementeting a Parallel ATMS on a Shared Memory Multiprocessor", *Proc. of the IJCAI*, pp. 199-205(1989).

[和田 90] 和田 真一: "疎結合型並列マシンにおける ATMS の並列化方式について", 人工知能学会全国大会 (第 4 回) 論文集, pp. 261-264(1990).

付録 : ラベル計算例

理由付けの集合 J を以下とし, 仮説集合の要素を大文字で表す:

$$\begin{array}{rcll} & A & \Rightarrow & p \quad (1) \\ B, & C & \Rightarrow & p \quad (2) \\ A, & C & \Rightarrow & q \quad (3) \\ & D & \Rightarrow & q \quad (4) \\ A, & D & \Rightarrow & \bot \quad (5) \\ p, & q & \Rightarrow & r \quad (6) \end{array}$$

p のラベル L_p は (1),(2) の理由付けより, {{A},{B,C}}(論理的には $A \vee (B \wedge C)$) であり, 同様に q のラベル L_q は (3),(4) の理由付けより, {{A,C},{D}} (論理的には $(A \wedge C) \vee D$) である.

また (5) より矛盾環境は {A, D} である. このとき, (6) の理由付けにしたがって, r のラベルを求める.

1. L_p と L_q の各環境の和集合に関して直積 L' を求める.

$$L' = \{\{A,C\},\{A,D\},\{A,B,C\},\{B,C,D\}\}$$

2. L' から矛盾環境を含包する要素を削除したのち, その極小集合を求める (L_{new}).

矛盾環境が $\{A,D\}$ であり, $\{A,C\} \subset \{A,B,C\}$ であるから,

$$L_{new} = \{\{A,C\},\{B,C,D\}\}$$

となる.