プログラミング-言語・基礎・実践- 21－6
(1995. 3. 9)

コネクションマシン上の並列論理型言語処理系

古川英司　田中二郎

筑波大学

茨城県つくば市天王台 1-1-1

0298(53)5165

age@softlab.is.tsukuba.ac.jp　jiro@is.tsukuba.ac.jp

あらまし

本研究では、並列論理型言語を、コネクションマシン CM-5 上のデータ並列言語 C* で実装する方法を提案する。並列論理型言語の複数のゴールを同時に書き換える処理と、複数のデータを同時に扱う処理に着目し、実際に実装を行ない、その処理系の評価を行なった。

キーワード　並列論理型言語, GHC, CM-5, C*

Parallel Programming System on Connection Machine CM-5

Eiji FURUKAWA　Jiro TANAKA

University of Tsukuba

1-1-1 Tennoudai, Tsukuba-shi, Ibaraki-ken, JAPAN

0298(53)5165

age@softlab.is.tsukuba.ac.jp　jiro@is.tsukuba.ac.jp

Abstract

The new implementation method of parallel logic programming language in C* on CM-5 is shown. Attention has been paid to the two kinds of parallelism, i.e., the parallelism of rewriting goals and that of handling data. The method has actually been implemented and the evaluation results are shown.

key words　Parallel logic programming language, GHC, CM-5, C*

1 はじめに

最近、各プロセッサにメモリが分散し、それぞれのプロセッサが一つの処理単位を構成するような高並列計算機が、商用のレベルになっている。

しかし、そのような計算機上では、言語処理系としてSIMD 的な言語が実装されている場合が多い。SIMD 的な言語でプログラミングを行なう場合、数値計算のような特殊な用途では有効であるが、汎用的なプログラムを記述するには向いていない。一方 MIMD 的な言語を計算機上に直接実装する方法もあるが、実装に要する手間や、他の計算機へのポータビリティを考えると適切な方法とはいえない。

そこで我々は MIMD 的な言語でプログラムを作成しつつ、SIMD 的な言語で実行することによる、汎用的でかつ性能の高いプログラミング手法を考案した。

本研究では、並列論理型言語 GHC[1] のサブセット (MIMD 的な言語) を、高並列計算機であるコネクションマシン CM-5[2] 上のデータ並列言語 C*[3] (SIMD 的な言語) で実装する方法 [4] について述べる。

2 並列論理型言語

並列論理型言語とは、論理型言語 Prolog を並列化したもので、シンタックス的には、すべての定義節はコミットオペレータを持っている。並列論理型言語のプログラムは、定義節の集合として表現され、それぞれの定義節は、ヘッド、ガード (コミットオペレータの前のゴール列)、ボディ (コミットオペレータの後のゴール列) の三つの部分からなる。

並列論理型言語で書かれたプログラムを実行するには、プログラムに対して、あるゴール列 (キュエリ) を与える。キュエリのすべてのゴールは並列に実行 (リダクション) される。リダクションは次のように行なわれる。

1. ゴールがユニフィケーションなどの組込み述語であればすぐ実行される
2. ゴールがユーザ定義述語であれば、プログラムの定義節の中からヘッドのユニフィケーションが可能でガードを満足する定義節が探索され、もとのゴールはそのボディのゴール列に展開される (図 1)
3. 実行されるべきゴール列が空であれば、実行が終了する

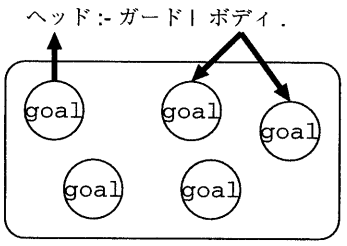

図 1: 実行イメージ

3 コネクションマシンと C*

3.1 C*

C* は、多くのオブジェクト (C* のデータ) に対する演算を並列 (データ並列) に行うため C を拡張した言語であり、おもに以下のような特徴が追加されている。

- 実プロセッサを仮想的に扱うために導入されたshapeによって、並列オブジェクトを容易に扱える
- shape を選択する with、並列オブジェクトを制限して操作する where がある
- 並列オブジェクトを扱うための、演算子・関数 の拡張がある

例えば、プログラム例を図 2の左に、その実行の様子を図 2の右に示す。

```
1: shape [10]S;
2: int:S p, q;
3: with (S) {
4:     p = 1;
5:     q = p * 3;
6: }
```

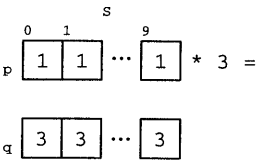

図 2: C* の例

このプログラムでは、1 行目で 10 個の仮想プロセッサを用意し、2 行目で各々の仮想プロセッサ上に int p, q という変数を定義している。つまりここで、shape []S の型がint の p, q という並列オブジェクト (並列変数) を定義している。3 行目で各々の仮想プロセッサがアクティブになり、4 行目で各々の仮想プロセッサ上の p に 1 を代入し、5 行目で各々の仮想プロセッサ上の p を 3 倍し

てから q に代入している。図2の右は、5行目の実行の過程を表している。

3.2 CM-5

コネクションマシン CM-5 は、多くのプロセッシングノード (PN) からなり、それぞれの PN は、データネットワークとコントロールネットワークで接続されている。一つの PN には、一つの RISC プロセッサと四つのベクトルユニット (VU) があり、メモリはそれぞれのベクトルユニットに接続されている (VU が無い構成の場合メモリは一つのメモリコントローラを介して接続されている)。また計算機資源をパーティションと呼ばれるものに分けることができ、それぞれのパーティションは他のパーティションと独立して動かせる。

本研究では、RWCP の CM-5 を用いて実験を行なっている。この CM-5 は 64 PN で、一つの VU に 8Mbytes のメモリがある。それが、32 PN づつの二つのパーティションで分けられていて、実験はその一つのパーティション (32PN つまり 128 VU) で行なっている。

4 並列論理型言語の C* による実装

本実装では、二つの並列性に着目した。一つは並列論理型言語の基本部分である **複数のゴールを同時に書き換える** 処理の並列性、もう一つは並列論理型言語の **複数のデータを同時に扱う** 処理の並列性である。前者は、`shape []Goals` によって並列性を引き出すので **Goals による並列性**、後者は、`shape []Terms` によって並列性を引き出すので **Terms による並列性** と呼ぶことにする。

Goals による並列性は、一つのゴールを、一つの仮想プロセッサ、つまり前に述べた `shape []Goals` の一つの要素に割り当てることで、`shape []Goals` の中のそれぞれの要素は同時に実行され、つまり複数のゴールを同時に処理できるようになる。

しかし、この同時に処理できるとは、(データ並列のため) 同じ処理を同時に実行できるだけで、異なる処理は同時に実行することができない。そこで、一つのゴールが行なう処理を、いくつかのフェーズに分解し、整理することにより、同じ処理を行ないやすくする、という方法をとっている。

一方 Terms による並列性は、一つのデータを、一つの仮想プロセッサ、つまり `shape []Terms` の一つの要素に割り当てることで、それぞれのデータに同時にアクセスできるようにすることで、複数のゴールの同時処理を支援する。

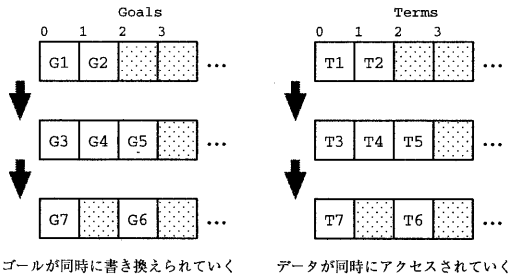

図 3: 並列性のイメージ

我々は、この二つの並列性を組み合わせることにより、並列論理型言語の C* による実装を行なった。この章では、その実装方法について具体的に説明する。

4.1 データ構造

並列論理型言語で使用されるデータは、C* のオブジェクトで表現されなければならないが、上で述べたような処理を実現するために、並列論理型言語のデータには

- 複数のゴールから同時に別々のデータをアクセスする
- 条件を満たすゴールの選択を、同時に行なう
- 複数のゴールの書き換えを、同時に行なう

といった機能が必要である。しかし、C* には、いくつかの表現上の制限があり、そのおもなものとして、

- ポインタ変数を要素に持つような構造体は、shape を使って並列オブジェクトにできない
- 並列変数は、構造体の要素になれない
- 並列でない配列の添字に、並列オブジェクト (この場合、オブジェクトは自然数) をもってくることができない。

がある。

このため、本実装の並列論理型言語のデータの構造を、C* のオブジェクトで表現するために、以下のように決めている。

- データは、タグとデータ本体の (Terms による) 並列構造体にする
- データへのポインタは、そのデータが並列構造体の何番めの位置にあるか、その数字 (仮想プロセッサの座標) で表す

一つのデータは構造体で、その要素としてタグとデータ本体からなる。タグは、データの種類を表す識別子で、データの種類としてVAR(論理変数), COMP(構造データ), INT(整数) 等がある。データ本体は、論理変数や構造データや整数等の共用体として表現される。そしてそのデータの集まりは、並列構造体で表現され、それぞれのデータに対して同時にアクセスすることが可能になる。

例えば、次のようなデータがあったとする。

```
add(100,200,X)
```

これは、構造データで、ファンクタが add、アリティが 3、引数はそれぞれ 100, 200, X でX は未定義の論理変数である。データの構造のイメージは図 4 のようになる。

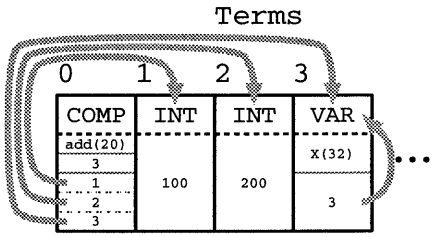

図 4: データの構造のイメージ (1)

この中の add(20) や X(32) は、アトム (文字列) とそのアトムのアドレス (文字列の位置を表す自然数) である。add のアドレスは 20 であることを表している。C* では、文字列はポインタで表現されることになっているが、「ポインタ変数を要素に持つような構造体は、並列オブジェクトにできない」という C* の制限があるため、いったんポインタを "アトムのアドレス" という自然数に変換してから、データを格納している。

以下説明のために、add(100,200,X) というデータの構造のイメージは図 5 のように簡略化することにする。

ところで、この並列構造体は、この処理系ではヒープの役割を果たしている。通常の逐次計算機のヒープと異なるところとして、そのメモリー空間が VU 上にあるというところである。VU 上のメモリを、C* によって仮想的に扱うことができる。

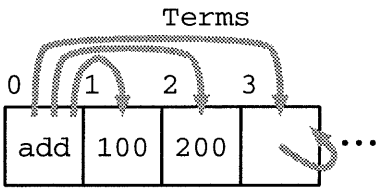

図 5: データの構造のイメージ (2)

プログラム (定義節の集まり) の内部表現は、ゴールの書き換えを同時に行なえるようにするために、データと同様に工夫する必要がある。複数のゴールから条件を満たすゴールの選択 (コミット) を、各々のゴールにおいて同時に行ないたいからである。そこで、以下のようにする。

- プログラムは、定義節の (Goals による) 並列配列、つまりshape []Goals を使って定義節の配列を並列オブジェクトで表す。その並列配列上のそれぞれの仮想プロセッサは、すべて同じ値の配列を持つようにする。つまりこの並列配列において、この shape の一つ一つの要素である配列が、全て同じ値を持つようにする。一つの定義節は構造体で表す。
- ゴールの選択に並列性を持たせるため、プログラムとは別に、述語の並列構造体を用意する。ある一つの述語の (アトムの) アドレス (自然数で表現されている) の値を仮想プロセッサの座標としてみて、その座標の仮想プロセッサ中の構造体により、その述語が (上の) プログラムの配列のどこから始まり、いくつ並んでいるかを表すことができるようにする (同じ述語の定義節はプログラムの配列上でとなり合うように配置されている)。

プログラムを表す並列配列は、基本的には普通の配列であるが、それでは複数のゴールからの同時アクセスができなくなるので、同じ内容の配列を並列に用意している。

プログラムを表す並列配列があるだけでは、ある述語の定義節が、プログラム中のどこにいくつあるのかは、すぐにはわからない。そこで、それを調べるためのテーブルが述語の並列構造体である。この並列構造体の中から、調べたい述語のアドレスを仮想プロセッサの座標とする位置にある、構造体を調べることにより、その述語がプログラム

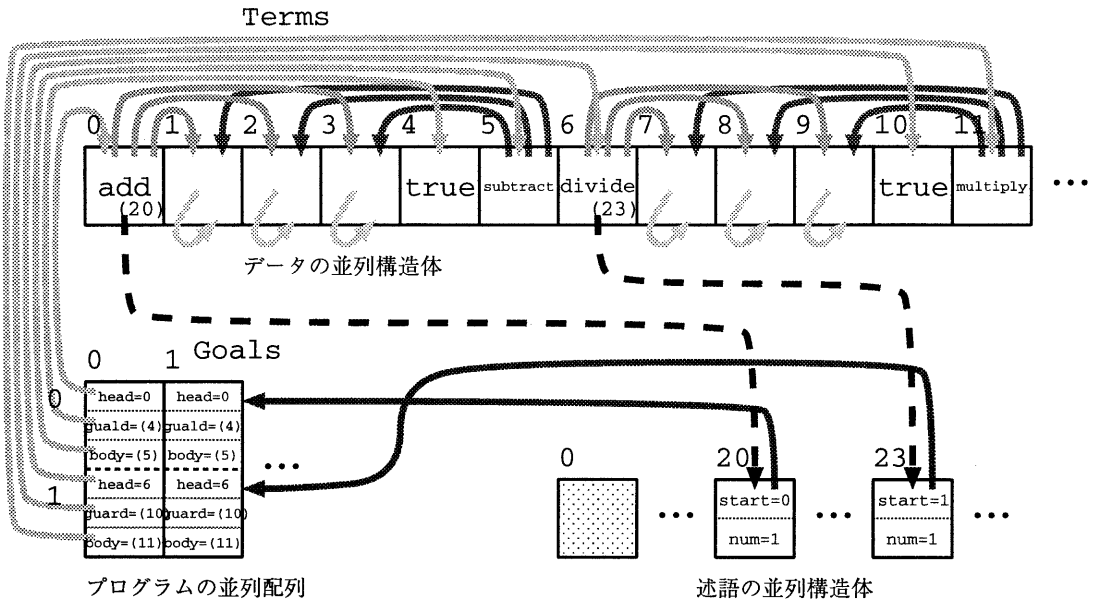

図 6: データ・プログラム・述語のイメージ

の並列配列のどこから始まって、いくつ並んでいるかがわかる。

例えば、以下のプログラムで、

```
add(X,Y,Z)    :- true | subtract(Z,Y,X).
divide(X,Y,Z) :- true | multiply(Z,Y,X).
```

プログラムの並列配列・述語の並列構造体のイメージを図 6 に示す。

図 6 のプログラムの並列配列は、縦軸が配列の軸、横軸が shape []Goals の軸である。この図からもわかるように、データとプログラムと述語は関連性を持っている。

4.2 実行機構

本処理系のメインは、以下のように進められる。

```
with (Goals) {
  初期ゴールの処理;

  while (アクティブなゴールがあるか)
    where (アクティブなゴールとサスペンドしているゴール)
      where (ゴールが組み込み述語か)
        組み込み述語の処理;
      else { /* ユーザ定義述語 */
        コミットできる節があるか調べ;
        where (コミットできる節があれば)
          ゴールの展開;
        else
          コミットできないゴールをいったんサスペンドする;
      }

  /* もうアクティブなゴールはない */
  if (サスペンドしているゴールが一つもない)
    success;
  else
    deadlock;
}
```

一つのゴールが行なう処理は、おもに次の三つのフェーズに分解することができる。

- (組み込み述語の場合) 組み込み述語の処理
- (ユーザ定義述語の場合) コミットできる節があるかどうかを調べる
- (ユーザ定義述語の場合) ゴールの展開

複数のゴールが同じフェーズに属している場合、それらのゴールは同時に実行できるようにする。

4.3 ユーザ定義述語

ゴールがユーザ定義述語の場合、コミットできる節があるかどうかを調べる処理と、(コミットした節のボディの)

ゴールを展開する処理とに分けることができる。

コミットは次のような処理になる。

```
with (Goals) {
  それぞれの述語が、述語の配列からプログラムの配列中のどこにあるかを調べる;
  result = NOCOMMIT;
  for (;;) {
    if (すべてのゴールでもうコミットを試すプログラムが無い) break;
    where (まだコミットを試すプログラムがある) {
      result = commit();
      where (result != NOCOMMIT)
        もうコミットを試さない;
    }
  }
}
```

まず、それぞれのゴールの述語が、述語の配列をインデックスとして、プログラムの配列中のどこの位置にいくつ並んでいるかを調べる。次に、それぞれのゴールにおいて、コミットが成功するまで繰り返す。

コミットが成功したゴールにおいて、そのボディゴールの展開を行なう場合、使用中の仮想プロセッサに割り当てることは、当然できない。したがって、仮想プロセッサに、ゴールを割り当てることができるかどうかは、調べる必要がある。本実装では次のようにしてゴールを割り当てる。

```
with (Goals) {
  /* ボディの一つめ */
  ボディゴールの一つめを、現在のゴールに割り当てる;

  /* ボディの二つめ以降 */
  ボディゴールの二つめ以降の数を数える;
  それをすべてのゴールで合計し、その分だけ空いている仮想プロセッサを予約する;
  それをそれぞれのゴールに分配する;
  for (;;) {
    if (すべてのゴールでボディゴールの展開がおわった) break;
    where (まだボディゴールの展開がおわっていないゴール)
      ボディゴールの二つめ以降を順に、それぞれのゴールに分配されている空き仮想プロセッサに割り当てる;
  }
}
```

まず、ボディゴールの一つめを、現在のゴールに割り当てることにする。これは、次に行なわれる空き仮想プロセッサを調べる処理が軽くないためであり、その処理を少しだけ軽くするためである。

次に、ボディゴールの二つめ以降の割り当てを行なう。しかし、それぞれのゴールで直接割り当てを行なおうとすると、空き仮想プロセッサが奪い合われることになり、処理が大変になる。そこで、空き仮想プロセッサは一括して確保し、後でそれぞれのゴールに分配し、それからそれぞれのゴールでボディゴールの割り当てを行なうことにする。

例を示す。

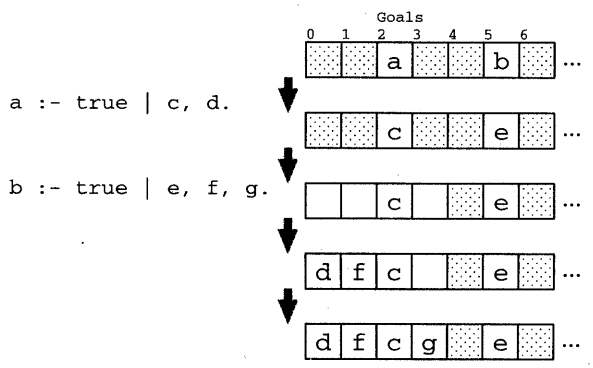

図 7: ゴールを書き換える様子

図7は、本処理系でゴールの書き換えの実行の過程を示している。左の図がその時与えられている定義節で、右の図が書き換えの様子である。はじめに、ゴール列にaとbがあり、まずそれぞれの定義節のボディの一つめのゴール(それぞれcとe)が元のゴールと置き換わる。次にボディの二つめ以降のゴールを数え合計し(この場合3)、その分だけまだゴールが割り当てられていないところからゴールを確保する(この場合、Goalsの0,1,3の場所)。次に確保されたゴールの位置に、ボディの二つめ以降のゴールを順に割り当てる。そして、ゴールの書き換えの処理を終了する。

このような処理を行なうことにより、複数のゴールの同時書き換えを行なっている。

4.4 組み込み述語

組み込み述語の数は少なくない。それぞれの組み込み述語は、一つ一つ処理が異なるので、直接的に実行すると、データ並列で実行できないため、一つ一つの述語を逐次的に実行しなければならず、あまり効率はよくない。本実装では、いくつかの異なる述語を同時に処理を行なうようにした。

例えば、四則演算を行なう場合、加算を行なう処理、減算を行なう処理、乗算を行なう処理、除算を行なう処理と、値を加減乗除するところだけが異なる処理であるが、組み込み述語として直接実装する場合、それぞれ別にしなければならない。

```
with (Goals)
  where (f == add)
    t3 = t1 + t2;
  else where (f == subtract)
    t3 = t1 - t2;
  else where (f == multiply)
    t3 = t1 * t2;
  else where (f == divide)
    t3 = t1 / t2;
```

これでは、同じ処理を行なうゴールは並列に実行できるが、異なる処理を行なうゴールは逐次にしか実行できない。また、where 文を多用するため、多くのコンテキストスイッチが発生し、大変効率が悪い。そこで、新しく「四則演算を行なう処理」を設定し、その中で、

```
with (Goals)
    t3 = (f == add) * (t1 + t2)
       | (f == subtract) * (t1 - t2)
       | (f == multiply) * (t1 * t2)
       | (f == divide) * (t1 / t2);
```

とするようにする。これにより、四則演算を行なう全てのゴールが同時にデータ並列で実行できるようになる。

これにより、組み込み述語の処理部分の実行時間の短縮になり、またソースコードが小さくなったことによるコンパイル時間の短縮ができる。

4.5 ユニフィケーション

組み込み述語の代表的な処理としてユニフィケーション(ボディ・ユニフィケーション)がある。次のような処理になる。

```
with (Goals) {
  変数をデレファレンスする;
  where (構造データ・変数以外で、それぞれのデータが等しい)
      success;
  else where (構造データで、ファンクタとアリティがそれぞれ等しい) {
    引数をそれぞれユニフィケーションする;
    where (全て成功すれば)
      success;
    else
      failure;
  }
  else where (どちらかが変数) {
    その変数をもう一方のデータをレファレンスするようにする;
    success;
  }
  else
    failure;
}
```

ユニフィケーションの基本的な処理は、

1. まず変数をデレファレンスして、
2. 構造データ／変数 以外のデータならば、両者が同じかどうか比較し、
3. 構造データならば、再帰的にユニフィケーション
4. 一方のデータが変数ならば、その変数をもう一方のデータをレファレンスするようにする

というようになる。ここでユニフィケーションされるデータは(Terms の)並列変数なので、複数のデータを同時にユニフィケーションできる。

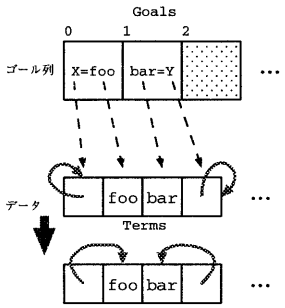

図 8: ユニフィケーションの実行

図 8は、ゴール列に、X=foo とbar=Y というゴールがある場合の実行の過程を示している。ここで X と Y は未定義の論理変数である。この時、上のアルゴリズムを適用すると、二つのゴールのユニフィケーション同時に実行され、完了する。

ガード・ユニフィケーション(ヘッドのユニフィケーションを含む)は、上で述べたボディ・ユニフィケーションとほぼ同じであるが、ガード側の変数のみ値を決めることができ、もとのゴール側の変数は値を決めることができない、という点が異なっている。

5 評価

評価は、二種類のプログラムで行なった。一つ目は、同時に解くゴールの数を増やした時の実行時間を調べることで、二つ目は、Goals と Terms を変化させてみた時の実行時間を調べることである。以下 Goals を 4096、Terms を 1048576 にして実験した。

まず、8 要素のリストと空リストとの連結をいくつか同時に実行した場合 (append) と、8 要素の乱数列のソーティングについて、いくつか同時に実行した場合 (qsort)。

	同時に実行するゴール数				
	1	2	4	8	16(同時)
述語 append	2.21	2.25	2.35	2.51	2.53(秒)
述語 qsort	5.35	5.44	5.56	6.10	6.81

表 1: ベンチマーク・プログラムの実行時間 (1)

表 1 より、append, qsort ともに、1 つ同時に実行した場合と 16 同時に実行した場合とで、あまり実行時間に変化が無いことがわかる。このことから、ゴールが複数あっても Goals の並列効果により実行時間はあまり変化しないことがわかる。

次に、5 queens について Goals と Terms を変化させてみた場合。

	Terms PN	Goals PN	
Goalsを固定	32	32	28.46 (秒)
Goalsを固定	16	32	28.97
Goalsを固定	8	32	29.59
Goalsを固定	4	32	29.42
Goalsを固定	2	32	31.50
Termsを固定	32	16	37.26
Termsを固定	32	8	59.69
Termsを固定	32	4	91.72

表 2: ベンチマーク・プログラムの実行時間 (2)

表 2 は、Terms と Goals を C* のライブラリ関数 `allocate_detailed_shape()` で設定し、実質的に使われる PN 変化させた時の結果である。。この表から、Terms に関しては全く台数効果はなく、Goals に関しては約 1.5 倍の台数効果があることがわかる。Goals による並列性は、単調に実行時間を減少させるので、かなり性質のよいものだと思われる。Terms による並列性は、実行時間に影響を与えないが、並列性による メモリ空間の増大や C* による記述力の向上を考慮すると、性質の悪いものではないと思われる。

6 関連研究の動向

次のような研究がある。

コンディショングラフによる KL1[5] Barklund らが、コネクションマシン上にコンディショングラフを実装し、KL1 をその上にマッピングする方法を提案

Fleng による GHC[6] Nilsson が、GHC を中間言語である Fleng にコンパイルした後、それを SIMD 計算機上で実装する方法を提案

本研究は、C* を用いて、ユーザプログラミング言語のレベルから実装を行なっており、それが違いである。

7 おわりに

本研究では、並列論理型言語を C* で実現する方法について述べた。Goals による並列性は、評価結果からかなり有効であることがわかる。Terms による並列性は、実行速度の面については優位性は無いが、C* による簡便な記述で PN(VU) 上のメモリを扱うことができるので便利である。今後は、本方式の効率化や、それに伴う言語の拡張を行なっていく予定である。

参考文献

[1] K. Ueda: Guarded Horn Clauses, ICOT Technical Report TR-103, ICOT, 1985.

[2] Thinking Machines Corporation: *CM-5 Technical Summary*, November 1993.

[3] Thinking Machines Corporation: *C* Programming Guide*, May 1993.

[4] 古川英司, 田中二郎: 並列論理型言語のデータ並列言語による実装, 情報処理学会第 49 回全国大会, 1994.

[5] Jonas Barklund, Nils Hagner, and Malik Wafin: KL1 in Condition Graphs on a Connection Machine, *The International Conference on Fifth Generation Computer Systems 1988*, 1988.

[6] Martin Nilsson: *Parallel Logic Programming for SIMD Supercomputers and Massively Parallel Computers*, PhD thesis, The University of Tokyo, 1988.