4P-5

並行プロセス記述言語NCCの改良

堀口 雅人、朴 泰佑、木村 哲郎、天野 英晴

慶応義塾大学理工学部

1. はじめに

本大学で開発された科学技術計算用MIMD型並列計算機$(SM)^2$-II[1]上に実装されている並行プロセス記述言語NCC(Node oriented Concurrent C)[2]の使用経験をもとに、この言語に改良を加えて記述性、移植性の向上を図り、言語としての充実度を増す。NCCは、NCモデル[3]と呼ばれる計算モデルに基づいており、全体のシンタックスはC言語をベースとして設計されている。$(SM)^2$-II上には、DIPROS[4]と呼ばれるOSが実装されており、NCCはこのOS上で実行される。$(SM)^2$-IIでは、PU間通信にRSM (Receiver Selectable Multicast) と呼ばれる、1つのPUから複数のPUへ同時にデータを転送する機能を用いている。これによって、プロセス間通信において、データのマルチキャストが可能となる。

2. NCモデル

解析すべき問題は、ノードとコネクティングラインにより構成される静的なネットワークで記述される。ノードは、通常のプロセスに対応し、システムの実行単位である。コネクティングラインは、チャネルに対応するノード間の関係を表す単方向通信路である。

- 各ノードは、入出力用のポートを持つ。それぞれのポート間は、コネクティングラインで結合される。ノードは、逐次処理を行い、ポートを介して外部と交信を行う。
- ネットワークは計算の開始から終了まで変化しない(静的である)。
- 一対多のマルチキャストによるデータ転送をサポート(一つの出力ポートを複数の入力ポートと接続できる)。
- 各ノードでは、データの到着順に処理を行うことができる(データ駆動型の処理が可能である)。

3. NCCの問題点及びその解決法

新バージョンのNCCでは、以下のような文法の改訂及び機能の追加を行うことで、記述性の向上と処理系の高速化を図る。

3.1 関数のクラスの設定

NCCでは、プロセスのスタート関数や、複数プロセスに共有される関数等を明示していないので、プログラムをわかりにくくしたり、バグの原因になる可能性があった。そこで**関数のクラス**を導入して、これを明示的に宣言する。関数のクラスとしては、以下の3つを考える。

1. **スタート関数**：プロセスが最初に実行する関数。
2. **局所関数**：1つのプロセスタイプからのみ呼ばれる関数。
3. **共有関数**：複数のプロセスタイプから呼ばれる関数。

この3つのクラスは排他的である。関数の定義を行う際に、その関数のクラスを明示的に宣言する。そのために、予約語として**main**、**private**、**shared**を導入する。

スタート関数	**main**	StartFunction()
局所関数	**private**	PrivateFunction()
共有関数	**shared**	SharedFunction()

ただし局所関数のみは、その宣言を省略することができる。

3.2 マルチ・レシーブ・エントリ構文の改訂

NCCでは、データ駆動型の処理を行うためにエントリ構文が用意されていた。エントリ構文で各受信エントリの実行後に他の受信エントリを受信するかどうか制御することが出来た。実際の処理を考えた場合、全エントリ実行か単一エントリ実行かのどちらかに限定することで解析性が上がる。NCCでは、ポート配列などのサフィックスの管理を機械的に行っていたために余分な処理をする必要があり、これがオーバーヘッドになっていた。そこで、この管理をユーザがすることで、オーバーヘッドが軽減され、処理系の高速化が見込める。

The new version of concurrent process description language NCC

Masato HORIGUCHI, Taisuke BOKU, Tetsuro KIMURA, Hideharu AMANO

Keio University

3.3 マッピング指定記述部の導入

NCCでは、プロセスのマッピングに関する情報をプログラム中に記述することができなかった。マッパが交信量やプロセス負荷によって自動的にマッピングするのには、限界がある。そこで、マッピング指定記述部を設けて、プログラマが明示的にプロセス間の関係を記述し、マッパが効率よくマッピングできるように情報を提供する。プログラマは、プロセス間の関係を記述するためのグループを定義する。このグループをドメインという。ドメインは、プロセスの論理的な集合であり、マッパは、与えられたドメイン情報を反映する形で、物理的なプロセスグループ、すなわち各PUへのマッピングを決定する。

4. プログラムの例

例として、一次元ポアソン方程式を離散化して解く（図1、図2参照）。プロセス宣言部で、プロセスタイプを宣言し、connect文でプロセス間の結合を行う。mapping文でマッパへのドメイン情報を記述している。pieceProg()関数、boundProg()関数が、スタート関数となっている。pieceProg()中のdoeach文でデータ駆動型の処理を行っている。doeachは、全エントリを受信する構文である。ここでは、両隣りからデータを受け取り、sendで送っている。

5. おわりに

コンパイラの実装は、YACC/LEXを用いて行う。このように処理系を系統的に作成することで、デバッグのための的確な情報を提供できるようになり、また、将来へ向けての機能の拡張性と移植性が向上する。以上のような改良を加えることで、NCCの記述性が向上し、処理系の高速化が見込まれ、言語としての完成度が増す。

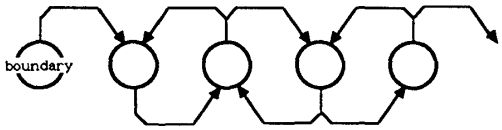
[参考文献]

[1] H. Amano, et al. ,"*(SM)²-II: The new version of the Sparse Matrix Solving Machine*," Proc. of 12th Ann. ISCA, June 1985

[2] 高橋範朗、他：「並列計算機$(SM)^2$-IIのための並行プロセス記述言語NCC」情報処理学会第32会全国大会

[3] T. Kudoh, et al. ,"*NDL : A language for solving scientific problems on MIMD machines*," Proc. of 1st Int. Conf. on Super Computing Systems, Dec. 1985

[4] T. Boku, et al. ,"*DIPROS : a distributed processing system for NDL on $(SM)^2$-II*," Proc. of 20th Hawaii Int. Conf. on Systetm Sciences, Jan. 1987



$$u_i^{n+1} = \frac{1}{2}(u_{i-1}^n + u_{i+1}^n)$$

図1：一次元ポアソン方程式の解法

```
#define         P_NUM           64
#define         BOUND           1.0
#define         INITVAL         0.0
#define         STEP            100
#define         D_NUM           16

process PieceNode {
    inport      left , right ;
    outport     out ;
    main        pieceProg() ;
} piece[P_NUM]() ;

process BoundNode {
    outport     out ;
    main        boundProg() ;
} bound[2]( BOUND ) ;

connect {
    int     i ;

    for ( i = 0 ; i < P_NUM ; i ++ ) {
        if ( i != 0 )
          con( piece[i].out , piece[i-1].right ) ;
        if ( i != P_NUM - 1 )
          con( piece[i].out , piece[i+1].left ) ;
    }
    con( bound[0].out , piece[0].left ) ;
    con( bound[1].out , piece[P_NUM-1].right ) ;
}

mapping {
    int     i ;
    domain d[D_NUM] ;

    for ( i = 0 ; i < P_NUM ; i ++ )
        member( piece[i] , d[i/P_NUM] ) ;

    member( bound[0] , d[0] ) ;
    member( bound[1] , d[D_NUM-1] ) ;
}

main    pieceProg()
{
    int     i ;
    double temp , u = INITVAL ;

    send( out , u ) ;
    for ( i = 0 ; i < STEP ; i ++ ) {
        u = 0 ;
        doeach {
            ( right , &temp ):
            ( left , &temp ):
                u += temp ;
        }
        send( out , u / 2 ) ;
    }
}

main    boundProg( data )
double  data ;
{
    while ( 1 ) send( out , data ) ;
}
```

図2：ポアソン方程式を解くプログラム