プログラミング－言語・基礎・実践－ 13－2
（１９９３．８．１９）

Bee-Fortran のマルチターゲット・コンパイル方式

渦原 茂‡　小前 晋†　杉森 英夫†　大谷 浩司*　安村 通晃‡
‡ 慶應義塾大学　† 住友金属工業株式会社　* 有限会社アックス
連絡先:〒 252 藤沢市遠藤 5322 慶應義塾大学環境情報学部安村研究室

本稿では、複数のアーキテクチャ、特に SIMD 型と MIMD 型の分散メモリ並列計算機に対して、共通のアプローチでコンパイルすることによってポータブルな Bee-Fortran コンパイラシステムの構築を試みる。現在の Bee-Fortran コンパイラは SIMD マシンをターゲットとして稼働している。Bee-Fortran コンパイラの設計においては、マシンに依存する部分としない部分に分け、マシンに依存しない部分に抽象的な並列マシンを設けてそのレベルで並列に関する最適化を行なう。本稿ではこれを Single Program Multiple Data(SPMD) 方式に基づいた MIMD マシンのコンパイラへと拡張する。またその際の SIMD と MIMD の相違点についても述べる。

Multi-Targetable Compilation Strategy of Bee-Fortran

Shigeru Uzuhara‡　Susumu Komae†　Hideo Sugimori†
Koji Ohtani*　Michiaki Yasumura‡
‡ Keio University　† Sumitomo Metal Industries,Ltd.　* Axe, Inc. .
Address:Keio University Yasumura Lab. 5322 Endo Fujisawa shi.

This paper presents the strategy of a Bee-Fortran compiler for various parallel architectures. We focus on SIMD and MIMD distributed-memory machines, and attempts a prototype design using a common strategy on both architectures. The current version of Bee-Fortran compiler has been implemented on a SIMD machine. The compilation process is separated into two phases: machine-independent phase and machine-dependent phase. Optimization for parallel execution is performed in the former phase, on basis of the abstraction of parallel arhcitectures. This paper describes a natural extension on Bee-Fortran compiler for generating MIMD code, using Single Program Multiple Data (SPMD) model. The difference between compilations on SIMD and MIMD is also presented.

1 はじめに

現在我々は超並列計算機用プログラミング言語Bee-Fortran[10]を開発している。Bee-FortranはFortran 90をベースにした並列プログラミング言語であり、またHigh Performance Fortran(HPF)[6]に準拠している。Bee-Fortranの目標の1つはアーキテクチャに依存しないプログラミング環境を提供することであり、そのためにはコンパイラによってアーキテクチャ間の違いを吸収することが重要となる。

本稿では、複数のアーキテクチャ、特にSIMD型とMIMD型の分散メモリ並列計算機に対して共通のアプローチでコンパイルすることによってポータブルなBee-Fortranコンパイラシステムの構築を試みる。Bee-Fortranコンパイラはその言語仕様に合ったSIMDモデルに基づいて設計されており、最初のターゲットもSIMDマシンであった。しかし設計にあたってマシンに依存する部分としない部分に分け、マシンに依存しない部分では抽象的な並列マシン[10]を設けてそのレベルで並列に関する最適化を行ない、具体的なマシンコードを生成する部分と分離した。本研究ではこれをSingle Program Multiple Data (SPMD)方式[2]に基づいてMIMDマシンのコンパイラへと拡張する。またその際のSIMDとMIMDの相違点についても述べる。

以下では、2節においてBee-Fortran言語について、3節においてBee-Fortranの最初のターゲットであるSIMDマシンSM-1についてそれぞれ紹介する。4節ではSIMDマシンのコード生成について説明し、5節ではSIMDからMIMDマシンのコード生成を行なうようにコンパイラを拡張する。6節では、SPMDコードとSIMDのコードの相違点をあげる。最後に7節において関連する研究との比較を行なう。

2 Bee-Fortran

Bee-Fortranは Fortran 90をベースにした並列プログラミング言語である。またHigh Performace Fortran(HPF)に準拠しており、コンパイラディレクティブによってプログラマが配列データをどのプロセッサに割り付けるかを指示することができる。図1(a)は、Bee-Fortranのプログラム例である。加算を行っている部分が配列計算の代入文で、図1 (b)はそれに相当するDOループである。!HPF$がついている行にはHPFのディレクティブが書かれている。PROCESSORSは2x2のプロセッサの集合(プロセッサ配列)を定義している。ただし、ここで定義しているのは抽象的なマシンであり、実際のマシンが2x2のメッシュ状に構成されている必要はない。DISTRIBUTEは配列Aを A(1:10,1:10),A(11:20,1:10),A(1:10,11:20), A(11:20,11:20)の4つに分割し、それぞれを4つのプロセッサのメモリに配置することをコンパイラに指示している。これをBLOCK分割という。これに対して、配列の次元をより細かいブロックに分割して、それらに順々にプロセッサを割り当てていき、プロセッサがなくなったら、また最初からプロセッサを割り当てていくというCYCLIC分割がある。ここでのALIGNの指定は、例えば、配列Bの要素B(i,j)が配列Aの要素A(i,j)のプロセッサと同じになるように、配置を指示している。

```
      REAL A(20,20), B(20,20), C(20,20)
!HPF$ PROCESSORS P(2,2)
!HPF$ DISTRIBUTE A(BLOCK,BLOCK) ONTO P
!HPF$ ALIGN (I,J) WITH A(I,J) :: B,C

      A(1:19,1:20)=B(2:20,1:20)+C(1:19,1:20)
```

(a)

```
      DO I2 = 1, 20
         DO I1 = 1, 19
            A(I1,I2)=B(I1+1,I2)+C(I1,I2)
```

(b)

図 1: Bee-Fortranのプログラム例

3 SIMDマシンSM-1

Bee-Fortranコンパイラが最初にターゲットにしたSIMD並列計算機SM-1[9]とそのプログラミング言語、並C[8]について説明する。

SM-1 SM-1は1024台のプロセッサを32x32の2次元メッシュ状に結合した分散メモリ型並列計算機である。SM-1システムは汎用のワークステーションをフロントエンドにして、そのコプロセッサという形で構成される。SM-1プロセッサの命令はフロントエンド側に置かれ、フロントエンドプ

ロセッサがSM-1の命令をフェッチするとそれがSM-1側に送られ、SM-1のシーケンサを通して全プロセッサが同時にその命令を実行する。プロセッサごとに命令を実行するかどうかはプロセッサごとにもつアクティビティと呼ばれるフラグによって制御することができる。これを模式的に表したのが図2である。矢印は命令流を表し、実線が実行した場合、破線が実行しなかった場合である。命令を実行するしないにかかわらず、同期して処理が進む。

並C 並Cは C言語をベースにした並列プログラミング言語である。Bee-Fortran コンパイラはSM-1に対するオブジェクトコードとして並Cのコードを生成する。並Cコンパイラは、フロントエンドプロセッサの命令とSM-1の命令を生成する。図3は並Cのプログラム例である。`double $`はその変数をSM-1の各プロセッサのメモリに割り付けることを宣言している(パラレルクラスの変数という)。この変数をもつ式はSM-1プロセッサ側で実行される。`pif`はCの`if`に相当するもので、条件部が真のSM-1プロセッサだけがその本体を実行する。一方、`for`はフロントエンド側で実行される。SM-1の説明で述べたように、`pif`の条件部が偽のプロセッサは、真のプロセッサが本体の実行を終了するまで待ち、`for`の次のイタレーションに進んだりはしない。並Cはマシンに依存した言語であり、パラレルクラスの変数などの実体の数は実プロセッサ数と同じになっている。

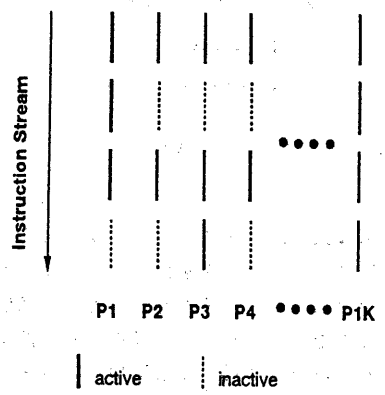

図 2: SM-1の実行モデル

```
double $ x[N], y[N], z[N];

for (i = 0; i < N; i++)
    pif (x[i] > 0)
        z[i] = x[i] - y[i];
```

図 3: 並Cのプログラム例

4 SIMDコードの生成

まずSM-1をターゲットにしたSIMDコードの生成方法を説明する。Bee-Fortran コンパイラは以下のようなフェーズでコード生成を行う。

- 中間コードの生成
- 配列の分割配置
- 計算の分割
- 通信コードの挿入

中間コードの生成 ソースプログラムはコンパイラ内部の中間コードに変換される。中間コードレベルでは、並列実行部分を一種の並列ループとして表現し、このレベルでマシンに依存しない最適化を行なう。図4(b)は(a)のプログラムの中間コードである。do_each_elementが並列ループのヘッダで、end_do_each_elementとで囲まれたループ内部の式がループ変数`i_1`, `i_2`をインデックスとして配列要素を選択し、計算を行なう。`do_each_ape`は抽象的な並列マシンを表している。

配列の分割配置 配列はSM-1のプロセッサに分割して配置される。現在のコンパイラは1次元配列に対してはSM-1のプロセッサアレイを1次元と見なして、2次元配列に対してはSM-1のプロセッサアレイを2次元として見なして割り付ける。3次元以上の配列はそのうちの2次元まで選択することができ、2次元配列の場合と同ようにそれらを割り付ける。

SIMDに関するデータの配置として、フロントエンド側とSM-1側の2つのメモリのどちらにあるかということを意識しなければならない。[1] スカラ変数はフロントエンド側にあり、必要に応じて低コストでSM-1側に放送される。配列は基本的にはSM-1側に配置されるが、フロントエンド

側に配置されるものもある。入出力はフロントエンド側で行なうので一時的に配列をフロントエンド側に移動するといったことを行なう。

計算の分割　配列の配置が決まると、それに合わせて計算を各プロセッサに割り振る。この方法としてBee-Fortranコンパイラでは Owner Comptutes Rule[7] を用いている。Owner Comptutes Rule とは、計算を行なうプロセッサを代入文の左辺の変数の配置に合わせて選ぶというヒューリスティクッスである。つまり、左辺の配列の各要素を計算するのはその要素をもっているプロセッサになる。

各プロセッサには自分が担当する配列要素のうちどの要素を計算すればよいかを識別するために計算しようとしている配列の各要素に対応したマスクを用意する。例えば、図5は1次元目を CYCLIC 分割した配列 A の A(1:N:3,:) の部分を計算する場合のマスクの例である。マスクは配列と同様に32個のプロセッサ (P1–P32) に配置されており、各プロセッサはループにおいてそのマスクが active のときにだけをアクティビティを active にして計算を実行する。[1]

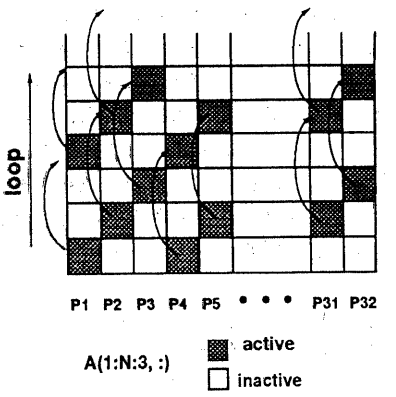

図 5: マスクの例

通信コードの挿入　計算するプロセッサが決まると、次に計算に必要なデータを他のプロセッサが持っている場合は、そのデータをとってくる通信コードが必要となる。これは、計算を始める前に必要なデータをまとめて移動する通信コードとして挿入される。

[1] マスクを使わずにインデックスを用いる方法もあるが、配列のアクセスが密な場合は SM-1 ではこの方が高速に実行できる

```
REAL X(100,100), Y(100,100), Z(100,100)
!HPF$   DISTRIBUTE(CYCLIC,CYCLIC) :: X,Y,Z

        X(1:99,:) = Y(1:99,:) + Z(2:100,:)
```

(a)

```
do_each_ape(L1)
do_each_element(L1, i_1=0:98, i_2=1:99)
    x[i_1][I_2] = y[i_1][i_2] + z[i_1+1][i_2];
end_do_each_element
```

(b)

```
BFMAIN__()
{
    /* some codes dropped .... */

    BF_make_real_PEarray(&r_1, r_1_body, &shape_1);
    SET_SEC_TRIPLET(sec_2[0], 1, 99, 1);
    SET_SEC_TRIPLET(sec_2[1], 1, 100, 1);
    SET_SEC_TRIPLET(sec_1[0], 2, 100, 1);
    SET_SEC_TRIPLET(sec_1[1], 1, 100, 1);
    BF_copy(&r_1, sec_2, &z, sec_1, sizeof(real));
    SET_SEC_TRIPLET(sec_3[0], 1, 99, 1);
    SET_SEC_TRIPLET(sec_3[1], 1, 100, 1);

    BF_make_activity(L_1, x.shape, sec_3);

    for (f_idx = 0; f_idx < 16; f_idx++) {
        pif (L_1[f_idx]) {
            x.body.PEreal[0 + f_idx * (1)] =
                r_1.body.PEreal[0 + f_idx * (1)]
              + y.body.PEreal[0 + f_idx * (1)];
}
    }
}
```

(c)

図 4: SIMD コード

図 4(c) は得られる SM-1 のコード例である。BF_copy が通信コード、BF_make_activity がマスクを作っている部分で、その下の for ループが計算を実行している。

5 SPMD コードの生成

ここでは、Bee-Fortran コンパイラを拡張して MIMD マシンのコードを生成する方法について述べる。生成されたコードの実行モデルには、Single Program Multiple Data(SPMD) 方式 [2] を採用する。SPMD とは、プロセッサごとのおのおのデータに対して同じプログラムを実行する方式である。これはちょうど SIMD の動きを模倣したモデルになっている。

中間コードからの変換 基本的には SPMD のコード生成は SIMD の場合と同様なフェーズで進む。SIMD では常に同期している必要があるが、並列ループを各プロセッサに一度割り振ってしまうと、ループ中は同期をとる必要がないので、SIMD がとるような同期をとる必要はない。図 6は図 4に対応した SPMD コードである。SM-1 コードとの違いは、各プロセッサ (myself) が自分がループのどこを回るのかを求めているところである (BF_set_loop_bound)。

バリア同期の挿入 並列ループ中では同期の必要がないことを述べたが、並列ループが終了すると、次の処理に移る前にバリア同期をとる必要がある (図 6の BF_barrier)。なぜなら、前の処理の最中に次の処理を始めるプロセッサが存在すると、そのプロセッサは他のプロセッサと通信をするかもしれないからである。もし通信相手がまだ前の処理を実行中で前の処理に参加にしているプロセッサと通信している場合に通信が混乱するおそれがあるからである。

6 SPMD 化と SIMD 化の相違点

ここでは SIMD コードと SPMD コードの相違点をあげる。ここにあげるものは中間コードレベルではなく、具体的なマシンのコード生成に近いコンパイルフェーズで起こる。

```
REAL X(100,100), Y(100,100), Z(100,100)
!HPF$   PROCESSORS P(4,4)
!HPF$   DISTRIBUTE(CYCLIC,CYCLIC) ONTO P :: X,Y,Z

        Z(1:99,:) = Z(1:99,:) + Z(2:100,:)
```

(a)

```
BFMAIN__()
{
    /* some codes dropped .... */

    BF_make_real_PEarray(&r_1, r_1_body, &shape_1);
    SET_SEC_TRIPLET(sec_2[0], 1, 99, 1);
    SET_SEC_TRIPLET(sec_2[1], 1, 100, 1);
    SET_SEC_TRIPLET(sec_1[0], 2, 100, 1);
    SET_SEC_TRIPLET(sec_1[1], 1, 100, 1);
    BF_copy(&r_1, sec_2, &z, sec_1, sizeof(real));
    SET_SEC_TRIPLET(sec_3[0], 1, 99, 1);
    SET_SEC_TRIPLET(sec_3[1], 1, 100, 1);

    BF_set_loop_bound(myself, lb1, ub1, sec_1[0])
    BF_set_loop_bound(myself, lb2, ub2, sec_1[1])

    for (i_1 = lb1; i_1 < ub1; i_1++) {
        for (i_2 = lb2; i_2 < ub2; i_2++) {
            x.body.PEreal[i_1 + i_2* size1] =
                r_1.body.PEreal[i_1 + i_2 * size1]
              + y.body.PEreal[i_1 + i_2 * size1];
}
    }
    BF_barrier();
}
```

図 6: SPMD コード

6.1 ループ制御の違い

ループ回数が不規則なためにアクセスする配列要素数が変化するループがある。図7は、2重ループが2次元配列をアクセスしている様子である(縦棒がアクセスしている部分)。これは1次元目のループ回数(縦棒の高さ)のバランスが悪いので、2次元目の分割をCYCLIC分割にして、負荷分散を行なっている例である。これを単純に2重ループにすると、MIMDでは各プロセッサが独立に働くことから予想通りの負荷分散が行なわれるが、SIMDではすべてのプロセッサが同じことをすることから一番長いループに合わせてしまう(図7(a))。これを防ぐために、ループを図7(b)のように一番長いループの終了を待つことなしに次のループに移れるようにループを平坦化する。[4] その並Cのプログラムが図8(b)である。pforはSM-1プロセッサ側で実行するforループであり、heightに入っているループ長を終了すると、ループの終了をチェックして、次のループに移るようにループインデックスを調節する。pbreakは他のプロセッサがすべて対応するpforを終了するのを待ってからループの外に出る制御文である。

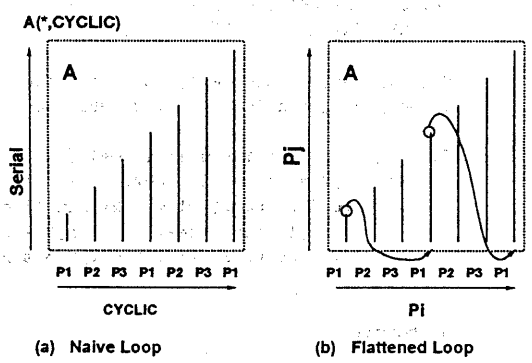

図7: ループの平坦化

6.2 通信コードの違い

MIMDにおける通信で待つというコードを、SIMDでは待つプロセッサとそうでないプロセッサというように条件分岐で陽に分けてやらなければならない。図9(a)はパイプライン的に並列実行できるループ例である。ループの先行する部分を担当するプロセッサの終了を待って次のプロセッサが実行を開始するが、終了した方のプロセッサは外側

```
double $ a[N][H];
int $    pi, pj;

pfor (pi=0; pi<N; pi += NP)
   pfor (pj=0; pj<=height[pi]; pi++)
      a[pi][pj] = ...

           (a)

pi = pj = 0;
pfor(;;) {

     a[pi][pj] = ...

     pif (pj >= height[pi]) {
         pi = pi + NP;
         pif (pi >= N)
             pbreak;
         else
             pj = 0;
     }
}
           (b)
```

図8: ループの平坦化

の次のループに進むために、全体としてはパイプライン的な並列性がある。これに対するSPMDコードが同図(b)である。通信待ちしなければならないプロセッサ(receiveの部分)は、そうでないプロセッサだけが、forループに進む。このコードではプロセッサは待ちが終るまで次を実行しないという前提で生成されている。それに対して、SIMDコードでは、図10のように、pif elseによって、実行する部分を明示的に分けてやらなければならない。

7 関連研究

Fortran D [7] はHPFの設計にも大きな影響を与えたプログラミング言語である。Fortran Dコンパイラは、プログラマによるデータ分割の指示を従来のFortarn 77のプログラムに与えることによってDOループを並列化し、SPMDコードを生成する。基本設計は分散メモリのMIMDマシンをターゲットにしている。

Data-Parallel C[5]コンパイラは、SIMDマシン用のプログラミング言語であるC*を汎用の共有メモリマシンや分散メモリマシンのコードに変換するコンパイラであり、さまざまなアーキテクチャ

```
      REAL,ZA(100)
!HPF$ PROCESSORS P(4)
!HPF$ DISTRIBUTE ZA(*, BLOCK) ONTO P
      DO I = 1, TIME
         DO J = 2,99
            ZA(J) = 1/3*(ZA(J-1)+ZA(J))
```

(a)

```
real za[0:26];
if (myself > 0) lb=1; else lb=2;
if (myself < 3) ub=25; else ub=24;

for (i=1; i<=time; i++) {
  if (myself > 0) receive(za[1],myself-1);
     for (j=lb; j<=ub; j++)
        za[j] = 1/3*(za[j-1]+ZA[j]);
     if (myself < 3) send(za[25],myself+1);
}
```

(b)

図 9: パイプラインを行なう SPMD コード

```
for (i=1; i<=time; i++) {
   pif (myself > 0)
      receive(za[1],myself-1);
   else {
      pfor (j=lb; j<=ub; j++)
         za[j] = 1/3*(za[j-1]+ZA[j]);
      pif (myself < 3) send(za[25],myself+1);
   }
}
```

図 10: パイプラインを行なう SIMD コード

に渡って 1 つのプログラミング言語をサポートした例である。Data-Parallel C ではプログラマが指定する仮想プロセッサに対する最適化を行なわなければならない点が Bee-Fortran のコンパイルとは異なる。Data-Parallel C のコンパイラが仮想プロセッサからなる SIMD マシンを効率よくシミュレートするためにバリア同期の挿入などを工夫している。

8 おわりに

本稿では、SIMD 型と MIMD 型の分散メモリ並列計算機に対して共通のアプローチでコンパイルすることによってポータブルな Bee-Fortran のコンパイラシステムの構築の方式を述べた。Bee-Fortran コンパイラの最初の方式は SIMD モデルに基づいて設計されており、本研究では生成されるコンパイルコードの実行モデルとして SPMD(Single Program Multiple Data) 方式を採用することにより、MIMD に拡張するという方法を取った。SPMD 方式は、SIMD の実行モデルに近いことから SIMD をターゲットモデルにしたコンパイラには親和性の高いアプローチだと考えられる。しかしながら、逆に MIMD 側が行なうことを SIMD 側にも反映するためには、ループの平坦化や通信コードの生成に違いがある。今後課題としては、共有メモリアーキテクチャに対するアプローチなどがあげられる。

参考文献

[1] Albert, E., K., Kathleen, Lukas, J., and Steele Jr., G., Compiling Fortran 8x Array Features for the Connection Machine Computer System, PPEALS, *1988*

[2] Darema, F., George, D., Norton, V., and Pfister, G., A Single Program Multiple Data Computational Model for EPEX/FOTRAN, Parallel Computing, 7, North Holand, *1988*

[3] Fortran90, ANSI X3J3, ISO/IEC 1539 *1991*

[4] Hanxleden, R. and Kennedy, K., Relaxing SIMD Control Flow Constraints using Loop Transformation, CRPC-TR92207, Rice Univ., *April, 1992*

[5] Hatcher, P. and Quinn, M., Data-Parallel Programming on MIMD Computers, MIT Press, *1991*

[6] High Performance Fortran Language Specification Version 1.0, High Performance Fortran Forum, *May 3, 1991*

[7] Hiranandani, S., Kennedy, K., and Tseng, C.,
Compiler Support for Machine-Independent Parallel Programming in Fortran D, COMP TR91-149, Rice Univ., *March, 1991*

[8] 貴島, 湯淺, SIMD 型超並列プログラミング言語「並C」とそのコンパイラ, 情報処理学会研究報告, 92-PRG-9, *October, 1992*

[9] 松田, 湯淺, SIMD 型超並列計算機 SM-1(仮称) の概要とその性能, 情報処理学会研究報告, 92-ARC-87, *August, 1992*

[10] 大谷, 小前, 杉森, 渦原, 安村, 超並列計算機用 Fortran コンパイラの設計と試作, 電子情報通信学会研究報告, COMP92-94, *May 11, 1993*