解 説

並列処理のためのシステムソフトウェア

6. High Performance Fortran†

郷 田 修††

1. はじめに

High Performance Fortran (HPF) は，Fortran 90[7] をもとにした並列記述言語で，次のようなことを目的とし作られている．

（1） データ並列 (data-parallel) プログラミングをサポートする．

（2） MIMD (Multiple Instruction, Multiple Data) および SIMD (Single Instruction, Multiple Data) 計算機の能力を最大限に引き出す．

（3） さまざまなアーキテクチャの計算機に適用可能な標準的な言語を定義する．

なお，基礎となる言語として Fortran 90 が選ばれた理由には，これまで多くの科学技術計算向けのプログラムが Fortran 言語で書かれてきたこと，Fortran 90 が配列代入といったデータ並列に適した機能をもっていることがあげられる．またこのような言語を定義しようとする動きの背景には，超並列 (Massivelly parallel) 計算機が科学技術計算の分野でより多く使われるようになるであろうという予想がある．

言語仕様の作成は，1992 年初めに米国で作られた HPFF (High Performance Fortran Forum) と呼ばれる団体によって行われている．HPFF には現在，40 を超える企業および大学が参加しており，一部日本の企業も参加している．

現在 HPF の言語仕様は Version 1.0 DRAFT[5] であり，これに多少の変更を加えたものが最終的な言語仕様になる予定である．なお，以下の記述はこの仕様にもとづいている．

2. 並列記述言語の系譜

並列記述言語には，低水準のものから高水準のものまで，また制御並列 (control parallelism) を扱うものと，データ並列 (data parallelism) を扱うものなど多くの言語があるが，ここでは HPF が目的としているデータ並列プログラミングをサポートしている言語を，いくつかみてみる．

（1） C＊

C＊[9] は Thinking Machine Corporation の Connection Machine のために設計された言語である．C＊では shape と呼ばれる概念によってデータの配置（次元，サイズ）を指定する．shape の各要素は個々の仮想プロセッサに対応し，変数の宣言で shape を指定することで，変数は shape の要素に対応する仮想プロセッサに置かれる．同じ shape をもった変数どうしの演算は，各プロセッサによって並列に実行される．

（2） CM Fortran

CM Fortran[10] も Connection Machine のための言語で，言語仕様は，Fortran 90 とほぼ同じである．CM Fortran では，配列の各要素はコンパイラによって自動的に分散されて仮想プロセッサ上に置かれ，これらに対する配列演算は仮想プロセッサによって並列に行われる．

（3） Data-Parallel C

Data-Parallel C[3),4)] は初期の C＊言語[8] をもとに作られた言語で，C言語を domain と呼ばれる概念によって拡張したものである．domain は Cの構造体と同様，変数の集まりで，domain の配列の各要素は各プロセッサに割り当てられる．domain に対する演算は，domain 配列の各要素が割り当てられたすべてのプロセッサによって並列に実行される．

（4） Fortran D

Fortran D[2] は Rice 大学で開発された言語で，Fortran 77 に DECOMPOSITION, ALIGN および DISTRIBUTE 文を追加して配列の分散方法，

† By Osamu GODA (IBM, Tokyo Research Laboratory).
†† 日本アイ・ビー・エム(株)東京基礎研究所

配列相互の位置関係を記述できるようになっている．Fortran D では互いに関連する配列を同一のテンプレート（仮想の配列）に ALIGN 文で関連付け，このテンプレートを DISTRIBUTE 文で分散する．演算は通常の Fortran 77 の文を使って記述するが，ALIGN と DISTRIBUTE によって配列を適切に分散することによって効率よく行われる．

3. 言語の特徴

High Performance Fortran は Fortran 90 言語をもとにして，以下の言語仕様上の拡張および制限を加えている．

（1） 新しいディレクティブの追加
（2） 新しい文（文法）の追加
（3） ライブラリ・ルーチンの追加
（4） Fortran 90 言語に対する制限

ディレクティブは，Fortran 90 の注釈行の形式（!HPF$ などで始まる）になっており，処理系に対して，効率のよいコードを生成するための情報を与える．ディレクティブは，計算の効率を別とすれば，プログラムが行う計算の内容には影響をあたえない．

3.1 データの相互配置と分散

分散システム上でデータ並列を実現する場合，データ（主に配列）を分割して各プロセッサに置き，それらに対して演算（できるだけ並列に）を行うことになるが，データを各プロセッサに一様に分散させるだけでは効率の良い計算を行うことはできない．たとえば，

```
REAL A (100), B (100)
DO I=1, 99
  A(I)=B(I+1)
ENDDO
```

というループを 2 台のプロセッサで計算する場合 A, B の最初の 50 個の要素をプロセッサ 1 に置き，残りをプロセッサ 2 に置くとすると，プロセッサ 1 が $I=50$ の計算を行うときに，プロセッサ 2 にある B (51) の値を得るための通信が必要になる．一方，A の最初の 50 要素と B の最初の 51 要素をプロセッサ 1 に置き，残りをプロセッサ 2 に置くと通信は発生しない．

また，データの分割方法についても同様なことが起こる．たとえば，以下のプログラムで，

```
REAL A (100, 100)
DO I=1, 100
  DO J=1, 99
    A (I, J)=A (I, J+1)
  ENDDO
ENDDO
```

A を各列が同一プロセッサに置かれるように分散したとすると，A (I, J) と A (I, $J+1$) は常に別のプロセッサに置かれることになり，多くの通信が発生するが，A の各行が同一プロセッサに置かれるようにすると通信は不要になる．

HPF では上記のような問題を考慮して，データを相互配置 (alignment) と分散 (distribution) という二段階に分けてプロセッサに割り当てるようになっている．また，データを分散するためのプロセッサの構成 (configuration) も指定できるようになっている．以下これらについて，説明の都合上，プロセッサ構成より順に説明する．

（1） プロセッサ構成の指定

データを分散するためのプロセッサの構成（次元と各次元に対するプロセッサ数）は，PROCESSORS ディレクティブによって指定する．ここで指定するプロセッサ数は論理的なプロセッサ (abstruct processor) で，実プロセッサへの割当て方法については処理系に任されている．

以下は，PROCESSORS ディレクティブの例で

```
!HPF$ PROCESSORS P4 (4)
!HPF$ PROCESSORS P22 (2, 2)
```

最初の例は，一次元の 4 プロセッサ構成を指定している．また，二番目は 4 プロセッサを 2 行 2 列に配置したプロセッサ構成の例である．

（2） データの分散

データの分散方法は DISTRIBUTE ディレクティブによって指定する．DISTRIBUTE ディレクティブには分散の対象となる配列名（または template），配列の各次元に対する分散方法および配列が分散されるプロセッサの構成を指定する．分散方法には，BLOCK (n) または CYCLIC (n) (n は省略可能）が指定できる．

以下に DISTRIBUTE ディレクティブの例を示す．（例では，前述のプロセッサ構成 P4 および P 22 を使う．また，A は 100 個要素からなる一次元配列，B を 100 列の配列とする）

```
!HPF$ DISTRIBUTE A (BLOCK) ONTO P4
```

!HPF$ DISTRIBUTE A (BLOCK(30)) ONTO P4

上記二つは BLOCK 分割の例で，最初の例では，Aは4つのプロセッサに分散するために，25個の要素からなる4つのブロックに分けられる．二番目の例ではAは要素数 (30, 30, 30, 10) の4つのブロックに分けられる．次の例は，CYCLIC 分割の例である．

!HPF$ DISTRIBUTE A (CYCLIC) ONTO P4

!HPF$ DISTRIBUTE A (CYCLIC (2)) ONTO P4

最初の例では，n (nは1から4)から始まる4とびの添字をもつ要素がプロセッサnに割り当てられる．二番目の例では，配列要素は先頭から二つずつの組に分けられ，n (nは1から4) 番目から始まる4とびの組に含まれる配列要素が，プロセッサnに割り当てられる．(たとえば，プロセッサ1には $A(1)$, $A(2)$, $A(9)$, $A(10)$ などが割り当てられる)

!HPF$ DISTRIBUTE B (BLOCK, BLOCK) ONTO P22

!HPF$ DISTRIBUTE B(*, BLOCK) ONTO P4

上記は，二次元配列を分散する例である．最初の例では，配列Bは行，列ともに二つのブロックに分割されて4つのプロセッサに割り当てられる．二番目の例では，配列 B は列方向に4つのブロックに分割され，行方向には分割されない．

(3) データの相互配置

データの相互配置は ALIGN ディレクティブによって指定する．ALIGN ディレクティブでは基準となる配列 (Template と呼ばれる仮想の配列を指定することもできる) と，相互配置の対象となる配列を指定する．

次に，ALIGN ディレクティブの例を示す．

!HPF$ ALIGN A(I) WITH B(I)

!HPF$ ALIGN A(I) WITH B(I+1)

!HPF$ ALIGN A(I, J) WITH B(J, I)

最初の例は，どちらも，A の I 番目の要素を B の I 番目の要素と同じプロセッサに置くように指定している．二番目の例は，A の I 番目の要素を B の $I+1$ 番目の要素と同じプロセッサに置くよう指定している．三番目は転置の例で，A の (I, J) 要素を B の (J, I) 要素と同じプロセッサに置くよう指定している．

次の例は，異なった次元の配列どうしの ALIGN の例である．

!HPF$ ALIGN A(I,*) WITH B(I)

!HPF$ ALIGN A(I) WITH B(I,*)

最初の例は，A の I 行目のすべての要素を B の I 番目の要素と同じプロセッサに置くよう指定している．二番目の例では，A の I 番目の要素を B の I 行目の要素をもつプロセッサに重複して置くよう指定している．

3.2 新たに追加された文

HPF では Fortran 90 の配列代入文をより一般的にした FORALL 文と FORALL 構造 (construct) が追加された．FORALL 文は，本体に代入文を一つだけもつ DO ループに似ているが，代入文の評価方法が異なっている．DO 文の場合は各インデックスの値について代入文を実行するが，FORALL 文ではすべてのインデックスについて代入文の右辺の値を評価した後，それらの値が一斉に左辺に代入される．このような評価方法をとると，前の繰返しで代入された結果を後の繰返しで使用するという，繰返し間の依存関係がなくなる．このため，右辺の評価をすべてのプロセッサで並列に行い，その結果を並列に代入することができるようになる．下記は，FORALL 文の例である．

```
FORALL (K=1, M-1) X(K+1)=X(K)
FORALL (I=1:N, J=1,N) X(I, J)=Y(J, I)
```

最初の例では，文の実行後，配列要素 $X(I)$ には実行前の $X(I-1)$ の値が入る．二番目の例では，文の実行後 X には Y の転置行列が代入される．

また，FORALL 文では論理式 (Mask) を指定して代入文の評価を制限することもできる．下記は，Mask 付きの FORALL 文の例である．

```
FORALL (K=1 : M, Y(K). NE. 0. 0) X(K)
  =1.0/Y(K)
```

この文では $Y(K)$ の値が0でない k に対して，$X(K)$ に $Y(K)$ の逆数が代入される．

FORALL 構造は，FORALL 文を一般化したもので FORALL と ENDFORALL の間に複数の文を書けるようにしたものである．ただし，FORALL 構造中で使用できる文は，代入文，FORALL 文，FORALL 構造，WHERE 文 (Fortran 90) および WHERE 構造である．

FORALL 構造中の文は，FORALL 文の場合と同様，FORALL で指定される添字の集合をもとに，順次実行される．下記は，FORALL 構造の例である．

```
FORALL (I=2:9)
  A(I)=A(I-1)+A(I+1)
  B(I)=A(I)
END FORALL
```

この例では，初めに2から9までの各 I について $A(I-1)+A(I+1)$ が評価され，その結果が $A(2)$ から $A(9)$ に代入され，次にこうして求められた $A(2)$ から $A(9)$ の値が $B(2)$ から $B(9)$ に代入される．

3.3 ライブラリ・ルーチン

HPF には，以下のようなルーチンが追加されている．

（1） システム問合せ関数 (System Inquiry Intrinsic Functions)

（2） マッピング問合せ関数 (Mapping Inquiry Intrinsic Functions)

（3） 演算用ライブラリ関数 (Computational Library Functions)

システム問合せ関数は，プロセッサに関する情報を返す関数で，使用可能なプロセッサ数を返す NUMBER_OF_PROCESSORS 関数と，プロセッサ配列の情報を返す PROCESSORS_SHAPE 関数とがある．

マッピング問合せ関数は，実行時にデータの分散や相互配置についての情報を返す．マッピング問合せ関数には，データの相互配置情報を返す HPF_ALIGNMENT 関数，テンプレートの情報を返す HPF_TEMPLATE 関数，データの分散状態の情報を返す HPF_DISTRIBUTION 関数がある．

演算用ライブラリ関数は，配列中の値の最大値，n 番目の要素までの和といったよく使われる演算を提供する．一般に，これらの演算は，Fortran でループを使って記述した場合よりも，ネットワークの特性などを考慮して作成されたライブラリ関数を使ったほうが効率がよい．HPF で提供されるライブラリ関数には次の5種類がある．

（1） リダクション (reduction) 関数

（2） 結合分散 (combining-scatter) 関数

（3） プリフィックス (prefix) およびサフィックス (suffix) 関数

（4） ソート (sorting)関数

（5） ビット操作 (bit-manipulation) 関数

3.4 Fortran 90 言語に対する制限

Fortran では，配列要素などのデータは，一次元の連続した記憶域に，一定の順序にしたがって配置される．また，これらのデータは，この記憶順序をもとに，EQUIVALENCE 文や，副プログラム呼出しにおける実引数と仮引数の結合によって互いに結びつけられる．このため，一次元配列の一部が，二次元配列として参照されたり，ある変数の一部が，他の変数の一部と記憶域を共有するといったことが起きる．これは HPF の配列を一つのオブジェクトとしてプロセッサに分散するという概念と一部矛盾する．例として，次のプログラムについて考えてみる．

```
      REAL A (100), B(100)
      EQUIVALENCE (A(51), B(1))
!HPF$ PROCESSORS P2(4)
!HPF$ DISTRIBUTE A (BLOCK) ONTO P2
```

このプログラムでは，2行目の EQUIVALENCE 文で A の最後の 50 要素と B の最初の 50 要素を共有するよう指定されており，4行目の DISTRIBUTE ディレクティブで A を4つのプロセッサに分散するよう指定されている．この場合，A と B の最初の 50 要素はディレクティブに従って分散できるが，B の後半の 50 要素については分散方法が指定されていない．HPF ではこのような指定は禁止されている．上記プログラムは，EQUIVALENCE によって結合された変数 A, B 全体を含む別の変数を導入してこれを分散するようにすると，正しい HPF プログラムになる．

```
      REAL A (100), B(100)
      REAL COVER (150)
      EQUIVALENCE (A(51),B(1))
      EQUIVALENCE (COVER(1),A(1))
!HPF$ PROCESSORS P2(4)
!HPF$ DISTRIBUTE COVER (BLOCK)
  ONTO P2
```

また，副プログラム呼出し時の実引数と仮引数の結合についても制限が加えられている．たとえば，Fortran 90 では，配列要素である実引数を，仮引数の配列に渡したり，二次元配列を一次元配

列に渡すことができるが，HPF でこのようなことを行うには SEQUENCE ディレクティブを使って次のように書く必要がある．

```
      REAL A (100, 100), B(200)
!HPF$ SEQUENCE A, B
      CALL S (A, B(51))
      END
      SUBROUTINE S (X, Y)
      REAL X (10000), Y (100)
!HPF$ SEQUENCE X, Y
      ...
      END
```

4. 応用事例

HPF は，まだ言語仕様を作成する段階にあり，応用事例も発表されていないが，HPF が目的とするデータ並列の応用例について知るものをあげておく．

文献 3), 4) には，行列の積をはじめとして，海洋循環モデル，大気モデルなどについてデータ並列プログラミングを使った例と，その実行効率についてのデータが報告されている．

また，地震の差分モデルを CM Fortran で記述しこれを CM-2 で実行して 5.6 gigaflops という結果を得たことが報告されている[9]．

5. 研究動向と今後の展望

ここでは，実際に処理系を作るにあたっての技術的な問題点や研究動向について述べる．

前述のように，HPF ではプログラマが ALIGN, DISTRIBUTE などのディレクティブを使って解くべき問題に適したデータの分散方法を指定できるようになっている．また並列演算に適した文や関数も用意されているが，生成される目的コードの効率は処理系に大きく依存する．

効率の良い目的コードを生成するための処理系を作成する上での問題点としては，以下のことがあげられる．

（1） できるだけ各プロセッサが並列に動作するようなコードを生成する．

（2） プロセッサ間の通信を，できるだけ少なく，また効率良く行えるようなコードを生成する．なお，これらを解決するために行われる最適化については，後述する．

5.1 SPMD プログラム

MIMD 型の計算機によって並列プログラムを実行する場合，各プロセッサに同一のプログラムのコピーを置く方法と，各プロセッサにそれぞれ異なったプログラムを置く方法とがあり，前者の方法は SPMD (Single Program Multiple Data) モデルと呼ばれる．SPMD モデルによる方法は，各プロセッサが行う演算が大きく異なる場合には向かないが，データ並列の場合のように，各プロセッサが行う演算がほぼ同一である場合に適しており，多くの並列化コンパイラで採用されている．

次に，HPF プログラム中の代入文をコンパイルして，SPMD プログラムにする場合を考えてみる．この場合，最も簡単なコンパイル方法は次のようになる．

（1） 各プロセッサは，自分がその所有者なら，代入文の右辺の評価に必要なデータを，代入文の左辺の所有者に送る．

（2） 代入文の左辺の所有者は，他のプロセッサから送られたデータをもとに右辺の式を評価して，左辺に代入する．以下の例は，HPF プログラムを SPMD プログラムに変換した例である．

（a） HPF プログラム

```
DO I=1, 100
  A(I)=B(I)+C(I)
ENDDO
```

（b） SPMD プログラム

```
DO I=1, 100
  IF(iown(A(I)))THEN
    IF(iown(B(I)))THEN
      T1=B(I)
    ELSE
      T1=receive(owner(B(I)))
    ENDIF
    IF(iown(C(I)))THEN
      T2=C(I)
    ELSE
      T2=receive(owner(C(I)))
    ENDIF
    A(I)=T1+T2
  ELSE
    IF(iown(B(I)))THEN
      send(owner(A(I)), B(I))
    ENDIF
```

```
    IF(iown(C(I)))THEN
      send (owner (A(I)), C(I))
    ENDIF
  ENDIF
ENDDO
```

(b)の SPMD プログラムで，最初の IF 文は，そのプロセッサが $A(I)$ を所有しているかどうかを調べ，所有者なら $B(I)$ と $C(I)$ の所有者 (owner ($B(I)$) と owner ($C(I)$) から $B(I)$ と $C(I)$ の値を受け取り，それらの和を $A(I)$ に代入する．また，最初の IF 文に対応する ELSE 以降は $A(I)$ を所有しないすべてのプロセッサによって実行され，左辺 $A(I)$ の所有者 (owner($A(I)$)) に所有しているデータを送る．

5.2 最　適　化

前記の例からも分かるように，HPF で書かれたプログラムは単純にコンパイルしたのでは効率のよい目的プログラムに変換することはできない．たとえば，前記の例では DO ループ中に通信が入っており，このために各インデックスごとに同期が取られてしまい，DO ループは実質的には並列に実行されない．したがって，HPF 用のコンパイラでは最適化が非常に重要になる．以下，いくつかの最適化の手法について述べる．

(1)　ループ分割

ループ分割 (loop distribution)[12] は，複数の文を含むループを，より少ない文を含む複数のループに分割する．この方法は，逐次プログラムのベクトル化などで使われるが，HPF プログラムの並列化にも有効である．

(a)
```
DO I=2,100
  A(I)=B(I)+C(I)
  D(I)=A(I-I)+1
ENDDO
```

(b)
```
DO I=2,100
  A(I)=B(I)+C(I)
ENDDO
DO I=2,100
  D(I)=A(I-1)+1
ENDDO
```

たとえば，上記プログラムで(a)のループは並列に実行できない（たとえば，インデックス k について実行するプロセッサが $A(k)$ への代入を行う前にインデックス $k+1$ について実行するプロセッサが $A(k)$ を参照するかも知れない）これを，(b)のように二つのループに分割すると，どちらのループも並列に実行できるようになる．

(2)　ループ・インタチェンジ

ループ・インタチェンジ[11),12)] は，多重ループの順序を入れ替えて，目的に応じて，内側または外側のループを並列実行できるようにする技法である．HPF では，ループの入替えにより通信を外に出せることもある．

```
      REAL A (100,100)
!HPF$ PROCESSORS P4(4)
!HPF$ DISTRIBUTE A (BLOCK, *) ONTO
  P4
      DO I=2,100
        DO J=1,100
          A(J,I)=A(J,I-1)+1.0
        ENDDO
      ENDDO
```

上記の例では，内側のループは並列に実行可能であるが，外側は並列実行可能ではない．このため I の各値について同期を取りながら実行しなければならない．一方，ループを入れ替えると，外側のループが並列実行可能となり，各プロセッサが内側のループを独立に実行可能となる．

(3)　メッセージのベクトル化

この技法は，SPMD プログラムの例で示したようなループ中で行われる通信を，まとめてループの外で行うためのもので，これにより通信に要する時間を大幅に短縮できる．

```
      REAL A (100, 100), B (100, 100)
!HPF$ PROCESSORS P2 (2)
!HPF$ DISTRIBUTE A(* ,BLOCK) ONTO
  P2
!HPF$ DISTRIBUTE B(*, BLOCK) ONTO
  P2
      DO I=1,99
        DO J=1,99
          A (I, J)=B (I+1, J+1)
        ENDDO
      ENDDO
```

たとえば，上記のプログラムの場合，ループ開始前に，B の 51 列目の要素をプロセッサ 1 にもってきておくことにより，通信なしでループを実行できるようになる．

(4) パイプライン化

二重以上のループでは，データ依存関係があって完全には並列化できない場合でも，各プロセッサが必要なデータがそろい次第実行を始めるといった方法をとることで並列化できる場合がある．

```
      REAL A (200, 200)
!HPF$ PROCESSORS P44 (4, 4)
!HPF$ DISTRIBUTE A (BLOCK, BLOCK)
  ONTO P44
      DO I=2, 200
        DO J=2, 200
          A(I,J)=A(I,J-1)+A(I-1,J)
        ENDDO
      ENDDO
```

上記のプログラム例では，ループの実行方法を変更して，初めにプロセッサ (1, 1) が (I=1, 50, J=1, 50) についてループを実行し，次にその結果である $A(1:50, 50)$ と $A(50, 1:50)$ を受け取ったプロセッサ (1, 2) とプロセッサ (2, 1) が並列に実行を開始するといった方法でも正しい結果が得られる．この場合，プロセッサの実行順序は，以下のようになり最大4つのプロセッサが並列実行されることになる．

P(1, 1)
P(1, 2), P(2, 1)
P(1, 3), P(2, 2), P(3, 1)
P(1, 4), P(2, 3), P(3, 2), P(4, 1)
P(2, 4), P(3, 3), P(4, 2)
P(3, 4), P(4, 3)
P(4, 4)

この技法はパイプライン化 (pipelining)[6] と呼ばれている．

5.3 今後の課題

以上，処理系について最適化を中心に述べてきた．例にあげた最適化の技法は，最適化コンパイラやベクトル化コンパイラのために開発されたもので，HPF のような言語の処理系においても有効である．しかし，HPF には，データの分散とそれにともなう通信の問題があり，これらの技法を適用する場面をより複雑にしている．上記のループインタチェンジの例でも，もしデータの分散方法が例のものと異なっているならば，ループインタチェンジは必ずしも有効ではない．また，HPF の仕様によれば，分散方法，プロセッサ数などが実行時まで決まらないことがあるが，このような場合に対していかに効率のよいコードを生成するかは今後の課題であろう．

6. おわりに

HPF は，まだ言語仕様自体も最終的に決まっておらず，本格的な処理系も発表されていない．したがって，今後 HPF がどのように使われていくかは未知数である．しかし，これまで Fortran プログラムを並列化する際に，手で通信用ライブラリを挿入したり，配列を分割していたことを考えると，HPF のような言語を使うことによって，これらの作業をコンパイラに任せることができ，並列化のための労力が軽減されることが期待できる．

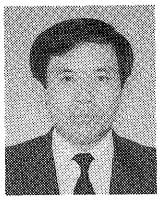
また，最近，高速な通信機能をもった分散記憶型の多重プロセッサがいくつか発表されているが，このようなプロセッサを生かす上で HPF のような言語が使われるようになると思われる．

参考文献

1) Bromley, S., Hellaer, S., McNerney, T. and Steele Jr., G. L.: Fortran at Ten Gigaflops: The Connection Machine Convolution Compiler, Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation (June. 1991).
2) Fox, J. et al.: Fortran D Language Specification, Report COMP TR90-141 (Department of Computer Science, Risce University) and SCCS-42c (Syracuse Center for Computing Science, Syracuse University) (Apr. 1991).
3) Hatcher, P. J. and Quinn, M. J.: Data-Parallel Programming on MIMD Computers, MIT press (1991).
4) Hatcher, J., Quinn, M. J., Lapadula, A. J., Seevers, B. K., Anderson, R. J. and Jones, R. R.: Data-Parallel Programming on MIMD Computers, IEEE Transactions on Parallel and Distributed Systems, Vol. 3, No. 2 (July 1991).
5) High Performance Fortran Forum: High Performance Fortran Language Specification, Version 1.0 DRAFT (Jan. 1993).
6) Hiranandani, S., Kennedy, K, and Tseng, C.: Compiling Fortran D for MIMD Distributed Machines, CACM, Vol. 35, No. 8, pp. 66-80 (Aug. 1992)).
7) ISO: Fortran 90, May 1991. (ISO/IEC 1539: 1991 (E)).
8) Rose. J. and Steele, G. Jr.: C*: An Extended C Language for Data Parallel Program-

ming, Proceedings of the Second International Conference on Supercomputing (May 1987).
9) Thinking Machine Corporation : C* Programming Guide (1990).
10) Thinking Machine Corporation : CM Fortran Reference Manual (July. 1991).
11) Wolfe, M.: Optimizing Supercompilers for Supercomputers, The MIT Press (1989).
12) Zima, H. with Chapman, B.: Supercompilers for Parallel and Vector Computers, ACM Press (1991).

(平成5年3月11日受付)



郷田 修 (正会員)

昭和51年上智大学理工学部物理学科卒業. 昭和53年電気通信大学大学院電気通信学研究科修士課程修了, 同年(株)三菱総合研究所入社. 昭和59年日本アイ・ビー・エム(株)入社, 現在, 基礎研究所にて, 主にコンパイラに関する研究に従事, ACM会員.