数式処理 *J.JSSAC (2005)*
Vol. 11, No. 3,4, pp. 125 - 140

Noda2005 論文

並列化コンパイラ fhpf の 正規化変換で用いられる数式処理

岩下英俊*

富士通 (株) ソフトウェア事業本部

概　　要

High Performance Fortran (HPF) is a parallel programming language for scientific computation. We have been developing an HPF compiler called fhpf, which is a translator reading HPF programs and writing Fortran programs. In order to handle huge variety of data distribution and alignment provided in HPF, fhpf normalizes mapping of data and computation into a standard internal representation. Normalization encourages compiler optimization and reduces the development cost. Normalization, a technique of compiler, is based on symbolic computation. And the procedure of normalization often generates redundant expressions which should be reduced using symbolic computation techniques. Such contribution of symbolic computation will be required more in the future from parallelizing compilers.

1　はじめに

ハイエンド計算の分野では，共有メモリ方式の並列度の限界から，SMP クラスタや ccNUMA にアーキテクチャのトレンドが移りつつあり，PIM (Processor in Memory) や Reconfigurable といった新しい考え方も生まれている．一方で PC クラスタやブレードサーバなどの比較的安価な並列計算の需要も広がりつつある．ハードウェアが多様化する中で，利用者は特定のアーキテクチャへの依存度が小さい抽象性の高いプログラミング言語を望んでいる [1].

High Performance Fortran (HPF)[2] は，分散メモリ型の並列計算機に適した抽象度の高い言語として知られている．HPF に実用性の観点から拡張を施した HPF/JA1.0 言語仕様 [3] が実装され普及が進み，近年になってやっと実世界プログラムで使えるものとなってきた.

我々は富士通のベクトル並列計算機 VPP5000 向けに HPF コンパイラを開発してきた [4]．現在はこれを発展させ，アーキテクチャの多様化への対応と HPF/JA 仕様の普及推進のために，PC クラスタからスパコンまでどのような環境にも対応できる HPF コンパイラ fhpf をプロトタイプ開発している [5]．fhpf は HPF プログラムを入力とし Fortran プログラムを出力とするコンパイラ（トランスレータ）である.

*iwa@soft.fujitsu.com

本稿では，fhpf コンパイラの中で用いられている数式処理，特にデータやループからプロセッサへのマッピングを正規化する処理について述べ，今後の数式処理の並列化コンパイラへの応用を議論する．

本稿の構成は以下の通りである．まず，並列計算機を対象とするコンパイラに共通する課題と，本稿で取り上げる正規化処理の重要性について次の第 2 章で述べる．第 3 章で fhpf コンパイラの構成と内部構造を簡単に紹介した後，fhpf コンパイラで用いている数式処理の基本部分と特徴について第 4 章で述べる．第 5 章では，この数式処理基盤を活用した例として，我々が提案する正規化変換アルゴリズムを紹介する．第 6 章でまとめとする．

2 分散メモリ型並列計算機向けコンパイラの課題

2.1 並列プログラミングの方法

並列計算機で数値計算を行う場合，以下のような方法がある．

1. MPI (Message Passing Library)[6] などの通信ライブラリや，マシンに固有な低レベルの通信ライブラリを使って，プロセッサ間の通信を含む計算機の挙動を直接記述する方法.
2. HPF (High Performance Fortran)[2, 3]，OpenMP[7, 8]，VPP Fortran[9]，Co-Array Fortran (CAF)[10] などの並列プログラミング言語を利用して，プログラムの並列化方法をユーザが指示する方法．この場合，通信や同期などの並列実行のために必要な細部の挙動はコンパイラが自動的に生成するので，ユーザは原則として記述する必要はない.
3. Fortran や C などの並列性を意識しないプログラムを元に，自動並列化コンパイラやインタラクティブなツール類によって並列化を行う方法.

多くのユーザは，プログラミングの容易さから 2. か 3. またはこれらの組み合わせを望む．共有メモリ型の並列計算機においては，OpenMP 言語が広く使われ始めていて，自動並列化もある程度使われていると言える．しかし分散メモリ型の並列計算機の場合には，2. や 3. では性能的に不十分と感じてやむなく 1. を使うユーザがまだ多い．コンパイラ技術によって方法 2. や 3. のさらなる高性能化が必要とされている．

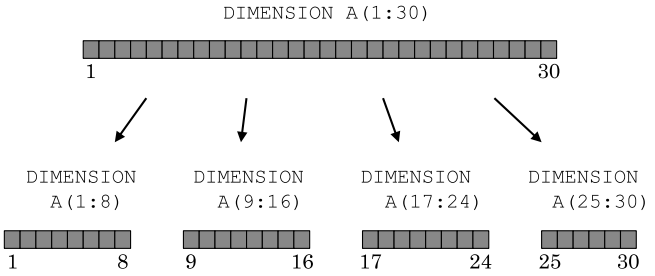
共有メモリ型の計算機と比べ，分散メモリ型の計算機ではプロセッサ間の通信が必要であり，また，通信を削減するための適切なデータの分割配置が必要である．2. のアプローチではデータの分割配置はユーザが指示することが多いが，その指示を生かしたプログラムの解析と変換をどれだけ適切に行えるかによって，通信の効率は大きく左右される．

2.2 整列と分散

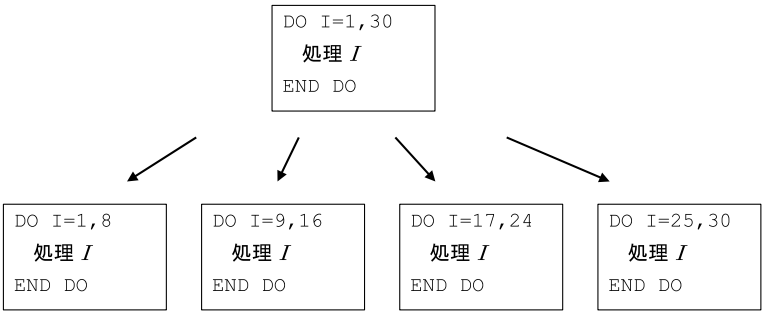
データの分割配置の指示は，HPF 言語では整列 (alignment) と分散 (distribution) という 2 つの概念で表現される．

分散は図 1 に示すようにデータや計算ループとプロセッサを対応付ける方法であり，その種別は HPF の仕様書 [2, 3] で定義される．図 2 は様々な分散種別で 30 要素を 4 プロセッサに分散する様子を示している．分散種別は，アプリケーションの性質に合わせてユーザが選択する．

整列は，配列変数どうし，または配列変数とループの相対的な位置関係を表現するものであり，その目的は分散メモリ型計算機で生じるプロセッサ間通信を最小にすることである．整列に



(a) データの分散



(b) ループの分散

図 1: データ分散とループ分散の例

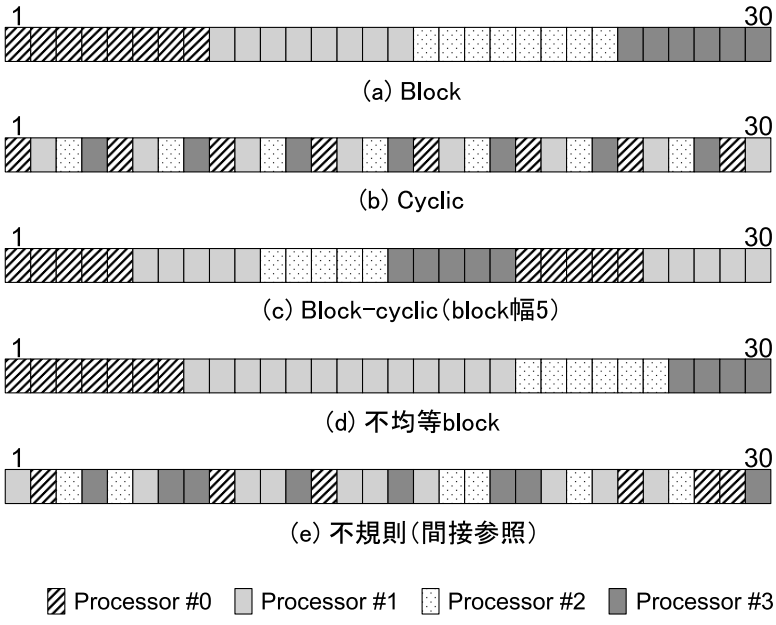
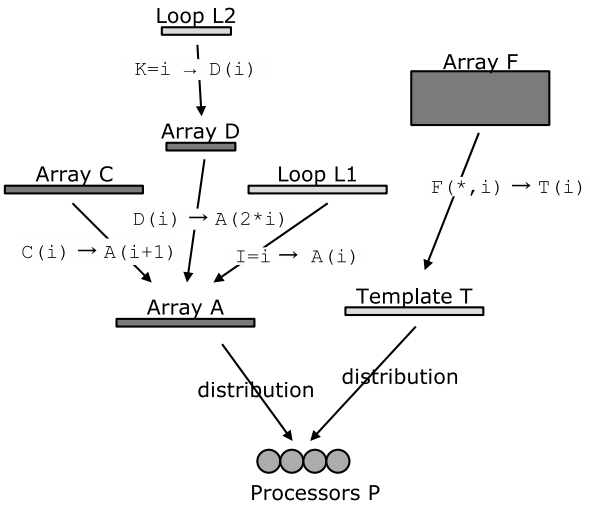


図 2: 分散の種類

```
 1 !hpf$ processors P(4)
 2       real A(1:30),C(1:29),D(1:15)
 3       real F(1:10,1:30)
 4 !hpf$ template T(30)
 5 !hpf$ distribute A(block) onto P
 6 !hpf$ distribute T(block) onto P
 7 !hpf$ align C(I) with A(I+1)
 8 !hpf$ align D(I) with A(2*I)
 9 !hpf$ align F(*,I) with T(I)
10
11 !hpf$ independent
12       L1: do I=2,29
13 !hpf$  on home(A(I)) begin
14         A(I)=C(I)*I+F(I,I)
15 !hpf$  end on
16       end do
17
18 !hpf$ independent
19       L2: do K=1,10
20 !hpf$  on home(D(K)) begin
21         D(K)=A(2*K)
22 !hpf$  end on
23       end do
24
25       end
```

(a) プログラム例



(b) 整列の様子

図 3: HPF プログラムの例

よって対応付けられた配列要素どうしは，分散種別によらず同じプロセッサに配置されることが保障される．整列によってループインデックスに対応付けられた配列要素は，そのループを並列実行するときに実行するプロセッサ上に配置されていて，プロセッサ間通信なしでアクセスできることが保障される．

図 3(a) に HPF プログラムの例を示し，その整列関係を同図 (b) に図示する．HPF は Fortran90 をベースとし，データの分散や並列化に関する指示を “!hpf$” で始まる指示文で与える言語である．例えば配列 A は 5 行目の DISTRIBUTE 指示文によってプロセッサ群 P への分割割付けが指定されていて，配列 D は 8 行目の ALIGN 指示文によって配列 A への整列が指定されているが，この組合せの効果により，配列 D の各配列要素がプロセッサ群 P のどこへ割り付けられるかが決定される．また，ループ L1 は 13 行目の ON 指示文によって配列 A との間の整列関係が指定されていて，A の分散との組合せの効果により，L1 の反復実行のうちどの部分範囲がどのプロセッサで実行されるべきかが指定されている．

より一般的には，HPF 言語のサポートには，多次元ループの整列と分散，図 2 で例示した種々の分散種別，重複配置，袖（シャドウ）付き分散などや，プロセッサ台数や分散方法が不定の場合や可変の場合も考慮に入れる必要がある．

VPP Fortran でも HPF の整列と分散に似た概念でデータとループのマッピングを記述することができる．OpenMP はループの分割に HPF の分散と同じような考え方がある．

2.3 正規化の重要性

分散メモリ型計算機での並列処理の効率化は，プロセッサ間通信の削減と効率化に尽きると言っても過言ではない．無駄な通信は致命的なオーバヘッドコストとなり，いくら計算負荷の分散がうまくいっても並列効果が得られないどころか，逐次実行と比較して数十倍，数百倍の実行時間がかかることも珍しくない．ソースコードの解析による通信の削減と最適化は，コンパイラの重要な機能である．

整列や分散のバリエーションの豊富さは，ユーザにとってはプログラムの汎用性を高める表現の自由度となるが，コンパイラにとっては解析の複雑化の要因となっている．変数の局所性や通信の削減に関する研究は古くからあり，論理的には多くの問題が解決しているが，それにも関わらず実際に商用化されているコンパイラで実プログラムに対して十分な性能が出ない場合があるのは，このバリエーションに対して十分に対応し切れていないためと言える．

例えば，HPF で許される整列のパターンは線形な変換 $i \to ai+b$（a,b は定数）に限定されているので，多段の整列を結合するには線形変換の合成でよい，ということは論理的に明らかであるが，では具体的に，図 3 の 14 行目の `F(I,I)` の参照について他プロセッサからの通信が必要か，ということをコンパイラが自動的に判定するのはそう簡単ではない．この例はむしろ簡単なケースであり，実プログラムで出現するバリエーションのすべてに完全に対応することは開発工数の面から事実上困難である．

こういったバリエーションの爆発に対して，コンパイラはユーザが頻繁に使うだろうと思うパターンを重点的にサポートし，それ以外については，効率は良くないが正常動作は保証する汎用的な処理に落すことが多かった．汎用的な処理では，通信が必要でないとの判断が難しい場合には，通信が必要か不要かを実行時に判断するコードを生成することになる．その場合，たとえ結果的に通信が不要であっても余計な命令実行が生じ，通信が不要と事前に分かっていた場合の命令実行（例えば load 命令 1 つ）に対して数倍〜数百倍のコストがかかることがある．加えて，本来コンパイル時に行うことのできる判断を実行時に持ち越しがちになるため，情報を実行時に伝えるインタフェースも必要になり，生成コードは大きく重いものとなる．そして，実行時ライブラリで扱うパターンも多くなり，そこでもバリエーション爆発の問題が起る．全体として，バリエーションの多さから大きなシステムとなりがちであり，開発コストが大きく，メンテナンスや改善が困難になる．

コンパイラや実行時ライブラリの開発コストを下げるためには，データと計算との相互の整列関係のバリエーションを最小にすること，すなわち，ある一定の標準形に正規化することが非常に有効である．コンパイル時に正規化を行うためには，プロセッサ数や自プロセッサ番号など，コンパイル時には評価できないパラメタを多く含む数式を処理対象とする必要がある．そのために我々は，数式の内部表現を工夫し，基本演算を整備して，数式処理の技巧を取り入れることでその解決を図った．

図 4: fhpf トランスレータのフロー

3 HPF コンパイラ fhpf

3.1 fhpf コンパイラの構成

HPF コンパイラ fhpf は，HPF プログラムを入力として，Fortran プログラムコードを出力するコンパイラ（トランスレータ）である．現在，Linux 版と Solaris 版があり，HPF 推進協議会から会員向けに無償配布を行っている [12]．fhpf の処理フローの概略を図 4 に示す．

fhpf の出力コードには MPI1.1 API に準拠した MPI ライブラリの呼出しが含まれる．fhpf の処理系は独自の実行時ライブラリを持たず，通信にはすべて標準の MPI インタフェースを使っている．また，出力する Fortran コードも標準の Fortran90 仕様の範囲なので，任意の Fortran コンパイラと MPI ライブラリを持つ環境であれば，どこでも翻訳し実行することができる．

fhpf の構成上の一つの特徴は，並列化・最適化処理の前の正規化処理を重視していることである．正規化処理は等価変換なので最適化の一種とも言える．

3.2 プログラムの内部表現

図 4 に示した各処理モジュールのうち，パーサはソースプログラムから内部表現への変換，コード生成は内部表現からファイル出力を行うが，それ以外のパスはすべて内部表現から内部表現への変換である．

内部表現の実体は，C 言語の構造体が相互にリンクされた構造である．図 5 に，簡単なプログラムに対応する内部表現の例を示す．左端にあるセルが，プログラム単位の表現の根となる Procblock 構造体である．Procblock は Syment 構造体のリスト Syment_l や，実行文のリスト Stmt_l などを指している．リストの内部表現は一般にこのように，head と tail のセルに挟まれた双方向リンクを持つセルの並びとする．Syment はプログラム中に出現する変数名，関数名な

どの個々のシンボルの情報を保持している．名前の検索を高速にするため，Syment はハッシュ表からも管理されている．実行文はその種別に応じて Expr_block（代入文），Do_block（DO 構文ブロック）などの構造体で表現される．ブロック構文は内部に Stmt_l 構造をリンクし，プログラムのブロック構造に対応した入れ子構造となる．

同図で影を付けた範囲はそれぞれ数式を表現している．数式の内部表現と扱いについては次章で詳細に述べる．

4 fhpf における数式処理

本章では，fhpf コンパイラにおける式の内部表現（4.1 節），式の基本操作（4.2 節），および式の簡単化（4.3 節）について述べる．

4.1 式の内部表現

図 5 で示したように，プログラムの内部表現の中では，様々な箇所で数式の内部表現が出現する．数式の内部表現は，Expr 構造体をセルとする再帰的なツリー構造が基本となる．Expr 構造体が保持する主な情報は以下のものである．

- 共通の情報
  - 式の種別：定数 (constant)，シンボル (symname)，一項演算 (unary)，二項演算 (binary)，配列要素や部分配列 (array)，実引数の並び (aargs)，添字三つ組 (triplet) など．
  - 型 (type)：整数型，実数型，複素数型，論理型，文字型，派生型（構造体）など．それぞれについて 4 バイト，8 バイトなどの精度の区別がある．
  - 次元数 (rank) と各次元の寸法 (extent)：式が多次元の形状を持つとき（配列変数の引用，部分配列や配列式など），その形状を表現する．
- 式の種別による情報
  - 定数のとき，その値．実数型と複素数型の値は，内部表現への変換・逆変換で精度誤差が生じることを避けるため，ユーザが記述した表現をそのまま文字列で保持する．
  - シンボルの引用のとき，Syment 構造体へのポインタ．Syment 構造体は各シンボルに関する情報を保持する．
  - 一項演算と二項演算のとき，演算子の種類と，オペランド (Expr) へのポインタ．
  - 配列要素や部分配列のとき，その親変数のシンボル (Expr) へのポインタと，各添字式 (Expr) へのポインタの配列．
  - 添字三つ組のとき，下限 (Expr)，上限 (Expr)，刻み幅 (Expr) へのポインタ．
  - 実引数の並びのとき，実引数を表現する ActualArg 構造体のリスト．ActualArg 構造体は，それぞれ 1 つの Expr 構造体へのポインタを持つ．

代入文は，左辺の式と右辺の式をオペランドとする二項演算として表現する．手続き（関数またはサブルーチン）の呼び出しもまた，手続き名と実引数の並びをそれぞれオペランドとする二項演算で表現する．

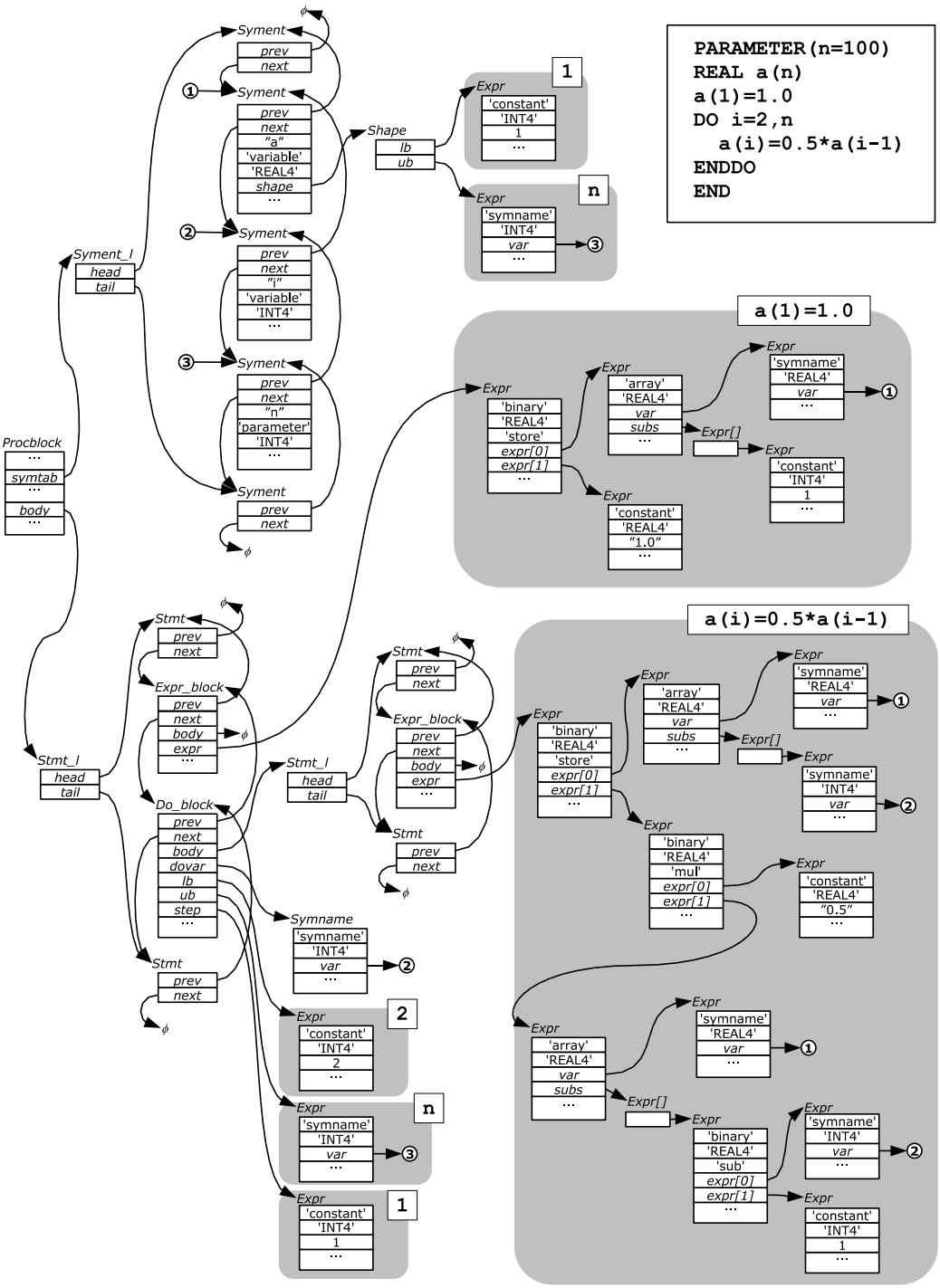


図 5: プログラムの内部表現の例

4.2 式の変換

4.2.1 式に対する基本操作

fhpf ではプログラム中に出現する数式に関して以下の 2 つの操作を頻繁に行っている．

演算 部分式 e に対して何らかの演算 f を作用させる．すなわち，$e \to f(e)$.
例えば，「配列 A の配列要素の参照で第 1 添字から 1 を引く」といった処理である．基本的には，e の式木を一旦リンクから取り外し，$f(e)$ に対応する式木を組み立てて，それを元の場所にリンクする．

置換 数式に含まれる変数や部分式 e_1 を，他の式 e_2 に置き換える．
例えば，「変数 k の参照をすべて式 (2*i+1) に置換する」といった処理である．基本的には，e_1 の式木をリンクから取り外して抹消し，代りに e_2 のコピーを同じ場所にリンクする．

一般に MAXIMA，Mathematica などの著名な数式処理システムでは，多項式などの数式については一定の標準形に変換して扱うが，fhpf では上記の基本操作が高速にできることを優先して特別な標準形を持たないこととした．

4.2.2 プログラムの等価変換

上述の基本操作を用いたプログラムの等価変換を紹介する．5 章で述べる正規化は，この等価変換と，整列の合成計算を原理とする．

配列形状のアフィン変換

m-次元配列 A の配列要素の空間に対する，次元独立なアフィン変換

$$\begin{pmatrix} i_1 \\ \vdots \\ i_m \end{pmatrix} \longrightarrow \begin{pmatrix} a_1 i_1 + b_1 \\ \vdots \\ a_m i_m + b_m \end{pmatrix}$$

は，以下のような数式の基本操作で可能である．

- A の形状宣言
  DIMENSION A(l_1:u_1,$\cdots$,l_m:u_m)
  の第 k 次元について（$k = 1, \cdots, m$），

$$\begin{aligned} l_k &\to a_k l_k + b_k \\ u_k &\to a_k u_k + b_k \end{aligned}$$

- A の引用 A(s_1,$\cdots$,s_m) の第 k 添字について（$k = 1, \cdots, m$），
  スカラ s_k であるとき，

$$s_k \to a_k e_k + b_k$$

添字三つ組 $s_{k1}:s_{k2}:s_{k3}$ であるとき，

$$\begin{aligned} s_{k1} &\rightarrow a_k s_{k1} + b_k \\ s_{k2} &\rightarrow a_k s_{k2} + b_k \\ s_{k3} &\rightarrow a_k s_{k3} \end{aligned}$$

- 全配列 A の引用は，原則として
  $\mathtt{A}(a_1 l_1 + b_1 : a_1 u_1 + b_1 : a_1, \cdots, a_m l_m + b_m : a_m u_m + b_m : a_1)$
  に変換する．（引数としての引用など特定の条件下では変換しない場合がある．）

$a_k \neq 1$ のとき，配列の宣言形状は次元 k の方向に a_k 倍に拡大されるので，メモリ消費量を考えると等価変換とは言えないが，プログラムは意味的には等価である．fhpf ではそのような変換も行っている．さらに一般に，次元の交換（permutation）や，大きさ 1 の次元を追加する次元拡張を行うこともある．

ループインデックスのアフィン変換

ループインデックス空間のアフィン変換

$$i \longrightarrow ai + b$$

は，以下のような数式の基本操作で可能である．

- DO ループ

  ```
  DO I = e1,e2,e3
    ...
  END DO
  ```

  のループパラメタ e_1, e_2, e_3 について，

$$\begin{aligned} e_1 &\rightarrow a e_1 + b \\ e_2 &\rightarrow a e_2 + b \\ e_3 &\rightarrow a e_3 \end{aligned}$$

- ループ中に出現するすべての DO 変数 I について，

$$\mathtt{I} \rightarrow (\mathtt{I} - b)/a$$

  この除算は必ず割り切れるという特徴がある．

4.3 式の簡単化

4.3.1 トランスレータでの式の簡単化の特徴

トランスレータ方式のコンパイラにおける数式の簡単化は，以下のような点で一般の数式処理システムとは異なる．

1. 整数除算を含む式では，切捨て誤差を伴う．例えば `(1+3)/2` と `1/2+3/2` では値が違うので，分配則の適用などで切捨て誤差の違いが出ないことを保証する必要がある．
2. 浮動小数点型の式は，演算順序を変えるとユーザの意図しない計算誤差を生じることがある．また，十進表現と内部表現の変換・逆変換だけでも誤差が生じることがある．
3. EQUIVALENCE 文やポインタの効果によって，名前が異なる変数であっても同じ実体を共有する場合がある（alias 問題）．
4. 副作用を持つ関数を含む式では，関数の呼び出しの回数が変化するとプログラムの意味が変化する場合がある．
5. プログラムの実行部では，変数の値は一定ではない．例えば `K=3` という代入文があっても，後続の実行文でいつまでも `K` を 3 に置換できるとは限らない．（宣言部の範囲では変数の値は一定であると仮定できる．）

1. について，式の簡単化の規則に整数除算の特殊性を考慮する必要がある．システムが生成した式など，割り切れることがあらかじめ分かっている場合には，その情報を使って簡単化が進む場合もある．2. について，flops コンパイラでは，浮動小数点数はユーザ記述のまま文字列の形で保持し，コード出力に使っている．実数型や複素数型の式は，数式の簡単化の対象から外している．3. について，alias の可能性のある変数は特別な扱いをする必要がある．alias された変数（の部分）を同一視するか，それができない場合には最適化を抑止する．4. については，式中の関数呼び出し `f(...)` を一時変数 `tmp` に置換し，式の直前に代入文 `tmp=f(...)` を生成することで回避できる．fhpf ではそのような処理モジュールを使用している．5. については，フロー解析や定数伝播などの従来からのコンパイラ技術 [13] とうまく組み合わせることが望ましいが，アルゴリズムとして確立できていない．

4.3.2 fhpf における式の簡単化

コンパイラ内部では，様々な目的でプログラム中に出現する式を頻繁に変換する．式の簡単化を行わず 4.2.1 で紹介した式の基本操作を繰り返すと，数式はすぐに長く冗長なものになる．コンパイラが自動生成する式には，`(N+0)*1` のように 0 や 1 を含む trivial な式や，`max(N,0)` のようなちょっとした情報があればすぐに簡単化できる式が多い．我々の初期のコンパイラではこのような変換を重ねて非常に冗長な式を生成することが多かった．この問題は並列化コンパイラなどトランスレータ方式に特有の問題かもしれない．人が記述したプログラムではこのような trivial な演算による冗長性は少ないので，Fortran コンパイラなどで大きな問題になることはなかった．

一般に式の簡単化はコストの大きな処理だが，トランスレータ方式では強力な式の簡単化を最後に一度だけ行うより，軽い簡単化を頻繁に行うようにしたほうがよいと考えられる．この理由は以下の通りである．

- コンパイラの処理フローでは種々の局面で式の評価が必要になるので，式は常に簡単であることが望ましい．例えば，式の値が負でないと分かっていれば生成コードが簡単になるケースや，複数の配列アクセスで添字式の一致が明らかなら通信コードを生成しないで済

むケースなどがある．

- まめに簡単化を繰り返しておかないと，複雑な式になってから一度では簡単化できないことが多い．これは 4.3.1 で述べたトランスレータ方式での数式処理の難しさが関係している．

fhpf では，4.2.1 で述べた演算操作と同時に，演算種別に応じた軽い簡単化の規則を適用する．例えば整数型の式 e_1 と e_2 を足し合わせて新しい式を作るとき，加算に関する以下のような簡単化規則 reduceAdd(e_1, e_2) を試みる．この関数は，結果の式と，成功または失敗の状態を返す．

- e_2 が定数のとき，
  - e_1 も定数なら，値を加算し定数の Expr 構造体を作成し，成功として終了．
  - e_1 が $e_{11} + e_{12}$ のとき，
    - reduceAdd(e_{11}, e_2) が成功して e_3 となるなら，reduceAdd(e_3, e_{12}) を行い，成功として終了．
    - reduceAdd(e_{12}, e_2) が成功して e_3 となるなら，reduceAdd(e_{11}, e_3) を行い，成功として終了．
    - そうでない場合には，$e_1 + e_2$ に相当する式を生成し，失敗として終了．
- e_2 が $-e_{21}$ のとき，減算に関する簡単化規則 reduceSub(e_1, e_{21}) を行い，成功として終了．
- e_2 が $e_{21} + e_{22}$ のとき，reduceAdd(e_1, e_{21}) を行い，その結果の e_3 を使って，reduceAdd(e_3, e_{22}) を行い，成功として終了．
- e_2 が $e_{21} - e_{22}$ のとき，reduceAdd(e_1, e_{21}) を行い，その結果の e_3 を使って，reduceSub(e_3, e_{22}) を行い，成功として終了．
- その他の場合，$e_1 + e_2$ に相当する式を生成し，失敗として終了．

このような簡単化以外に，正規化処理，コード生成処理など，性能上重要な判断が行われる処理の直前のポイント数カ所で，プログラム全体に含まれる式の簡単化を行っている．

5 数式処理による正規化変換

この章では，第 4 章で述べた数式処理が実際に使用される有効な例として，我々が提案するプログラムの正規化のアルゴリズムを紹介する．

アルゴリズムは，マッピングの正規化（5.1 節），データの正規化（5.2 節），ループの正規化（5.3 節）の 3 つの手順で構成される．例として，図 3 に示した HPF プログラムに対して適用した様子を 5.4 節に示す．

5.1 マッピングの正規化

1. 分散の正規化
   (a) すべての分散について，分散の種類に応じて分散のパラメタを正規化する．
   - block 分散の場合，ブロック幅が指定されていない場合，ブロック幅 w を計算する式またはその結果の値を求めて保持する．

- 不均等 block 分散の場合，ブロック幅配列 $W(0:P-1)$ の領域を確保し値を設定する．
- 不規則分散の場合，マッピング配列 $M(0:N-1)$ の領域を確保し値を設定する．

(b) すべてのプロセッサ群について，各次元の上下限を正規化する．すなわち，プロセッサ群のある次元の下限が P_1，上限が P_2 のとき，下限を 0，上限を (P_2-P_1) に変換する．同時に，その次元に分散する不規則分散がある場合，M の各要素 $M(k)$ $(0 \le k < N)$ の値を，$(M(k)-P_1)$ に変換する．

2. 整列の正規化

(a) テンプレートに分散されていない次元があるとき，その次元についてテンプレートを縮退する．

(b) テンプレートの各次元の上下限を，下限が 0 になるようにシフトする．同時にそのテンプレート次元に整列している変数またはループは，テンプレートのシフトに合せて整列の定数項をシフトする．

(c) プロセッサ群に直接分散されている変数は，分散されている各次元について，下限を 0 とするテンプレートを生成し，そのテンプレートへの整列に変換する．

(d) プロセッサ群に直接分散されているループは，その分散を，下限値を 0 とするテンプレートへの整列に変換する．

(e) 変数 X に対して整列している変数またはループ Y は，整列の合成を繰り返し計算し，Y からテンプレート T への直接の整列になるよう変換する．
整列は線形結合によって合成することができる．X が T へ $i \to ai+b$ と整列し，Y が X へ $i \to ci+d$ と整列しているとすると，Y から T への整列のパラメタは以下のように計算できる．

$$a(ci+d)+b = aci+ad+b$$

すなわち，係数 ac，定数項 $(ad+b)$ となる．結果として，テンプレートに対して間接的に整列するすべての変数とループを，テンプレートへの直接の整列に変換することができる．

(f) テンプレートをできるだけ共通化する．

5.2 データの正規化

整列先 T に整列しているすべての変数について，以下の処理を行う．

1. T に整列している次元の宣言上下限を，T の宣言上下限に置き換える．
2. T への整列を恒等的（identical）な形（すなわち係数 $m=1$，定数項 $n=0$）に変換する．
3. T への整列が係数 m，定数項 n であるとき，その変数のすべてのアクセスについて，T に整列している次元の添字式 s を $(ms+n)$ に変換する．

5.3 ループの正規化

整列先 T に整列しているすべてのループについて，以下の処理を行う．

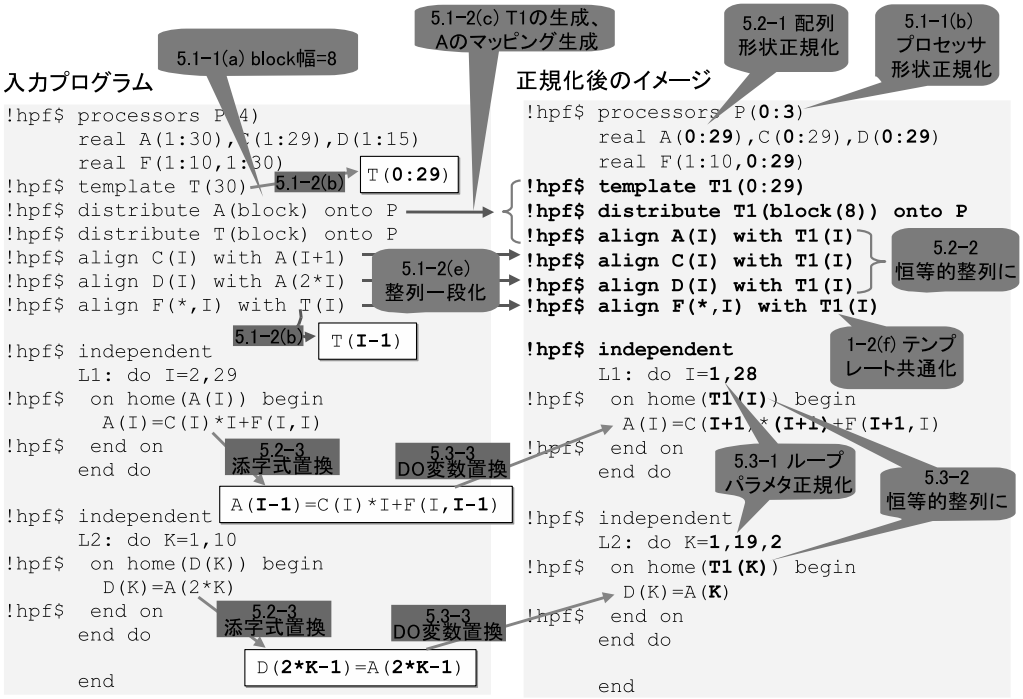


図 6: 正規化の例（コード変換イメージ）

1. 整列が係数 m，定数項 n であるとき，対象ループの初期値 I_1，終値 I_2，増分 I_3 を，それぞれ，(mI_1+n)，(mI_2+n)，mI_3 に変換する．
2. T への整列を恒等的（identical）な形（すなわち係数 $m=1$，定数項 $n=0$）に変換する．
3. そのループ内のループ変数 I の引用を $(\mathtt{I}-n)/m$ に変換する．

ループ変数の変換で生成される整数除算は割り切れることが分かっているので，それを前提に変換後の式を簡単化することができる．

5.4 正規化の例

本アルゴリズムを使用した場合の変換の様子を図 6 に示す．同図の左側が図 3 で示した HPF プログラムであり，この内部表現が本アルゴリズムの入力となる．右側が本アルゴリズムの出力に対応し，HPF 言語で表現している．図中の吹き出しにはその変換に対応する処理を節番号などと共に示している．

図 7 は，入力プログラムと正規化後のコードの，データとループの整列の様子を図示したものである．入力プログラムに表現された複雑な整列関係が，非常に単純な整列関係に正規化されることが分かる．

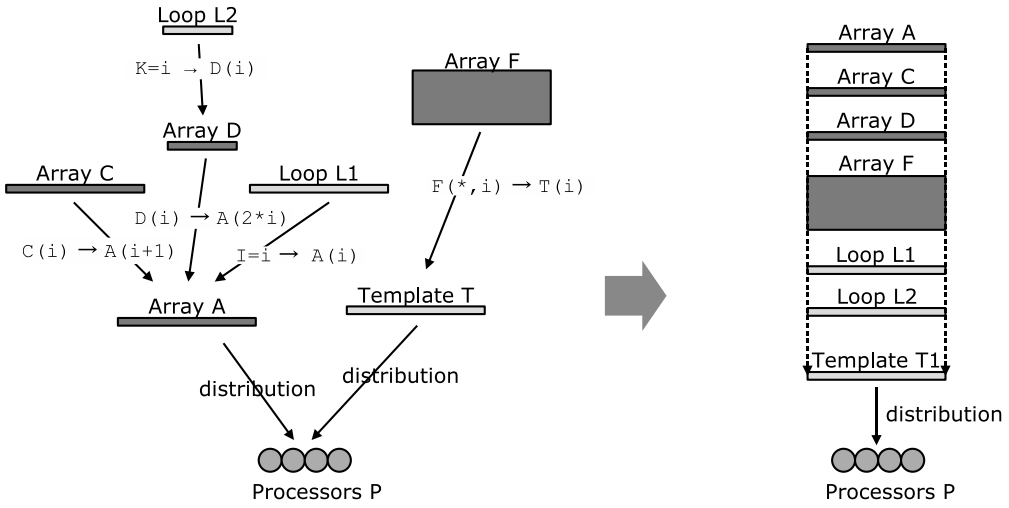


図 7: 正規化の例（整列の様子）

6 まとめ

整列と分散に関する正規化処理を実装することにより，コンパイラの最適化を促し，また，コード生成など後続パスの開発の負担軽減を果たした．結果として性能改善に繋がっている．

正規化処理は，フロー解析やデータ依存解析をベースとする従来のコンパイラ技術とは傾向の異なる技術であり，むしろ数式・記号処理の技術に近いものであると考えられる．また，正規化処理に代表される数式置換を基本とする変換は，数式処理の基本である式の簡単化を必ず必要とするため，その点でも数式処理技術が役に立つ．

今後の並列化コンパイラでは，プロセッサ台数だけでなく，クラスタ内・外の CPU 数や，計算と通信の比率など，実行時に評価が遅延されるパラメタが増える傾向にあり，コンパイル時には数式のまま扱う必要も増加すると考えられる．また，ユーザが記述する数式と比較して，システムが自動生成する数式には無駄な冗長性が多く生じる傾向があるので，代数的な式の簡単化が必要である．並列化コンパイラの分野では，数式処理技術の応用は今後さらに重要になって来ると思われる．

参考文献

参 考 文 献

[1] 岡部寿男，妹尾義樹，岩下英俊．次世代コンピュータに向けた技術課題と展望．**プラズマ・核融合学会誌**．Vol.80, No.5, pp.382-389. 2004 年

[2] High Performance Fortran Forum. *High Performance Language Specification Version 2.0*. 1997.

[3] High Performance Fortran Forum 著, 富士通, 日立, 日本電気訳, **High Performance Fortran 2.0 公式マニュアル**．シュプリンガー・フェアラーク東京．ISBN4-431-70822-7．1999 年.

[4] Hidetoshi Iwashita, Naoki Sueyasu, Sachio Kamiya, and Matthijs van Waveren. VPP Fortran and the Design of HPF/JA Extensions. *Concurrency and Computation: Practice and Experience*. Vol.14, pp575-588. John Wiley & Sons, Ltd. 2002.

[5] Hidetoshi Iwashita, Kohichiro Hotta, Sachio Kamiya, and Matthijs van Waveren. Toward a Lightweight HPF Compiler. *Lecture Notes in Computer Science*. Vol.2327, pp.526-538. 2002.

[6] *Message Passing Interface Forum*. http://www.mpi-forum.org/

[7] OpenMP Architecture Review Board. *OpenMP Fortran Application Program Interface Version 2.0*. November 2000. (http://www.openmp.org/specs/mp-documents/fspec20.pdf)

[8] OpenMP Architecture Review Board. *OpenMP C/C++ Application Program Interface Version 1.0*. October 1998. (http://www.openmp.org/specs/mp-documents/cspec10.pdf)

[9] 岩下英俊，進藤達也，岡田信．VPP Fortran: 分散メモリ型並列計算機向けプログラミング言語．**情報処理学会論文誌**．Vol.36, No.7．1995 年

[10] Co-Array Fortran (CAF). (http://www.co-array.org/)

[11] 日本規格協会．bf 日本工業規格プログラム言語 Fortran．1994 年.

[12] HPF 推進協議会ホームページ (http://www.hpfpc.org/)

[13] 中田育男．**コンパイラの構成と最適化**．朝倉書店．1999 年.