NEC

輸出する際の注意事項

本製品（ソフトウェアを含む）は，外国為替および外国貿易法で規定される規制貨物（または役務）に該当することがあります。

その場合，日本国外へ輸出する場合には日本国政府の輸出許可が必要です。

なお，輸出許可申請手続きにあたり資料等が必要な場合には，お買い上げの販売店またはお近くの当社営業拠点にご相談ください。

は　し　が　き

本書は,ベクトルエンジン向けNEC HPFコンパイラの使用法について説明しています。本書の最新版は，NEC Aurora Forumから入手可能です。

https://sxauroratsubasa.sakura.ne.jp/wiki/index.php?title=Special:WikiForum&forum=68

現在，スーパーコンピュータを必要とするような大規模科学技術計算プログラムでは，実行性能を得るため，あるいは大量のメモリを利用するために，プログラムの並列化が必要不可欠となっています。
ところが，特に分散メモリ型マルチプロセッサシステム上での 並列プログラムの開発は，データや処理のプロセッサへの割当てやデータ転送を，プログラマが明示的に記述しなければならないため，大変手間のかかる作業です。

High Performance Fortran (HPF)は， プログラマが，容易に並列プログラムを開発できるよう，Rice大学のKen Kennedy教授を中心に結成されたHPF フォーラム（HPFF)において， Fortran 95の拡張として策定された並列プログラミング言語です。
HPFFの活動は，1991年11月に開始され，早くも1993年5月にはHPF1.0がリリースされました。1993年11月には，細かな改良が施されたHPF1.1が定められました。 その後もHPFF2において検討が進められ， 安定した処理系の一刻も早い供給を促進するために， HPF1.1の仕様を縮小したHPF2.0が，1997年1月に定められました。 また，この規格書中には，HPF2.0では機能的に不十分な点に対応するため， HPF公認拡張仕様が定められています。

日本でも，1997年1月に，国内の主要なHPFコンパイラ開発者， スーパーコンピュータユーザから構成されたHPF合同検討会（JAHPF）というインフォーマルな検討会がスタートしました。1999年1月には，HPF2.0仕様とHPF2.0公認拡張の主要な機能 に加えて，プログラマが並列化やデータ転送をより詳細に制御できる 拡張機能を取り入れたHPF/JA1.0が策定されました。

本書における High Performance Fortran (HPF)言語の説明は，HPF フォーラムから公開されている以下の文書を参考にしています。

- High Performance Fortran Language Specification, High Performance Fortran Forum, November 10, 1994 Version 1.1
- High Performance Fortran Language Specification, High Performance Fortran Forum, January 31, 1997 Version 2.0

HPFの拡張仕様 HPF/JAの説明は，以下の文書を参考にしています。

- HPF/JA Language Specification, JAHPF (Japan Association for High Performance Fortran), January 31, 1999 Version 1.0
  English Version 1.0 November 11,1999

HPF や HPF/JA に関してさらに詳しい仕様をお知りになりたい方は，下記のサイトで公開されている文書をご参照ください。

- http://hpff.rice.edu/versions/
- http://site.hpfpc.org/home/former_hpfpc/gengo-shiyou

(注)上記情報は，2022 年 7 月現在のものです。

HPF および HPF/JA の言語仕様に関しては，シュプリンガー・フェアラーク東京社から 日本語版が刊行されています。

- High Performance Fortran 2.0 公式マニュアル

  著者: High Performance Fortran Forum

  編者: 財団法人 高度情報科学技術研究機構

  訳者: 富士通株式会社，株式会社日立製作所，日本電気株式会社

  発行: シュプリンガー・フェアラーク東京

  ISBN4-431-70822-7

関連説明書

本書を使用するにあたり，関連する説明書として次のものがあります。

- NEC Fortranコンパイラ(nfort)の使用法
  Fortranコンパイラユーザーズガイド（G2AF02）
- NEC MPIの使用法
  NEC MPIユーザーズガイド（G2AM01）
- FTRACE機能
  PROGINF/FTRACEユーザーズガイド（G2AT03）
- NQSVの使用法
  NEC Network Queuing System V(NQSV）利用の手引［操作編］(G2AD03)

商標，著作権について

用語定義・略語

用語・略語	説　明
ベクトルエンジン (VE，Vector Engine)	SX-Aurora TSUBASAの中核であり，ベクトル演算を行う部分です。PCI Expressカードであり，x86サーバーに搭載して使用します。
ベクトルホスト (VH，Vector Host)	ベクトルエンジンを保持するサーバー, つまり, ホストコンピュータを指します。
ホスト	VH または VE。
NQSV	SX-Aurora TSUBASAのジョブスケジューラ
VE番号	1つのVHに接続されているVEの識別番号。0から始まる連続した整数値です。
VH名	VHであるホストコンピュータのホスト名。
MPI	Message Passing Interfaceの略語です。主にノード間で並列コンピューティングを行うための標準化規格です。同一ノード内であっても，プロセス間のデータ転送にMPIを使用することが可能です。OpenMPとの併用も可能です。

目 次

表目次

図目次

第1章　　入門

1.1　HPF入門

1.1.1　HPF による分散メモリ型並列プログラミング

分散メモリ型の並列コンピュータシステム上で，プログラムを開発する場合，まず考慮しなければならないのは，次の 3 点です。

- データマッピング

  各データをどのプロセスのメモリ上に割り付けるかを決定する必要があります。これをデータマッピングと言います。

  リモートプロセスのメモリ上に割り付けられたデータのアクセスは，ローカルプロセスのメモリ上に割り付けられたデータのアクセスに比べて大きなオーバヘッドが伴うので，一つの処理に関連するデータは，できるだけ同じプロセスのメモリ上に割り付ける必要があります。

- 処理の分担（計算マッピング）

  演算，代入，および分岐といった処理を，どのプロセスが実行するのかを決定する必要があります。これを計算マッピングと言います。一般に，N 個のプロセスを利用して，N 倍の Speed Up を実現するためには，各プロセスが均等に処理を分担するとともに，各プロセスの処理が同時並行的に行える必要があります。

- データ転送

  例えば，ある処理を行うプロセスと，その処理に必要なデータが割り付けられているプロセスが異なる場合，リモートプロセスのメモリ上にあるデータには直接アクセスできないので，図 1 のように，必要なデータを転送する必要があります。

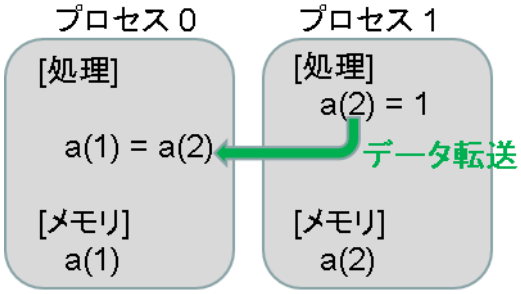

図 1　データ転送

さらに，データ転送の開始または完了のタイミングを待ち合わせるために，プロセス間での同期も必要です。

MPI を用いて並列プログラムを記述する場合のように，これら全てのことを考慮にいれた上で，効率の良い 並列プログラムを作成することは，大変に手間のかかる作業です。 HPF の基本的な考え方は，プログラマには，データマッピングだけを指定してもらい，HPF コンパイラは，それに基づいて，自動的に処理の分担を決め，その際に必要となるデータ転送やプロセッサ間の同期，といった処理も全て自動的に行う，というものです。 そのため，プログラマは，データがローカルプロセスのメモリ上 にあるか，リモートプロセスのメモリ上にあるか，といった 区別をせずに， 直接全てのデータを参照できるかのようなプログラミングが可能です。このようなプログラミングモデルを，HPF では グローバルモデルと呼びます。

HPF コンパイラは，主として，並列化可能なループの繰返しを各プロセスに割り当てることで並列化を行います。その際，プログラマが指定した データマッピングに基づいて，データアクセスができるだけプロセス内に閉じるように，処理の分担を決定します。 従って，HPF プログラミングにおいて，プログラマが行う主要な作業は，配列を，並列化可能なループでアクセスされる次元で，プロセス間に分割配置することです(図 2 参照)。

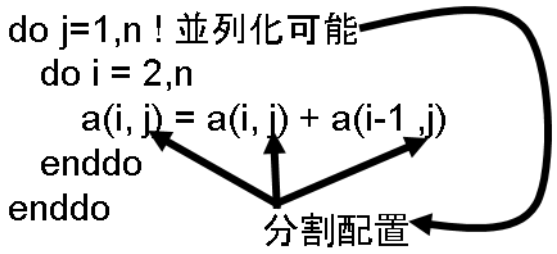

図 2　HPF プログラミング

1.1.2　HPF プログラムの例

次の Fortran プログラムは，2 つの配列 a および b の和を配列 c に代入しています。

```
real a(10), b(10), c(10)
    :
do i=1, 10
   c(i) = a(i) + b(i)
enddo
```

図 3　Fortran プログラム例

このままでも，HPF コンパイラでコンパイル・実行することは可能です。しかし，このままでは，HPF コンパイラは，全く並列化を行いません。 HPF コンパイラは， HPF 指示文が指定されていない配列は，全てのプロセスそれぞれに，配列全体を割り付けます。そして，できるだけ各プロセス上に割り付けられているデータだけをアクセスするように処理の分担を決定するので，結局，全プロセスが，全ての処理をそれぞれ実行することになり，何個のプロセスで実行しても，1 個のプロセスで

実行する以上の性能は得られないことになります。

このプログラムを HPF によって並列化するためには，配列をどのようにプロセス間に分割配置するかを指定する必要があります。図 4 の 2 行目は，配列をプロセス間に分割配置する最も基本的な指示文である DISTRIBUTE 指示文です。配列 a, b, および c をプロセス間に均等に分割配置することを指定しています。

```
      real a(10), b(10), c(10)
!HPF$ DISTRIBUTE (BLOCK) :: a, b, c
        :
      do i=1, 10
        c(i) = a(i) + b(i)
      enddo
```

図 4　HPF プログラム例

このプログラムを HPF コンパイラでコンパイル・リンクし，2 個のプロセス 0 および 1 で実行すると，各配列は均等に 2 分割され，前半部分がプロセス 0 に，後半部分がプロセス 1 に割り付けられます。そして，できるだけ各プロセス上に割り付けられているデータだけをアクセスするように，ループの前半部分をプロセス 0 が，後半部分をプロセス 1 が実行するよう処理が分担され，うまく並列実行できます。

このように，HPF プログラミングの基本的な作業は，逐次 Fortran プログラムに，Fortran コンパイラには注釈行と扱われる HPF 指示文を挿入して，データマッピングを指定することです。

1.1.3　HPF の仕様

HPF の仕様は，HPF2.0 仕様，HPF 公認拡張仕様，および HPF/JA 拡張仕様から構成されています。 また，いくつかの独自拡張機能も利用することができます。 これらを機能面で分類すると，大きく分けて，次の 3 つに分類できます。

- データマッピング関連の指示文
  配列のプロセスへどのように分割配置するかを指定します。HPF の中心となる機能です。

- 計算マッピングとデータ転送関連の指示文
  HPF コンパイラは，並列化可能なループを，並列化可能であると解析できなかったり，最適な処理の分担が選択できなかったり，データ転送の必要がないのに，データ転送を生成してしまうことがあります。
  このような場合，プログラマが，コンパイラに対して，ループが並列化可能であることを教え

たり（INDEPENDENT 指示文），最適な処理の分担を指示したり（ON-HOME 指示文），データ転送が必要ないことを明示したり（LOCAL 節）することにより，性能向上を図るための機能が用意されています。

- その他の機能
  NUMBER_OF_PROCESSORS()に代表される組込み手続，マッピング問合せ および 配列演算のためのライブラリ群，ならびに HPF から HPF 以外の手続を呼び出すための外来手続機能などが用意されています。

HPF 指示文の使い方は，第 4 章で解説します。各指示文の正確な仕様に関しては，High Performance Fortran Language Specification および HPF/JA Language Specification を参照してください。

1.2　NEC HPF コンパイラ入門

1.2.1　HPF プログラムのコンパイル・リンク

HPF プログラムをコンパイル・リンクするには，HPF ソースプログラムに対して HPF コンパイルコマンド ve-hpf を実行します。HPF コンパイルコマンド ve-hpf を実行すると，図 5 のように，HPF ソースプログラムから，並列化された HPF 実行プログラムが生成されます。NEC HPF コンパイラを使用するためには，NEC SDK 中の NEC Fortran コンパイラ(version 3.0.7 以降) および NEC MPI が必要です。HPF プログラムのコンパイル・リンク方法の詳細は，第 2 章を参照してください。

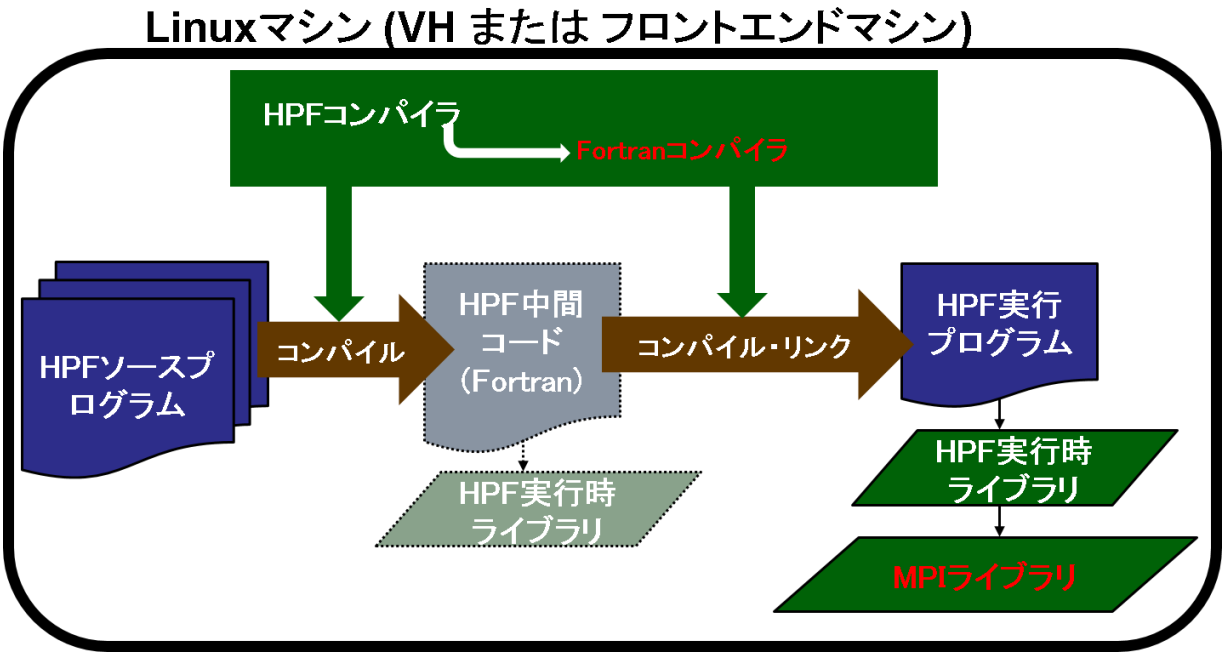

図 5　HPF プログラムのコンパイル・リンク

1.2.2　HPF プログラムの実行

HPF 実行プログラムは，実質的には MPI ライブラリの呼出しを含む MPI 実行プログラムです。従って，MPI プログラムと全く同様に，mpirun コマンド または mpiexec コマンドで実行します。実行方法の詳細は，第 3 章を参照してください。

1.2.3　注意制限事項

- 書式付き入出力文の実行性能は，十分にチューンされているとはいえません。従って，大きなサイズのデータを入出力する場合，可能な限り書式なし入出力を使用してください。入出力の高速化については，5.1.4 項で説明している図 103，図 106，図 109 及び 図 113 のような方法もご検討ください。
- 派生型は，シャドウ領域を宣言する用途に限定して使用できます。 そのため，1 つの整数型一次元配列を成分とする派生型配列だけを記述することができます。また派生型配列を分割配置することはできません。
- DATA 文中に，派生型配列を記述することはできません。
- DATA 文中に，配列構成子を成分としてもつ派生型構成子を記述することはできません。
- ポインタ仮配列の特性を，他の変数を宣言するために使用することはできません。例えば，組込み関数 LBOUND, UBOUND, または SIZE の引数として，次のように引用することはできません。

```
subroutine sub(p)
integer, pointer :: p(:,:)
integer, dimension(lbound(p,1):ubound(p,1), size(p,2)) :: a   ! ポインタ仮引数の引用
```

- 名前付き多次元配列定数は，初期値式中で使用することはできません。 特に，以下のような文脈での使用はできません。
  - CASE 文の場合式中
  - 宣言文の種別パラメタ中
  - 組込み手続の KIND 引数中
  - PARAMETER 文 または 宣言文中の初期値式中。例えば，次のような記述はできません。

```
integer, parameter, dimension(2,2) :: x = reshape((/1,2,3,4/), (/2,2/))
integer, parameter :: y = x(1,2)          ! 初期化式中の名前付き多次元配列定数
```

- モジュール中で宣言された名前付き配列定数は，参照結合により初期値式中で利用することはできません。特に，モジュール中で宣言された名前付き配列や派生型配列は，次の場所に記述

することはできません。

- CASE 文の場合式中
- 宣言文の種別パラメタ中
- 組込み手続の KIND 引数中
- PARAMETER 文 または 宣言文中の初期値式中

第2章　　HPF プログラムのコンパイル・リンク

本章では，HPF プログラムのコンパイル・リンク方法を説明します。

2.1　HPFプログラムのコンパイル・リンク

最初に，NEC MPI および NEC Fortran コンパイラの環境設定を行うため，次のコマンドを実行して，MPI セットアップスクリプトを読み込んでください。この設定は VH からログアウトするまで有効です。ログアウトすると無効となりますので，VH にログインするたびに再実行してください。

(bash の場合)
%> **source /opt/nec/ve/mpi/*{version}*/bin/necmpivars.sh**
(csh の場合)
%> **source /opt/nec/ve/mpi/*{version}*/bin/necmpivars.csh**

上記の *{version}* は,ご使用になる NEC MPI のバージョンに対応するディレクトリ名です。例えば，NEC MPI バージョン 2.5.0 を bash 上で使用する場合，次のコマンドを実行します。

(バージョン 2.5.0 および bash の場合)
%> **source /opt/nec/ve/mpi/2.5.0/bin/necmpivars.sh**

詳細は，NEC MPI ユーザーズガイドを参照してください。

上記の環境設定後， 次のように HPF コンパイルコマンド ve-hpf を実行して，HPF プログラムの・コンパイル・リンクを行います。

%> **ve-hpf**　[<options>]　<sourcefiles>　[<options>]

ここで，

- <options>は，コンパイラオプションです。コンパイラオプションとしては，HPF コンパイラオプション，ならびに 主要な NEC Fortran コンパイラオプションおよび NEC MPI コンパイラオプションを指定できます。
- {sourcefiles} は，HPF ソースプログラムです。
- []内の記述は，省略可能です。

2.2　ファイル名規約

2.2.1　入力ファイル

HPF コンパイラは，入力ファイルを，そのサフィックスによって表 1 のように処理します。

表 1　入力ファイルのサフィックス

サフィックス	処理
.hpf	固定形式の HPF ソースプログラムとしてコンパイルします。
.f	固定形式の HPF ソースプログラムとしてコンパイルします。
.F	前処理後，固定形式の HPF ソースプログラムとしてコンパイルします。
.for	固定形式の HPF ソースプログラムとしてコンパイルします。
.f90	自由形式の HPF ソースプログラムとしてコンパイルします。
.F90	前処理後，自由形式の HPF ソースプログラムとしてコンパイルします。
.f95	自由形式の HPF ソースプログラムとしてコンパイルします。
.F95	前処理後，自由形式の HPF ソースプログラムとしてコンパイルします。
.o	オブジェクトファイルとして，リンク対象とします。
.a	オブジェクトファイルのライブラリとして，リンク対象とします。

2.2.2　出力ファイル

HPF コンパイラは，入力ファイルやコンパイラオプションに応じて，表 2 のようなサフィックスのファイルを出力します。HPF コンパイラが出力する HPF 実行ファイルの名前は，コンパイラオプション-o により指定できます。既定値では，a.out です。

表 2　出力ファイルのサフィックス

サフィックス	説明
.d	コンパイル過程で生成される静的データ初期化ファイルです。HPF コンパイラオプション-Mkeepstatic を指定した場合，保存されます。
.f	HPF 実行時ライブラリ呼出しを含む Fortran 中間ソースファイルです。HPF コンパイラオプション-Mftn または –Mkeepftn が指定された場合，保存されます。

.mod	モジュールを含む HPF ソースファイルに対して出力されるモジュールファイルです。
.o	オブジェクトファイルです。

2.3　コンパイラオプション

本節では,HPF コンパイルコマンド ve-hpf に指定可能なコンパイラオプションについて説明します。表 3 に, 共通コンパイラオプションを示します。共通コンパイラオプションは, HPF コンパイラ　および　バックエンドの Fortran コンパイラの両方に対して効果をもちます。表中の[]で囲まれた記述は，省略可能であることを示します。

表 3　共通コンパイラオプション

オプション	サブオプション	機能
-c		翻訳終了後，結合処理を行わずオブジェクトファイルを出力します。
-D	*name*[=*value*]	プリプロセッサマクロ *name* を定義します。 =*value* が指定された場合，プロプロセッサマクロ *name* の値を *value* で定義します。
-E		プリプロセッサにより前処理を行った結果を標準出力に出力します。コンパイル・リンクは行いません。
-F		プリプロセッサにより前処理を行った結果をサフィックスが.f のファイルに出力します。コンパイル・リンクは行いません。
-I	ディレクトリ名	INCLUDE 行に指定されたファイル および モジュールファイルが存在するディレクトリを指定します。
-L	ディレクトリ名	リンク時に必要なライブラリを，指定したディレクトリから検索することを指定します。本オプションを複数指定した場合，指定された順にディレクトリを検索します。
-l	ライブラリ名	リンク時に必要なライブラリ名を指定します。本オ

		プションを複数指定した場合,指定された順に ライブラリを検索します。
-O	HPF コンパイラおよびバックエンドの Fortran コンパイラによる最適化レベルを指定します。	
	0	HPF コンパイラによる最適化なし。 バックエンドの Fortran コンパイラには，-O0 を渡します。
	1	HPF コンパイラによる最適化なし。 バックエンドの Fortran コンパイラには，-O1 を渡します。
	2	HPF コンパイラによる最適化なし。 バックエンドの Fortran コンパイラには，-O2 を渡します。
	3	HPF コンパイラによる最適化あり。 バックエンドの Fortran コンパイラには，-O3 を渡します。
	4	HPF コンパイラによる最適化あり。 バックエンドの Fortran コンパイラには，-O4 を渡します。
-o	ファイル名	生成する HPF 実行プログラム名を指定します。
-U	*name*	プリプロセッサマクロ *name* を未定義にします。
-V		ドライバ および コンパイラのバージョンを表示します。
-v		HPF コンパイラ，バックエンドの Fortran コンパイラ，および リンカの起動状況を表示します。

表 4 に HPF コンパイラオプションを示します。 これらのオプションは，それぞれ-M に続けて指定しなければなりません。 またサブオプションは，オプションに続けて指定しなければなりません。

表 4　HPF コンパイラオプション

オプション	サブオプション	機能
allow_nfort_cncall		Fortran コンパイラ指示行 cncall を有効にします。ただし，並列化されたループ中で呼出される手続中でデータ転送が発生した場合，プログラムの動作は保証されないこ

		とに注意してください。
allow_nfort_paralleldo		Fortran コンパイラ指示行 parallel do を有効にします。ただし，Fortran コンパイラにより並列化されたループ中でデータ転送が発生した場合，プログラムの動作は保証されないことに注意してください。
autodist	配列の分散を指定します。 以下に示すサブオプションを省略した場合，全ての配列を最後の次元で BLOCK 分散します。 ● COMMON 文と NAMELIST 文 の両方に出現する配列を分散することはできません。 分散した場合，プログラムの動作は保証されません。 ● 次のような配列は分散されません ・ DISTRIBUTE 指示文，ALIGN 指示文，INHERIT 指示文，または DYNAMIC 指示文中に出現する配列。 ・ SEQUENCE 指示文中に出現する配列 ・ PARAMETER 文，EQUIVALENCE 文，または NAMELIST 文中に出現する配列 ・ 文字型 または 派生型の配列 ・ POINTER 属性 または TARGET 属性をもつ配列 ・ 大きさ引継ぎ配列 ・ LOCAL 手続中の仮引数以外の配列 ● 共通ブロック または 引用仕様宣言を使用するプログラムの場合，1 つの実行プログラムを構成する全てのファイ ルに対して同一のオプションを指定する必要があります。 ● 各配列の分散は，HPF コンパイラオプション -Minform=inform を指定することにより表示できます。 ● 指定した分散が適切かどうかは，HPF コンパイラオプション -Mlist2 により出力される 並列化情報ファイルにより判断することができます。	
	=all[:*b*]	全ての配列を分散します。

		2 進整数 *b* は，その各ビットが，最下位ビットから順に， 配列の各次元に最後の次元から最初の次元へと一対一に対応し，1 のビットに対応する次元が BLOCK 分散されます。省略時は，最後の次元だけが BLOCK 分散されます。
	=rank*:a*[:*b*]	*a*次元配列を分散します。*a*は，整数値です。2 進整数 *b* は，その各ビットが，最下位ビットから順に， 配列の各次元に最後の次元から最初の次元へと一対一に対応し，1 のビットに対応する次元が BLOCK 分散されます。省略時は，最後の次元だけが BLOCK 分散されます。
backslash nobackslash		引用符で囲まれた文字列のバックスラッシュを通常の文字として扱いエスケープ文字としません。
		引用符で囲まれた文字列のバックスラッシュをエスケープ文字として扱います。[既定値]
chkhome		ループの制御変数が始値から終値まで変化するとき，配列の宣言の範囲外の参照がないことが静的に判明する配列のみをループ並列化の 基準配列(HOME 配列)に選択することを指定します。 ループの実行時に，配列の宣言範囲外への参照が発生した場合，invalid alignment の実行時内部エラーが発生しますが， 本オプションはそれを防ぐ効果を持ちます。 ただし，結果として性能の低下を招く場合があります。 詳細は， INDEPENDENT DO ループの並列化の節の注 を参照してください。

commonchk		実行時に，共通ブロック変数の宣言における，手続間での矛盾を検査します。 本オプションは，1 つの実行プログラムを構成する全てのプログラム単位に対して指定する必要があります。 本オプションと，オプション-Mnoentry または-Mnoerrline を同時に使用することはできません。 同時に指定された場合，後に指定されたオプションの機能だけが有効となり ます。
cprop		定数伝搬最適化を促進します。 但し，最適化の結果，除算の分母がゼロになる場合，バックエンドの Fortran コンパイラにより，コンパイルエラーが検出される場合があります。
nocprop		除算の分母に対する定数伝搬最適化を行いません。(既定値)
dclchk		全ての変数の宣言を必須とします。
nodclchk		変数の宣言を必須としません。(既定値)
dintrin		次の Fortran の組込み手続，HPF ライブラリ手続，または HPF ローカルライブラリ手続が引用された場合，結果の型が 8 バイト整数型であるような，対応する 拡張組込みまたはライブラリ手続が引用されたものと見なします。 ● Fortran 組込み手続 COUNT，LBOUND，MAXLOC，MINLOC，SHAPE，SIZE，または UBOUND ● HPF ライブラリ手続 COUNT_PREFIX，COUNT_SCATTER，COUNT_SUFFIX, GRADE_DOWN, または GRADE_UP，

		● HPF ローカルライブラリ手続 GLOBAL_SHAPE, GLOBAL_SIZE, LOCAL_BLKCNT , LOCAL_LINDEX , または LOCAL_UINDEX 拡張組込み／ライブラリ手続の仕様に関しては， 拡張組込み／ライブラリ手続の節 および HPF LOCAL ライブラリの節 を参照してください。
dlines		固定形式の場合，ソース内の 1 文字目が'*', 'd', または'D'で始まる行をコメント文として扱わず有効な文として処理します。 自由形式の場合,"!!"で始まる行をコメント文として扱わず有効な文として処理します。
nodlines		固定形式の場合，ソース内の 1 文字目が'*', 'd', または'D'で始まる行をコメント文として扱います。 自由形式の場合,"!!"で始まる行をコメント文として扱います。(既定値)
extend		ソースの 1 行分として許す文字数を 2048 文字にします。
f90		Fortran90 言語で記述された手続として，コンパイルを行います。 明示的に外来種別が指定されている手続に対しては， 効果を持ちません。
fixed		固定形式の入力ソースを受け付けます。
free[form]		自由形式の入力ソースを受け付けます。
nofree[from]		固定形式の入力ソースを受け付けます。(既定値)
ftn		HPF プログラムソースから HPF 実行時ライブラリ呼出しを含む Fortran の中間ソースフ

		ァイルを出力します。中間ソースファイルのサフィックスは，.f です。バックエンドの Fortran コンパイラによるコンパイルは行いません。
fullref		部分 REFLECT 指示文において,部分的なシャドウの指定を無視して，常に, 全てのシャドウ実体に対して，対応するデータ実体の値を反映させます。 シフト転送のパタンが毎回変化するような場合，シャドウ実体の一部だけにデータ転送を行うよりも，データ転送スケジュールの再利用により高速になる場合があります。 このオプションを指定する場合，一つの実行プログラムを構成する全ての手続に対して，一貫して指定する必要があることに注意してください。
g		-Mkeepftn オプションを有効にし，バックエンドの Fortran コンパイラを-g オプション付きで起動します。
hpfout		データ分散指定オプション-Mautodist で指定した DISTRIBUTE 指示文が挿入された HPF ソースプログラムを生成します。生成される HPF ソースプログラムのサフィックスは.hpf.src です。
info		ループの並列化に関するコンパイル情報を作成し，標準エラー出力に出力します。
inform	出力する診断メッセージのレベルを指定します。	
	=fatal	fatal レベルの診断メッセージだけを出力します。
	=severe	severe および fatal レベルの診断メッセージを出力します。

	=warn	warn，severe，および fatal レベルの診断メッセージを出力します。(既定値)
	=inform	全てのレベルの診断メッセージを出力します。
keepftn		HPF 実行ファイルに加えて，HPF 実行時ライブラリ呼出しを含むFortranの中間ソースファイルを出力します。中間ソースファイルのサフィックスは，.f です。
keepstatic		実行可能ファイルに加えて，コンパイル過程で生成される静的データ初期化ファイルを出力します。静的データ初期化ファイルのサフィックスは，.d です。
list		リストファイルを作成します。リストファイルのサフィックスは.lst です。
list2		分散並列化およびデータ転送の情報を含む並列化情報リストファイルを作成します。並列化情報リストファイルのサフィックスは.lst です。 バックエンドのFortranコンパイラの編集リスト出力オプション-report-all または –report-format を同時に指定すると，ベクトル化と共有並列化の情報もマージされます。ただし，この場合，バックエンドの Fortran コンパイラが出力するメッセージには，HPF コンパイラにより並列化された中間ソースの行番号が表示されることに注意してください。
list3		分散並列化，データ転送，および中間ソースの情報を含む並列化情報リストファイルを作成します。並列化情報リストファイルのサフィックスは.lst です。 バックエンドのFortranコンパイラの編集リ

		スト出力オプション-report-all または –report-format を同時に指定すると，ベクトル化と共有並列化の情報もマージされます。ただし，この場合，バックエンドの Fortran コンパイラが出力するメッセージには，HPF コンパイラにより並列化された中間ソースの行番号が表示されることに注意してください。
nolist		リストファイルを作成しません。（既定値）
local		LOCAL モデルの手続として，コンパイルします。 明示的に外来種別が指定されている手続に対しては，効果を持ちません。
noentry		実行時エラーメッセージ用の情報を一切出力しません。そのため，プログラムの実行性能が向上する場合があります。 ただし，実行時エラーが発生した場合，プログラムの動作は 保証されません。そのため，正常に実行できることを確認ずみのプログラムに対してだけ，本オプションを指定することを強く推奨します。 なお，本オプションとオプション-Mcommonchk，-Mprof，または -Msubchk を同時に使用することはできません。 同時に指定された場合，後に指定されたオプションの機能のみが有効となり ます。
noerrline		実行時エラーメッセージ用の行情報を出力しません。そのため，プログラムの実行性能が向上する場合があります。 ただし，実行時エラーが発生した場合，エラーの原因となった行情報を得ることはできません。

		本オプションとオプション-Mcommonchk，-Mprof，または -Msubchk を同時に使用することはできません 同時に指定された場合，後に指定されたオプションの機能のみが有効となり ます。
nogenblock		全ての GEN_BLOCK 分散を BLOCK 分散として扱います。
noindependent		全ての INDEPENDENT 指示文を無効にし，自動分散並列化のみを行います。
nolocal		LOCAL 節を無効にします。
nomapnew		INDEPENDENT ループ中の暗黙的に分割配置された集計変数ではない配列を NEW 変数として扱います。
overlap	**=size:***n*	BLOCK 分散または GEN_BLOCK 分散された次元に対するシャドウ領域の幅を *n* に指定します。 既定値は 4 です。
preprocess		プリプロセッサによる前処理を実行します。
r8		基本実数型の変数 および 定数を倍精度実数型に基本複素数型の変数 および 定数を倍精度複素数型に変換します。
recursive		再帰呼出しを許可するコードを生成します。局所変数は静的なメモリの代わりにスタック上に置かれます。本オプションの使用は性能を低下させる原因となる可能性もあります。また，入出力を行う手続や SAVE 変数または COMMON 変数を確定する手続は再帰の対象とはなりません。
res2local		全ての RESIDENT 節は，LOCAL 節とみなされます。
scalarnew		INDEPENDENT ループ中の全てのスカラ

		変数を NEW 変数として扱います。ただし，REDUCTION 変数はこの対象となりません。
sequence		全ての変数が SEQUENCE 属性の変数として扱われます。これらは連続的に記憶領域に確保されます。
nosequence		SEQUENCE 指示文による指定があるか，大きさ引継ぎ配列，または EQUIVALENCE 文中に出現する変数以外は，全て SEQUENCE 属性をもたない変数として扱われます。(既定値)
serial		SERIAL モデルの手続として，コンパイルします。 明示的に外来種別が指定されている手続に対しては， 効果を持ちません。
subchk		実行時に，配列の各次元の添字が宣言範囲内にあるか否かの検査を行います。 ただし，LOCAL 手続は検査の対象となりません。 本オプションとオプション-Mnoentry，または -Mnoerrline を同時に使用することはできません。 同時に指定された場合，後に指定されたオプションの機能のみが有効となります。
upcase		ソースファイル中の大文字・小文字の区別を行います。 例えば，識別子"X"と"x"は，異なる識別子として扱われ， キーワードは小文字でなければなりません。
noupcase		ソースファイル中の大文字・小文字を区別しません。(既定値)

2.3.1　NEC Fortran コンパイラ指示行

主要な NEC Fortran コンパイラ指示行を指定できます。ただし，次の指示行を除きます。

cncall, forced_collapse, loop_count(n), option, outerloop_unroll(n), parallel do

また，vreg 指示行は，実行部 または 宣言部の最後にだけ指定できます。

詳細は，NEC Fortran ユーザーズガイドを参照してください。

2.3.2　NEC Fortran コンパイラオプション

共通コンパイラオプションの他，主要な NEC Fortran コンパイラオプションを指定できます。ただし，次の NEC Fortran コンパイラオプションは，指定できません。詳細は，NEC Fortran ユーザーズガイドを参照してください。

-S, -cf, -clear, -fsyntax-only, -x, @<file-name>, -fivdep, -floop-count=n, -fopenmp, -pthread, -fdefault-integer=n, -fdefault-double=n, -fdefault-real=n, -fextend-source, -ffree-form, -ffixed-form, fmax-continuation-lines=n, -realloc-lhs, -frealloc-lhs-array, -frealloc-lhs-scalar, -std=standard, -use, -w, -report-file, -report-append-mode, -[no-]report-cg, -[no-]report-diagnostics,-[no-]report-inline, -[no-]report-vector, -dD, -dl, -dM, -fpp,-nofpp, -fpp-name, -dN, -E, -H, -I-, -M, -MD, -MF <filename>, -MP, -MT <target>, -fpp, -nofpp, -fpp-name, -isysroot, -isystem, -nostdinc, -P, -Wp, -Bdynamic, -Bstatic, -static, -shared, --sysroot, -B, -fintrinsic-modules-path, -module, -J, --help, -print-file-name, -print-prog-name, -noqueue, --version

NEC Fortran コンパイラオプション-report-all または -report-format を指定した場合，Fortran コンパイラが出力する編集リストには，HPF コンパイラにより並列化された中間ソースが出力され，メッセージには，中間ソースにおける行番号が表示されることに注意してください。この時，HPF コンパイラオプション-Mlist2 または -Mlist3 を同時に指定すると，HPF コンパイラによる並列化情報リスト中に，ベクトル化と共有並列化の情報もマージされます。

2.3.3　NEC MPI コンパイラオプション

NEC MPI コンパイラオプション-mpiprof，-show，-ve，-static-mpi，および –shared-mpi が指定できます。詳細は，NEC MPI ユーザーズガイドを参照してください。

2.4　環境変数

本節では，コンパイル時に指定できる環境変数を説明します。

- VE_HPF_COMPILER_PATH

標準のパスである/opt/nec/ve/bin/ve-hpf 以外のパスにある HPF コンパイラを使用する場合，この環境変数を使用すると，パスの指定を省略できます。例えば，/opt/nec/ve/hpf/1.0.0/bin/ve-hpf を使用する場合，次のように指定してください。

(bash の場合)

```
%> export VE_HPF_COMPILER_PATH=/opt/nec/ve/hpf/1.0.0
%> export PATH=${VE_HPF_COMPILER_PATH}/bin:$PATH
%> ve-hpf
```

第3章　　HPF プログラムの実行

本章では，HPF プログラムの実行方法を説明します。

3.1　HPFプログラムの実行

NEC MPI および NEC Fortran コンパイラの環境設定を行うため，最初に，次のコマンドを実行して，MPI セットアップスクリプトを読み込んでください。この設定は VH からログアウトするまで有効です。ログアウトすると無効となりますので，VH にログインするたびに再実行してください。

(bash の場合)
%> **source /opt/nec/ve/mpi/*{version}*/bin/necmpivars.sh**
(csh の場合)
%> **source /opt/nec/ve/mpi/*{version}*/bin/necmpivars.csh**

上記の *{version}* は,ご使用になる NEC MPI のバージョンに対応するディレクトリ名です。例えば，NEC MPI バージョン 2.5.0 を bash 上で使用する場合，次のコマンドを実行します。

(バージョン 2.5.0 および bash の場合)
%> **source /opt/nec/ve/mpi/2.5.0/bin/necmpivars.sh**

HPF 実行プログラムの実行は，MPI 実行プログラムと同様に，MPI 実行コマンド mpirun または mpiexec を次のように実行します。

%> **mpirun** [{mpioptions}] {hpfexec} [{args}] [**-hpf** {hpfoptions}]

%> **mpiexec** [{mpioptions}] {hpfexec} [{args}] [**-hpf** {hpfoptions}]

ここで，

- {mpioptions} は，NEC MPI 実行時オプションです。
- {hpfexec}は，HPF 実行指定です。HPF 実行指定には，HPF 実行プログラム{hpfexecfile} または HPF 実行プログラムを実行するシェルスクリプトが指定できます。HPF 実行指定は，MPI 実行コマンドに 1 つしか指定できないことに注意してください。
- {args}は，省略可能な HPF 実行プログラムに対する引数です。
- {hpfoptions}は，省略可能な HPF 実行時オプションです。
- []内の記述は省略可能です。

3.2　実行時オプション

表 5 は，HPF 実行時オプションです。HPF 実行時オプションは，-hpf の後に指定する必要があります。次の例では，HPF 実行プログラム a.out の実行時に，HPF 実行時オプション-version を指定しています。

```
%> mpirun –np 2 ./a.out -hpf -version
```

次のように環境変数 HPF_OPTS を使用して，-hpf による指定と同じ指定が可能です。ただし，環境変数による指定よりも，-hpf による指定のほうが優先されます。

```
%> setenv HPF_OPTS “-version”
```

表 5　HPF 実行時オプション

実行時オプション	環境変数	機能
-commmsg	HPF_COMMMSG	手続の境界でデータ転送が発生する場合，警告メッセージを出力します。
-maxxfer [*n*]	HPF_MAXXFER [*n*]	データ転送時に使用する領域の最大の大きさ *n* を設定します。単位は MB で,範囲は 16～1024 です。 既定値は，32 MB です。
-no_stop_message	HPF_NO_STOP_MESSAGE	並びをもたない STOP 文が実行された場合，既定値の STOP のメッセージ出力を 抑制します。
-subchk [warn|fatal]	HPF_SUBCHK [warn|fatal]	HPF コンパイラオプション-Msubchk を指定してコンパイルされている場合， 実行時に配列の各次元の添字が宣言範囲内にあるか否かの検査を行います。 検査の際，宣言範囲外の添字を検出した時点でプログラムを強制終了させるか， 警告メッ

		セージを出力した上でプログラムを続行するかを設定します。省略可能な引数は，次のとおりです。 **warn** 添字が宣言範囲外であるとき，警告メッセージを出力した上でプログラムを続行します。[既定値] **fatal** 添字が宣言範囲外であるとき，プログラムを強制終了させます。
-version	**HPF_V**	実行時ライブラリのバージョン情報を出力します。
-V	**HPF_VERSION**	実行時ライブラリのバージョン情報を出力します。
-zmem [yes|no]	**HPF_ZMEM** [yes|no]	割付け配列や分割配置された配列のために動的に確保したメモリ領域を 0 に初期化するかどうかを指定します。省略可能な引数は次のとおりです。 **yes** 初期化します。 **no** 初期化しません。[既定値]

3.2.1　NEC Fortran コンパイラ実行時環境変数

主要な NEC Fortran コンパイラ実行時環境変数を指定できます。ただし，次の環境変数は効果を持ちません。

VE_ERRCTL_ALLOCATE ， VE_ERRCTL_DEALLOCATE ， VE_FMTIO_OFFLOAD ， VE_FMTIO_OFFLOAD_THRESHOLD ， VE_FORT*n* ， VE_FORT_DEFAULTFILE ，

VE_FORT_FILEINF ， VE_FORT_FMT_NO_WRAP_MARGIN ， VE_FORT_FMTBUF[n] ，
VE_FORT_FOR_PRINT ， VE_FORT_FOR_READ ， VE_FORT_FOR_TYPE ，
VE_FORT_NML_DELIM_BLANK ， VE_FORT_NML_REPEAT_FORM ， VE_FORT_PAUSE ，
VE_FORT_RECORDBUF[n] ， VE_FORT_SETBUF[n] ， VE_FORT_PAUSE ，
VE_FORT_UFMTADJUST[n]，VE_FORT_UFMTENDIAN，VE_FORT_UFMTENDIAN_NOVEC

詳細は，NEC Fortran ユーザーズガイドを参照してください。

3.2.2　NEC MPI 実行時オプション

主要な NEC MPI 実行時オプションを指定できます。ただし，HPF 実行プログラムは，VE 上でだけ実行できるので，次の NEC MPI 実行時オプションは指定できません。

-vh, -sh, -vpin, -vpinning, -pin_mode, -pin_reserve, -cpu_list, -pin_cpu, -veo, -cuda

詳細は，NEC MPI ユーザーズガイドを参照してください。

3.2.3　NEC MPI 環境変数

全ての NEC MPI 環境変数を指定できます。詳細は，NEC MPI ユーザーズガイドを参照してください。

第4章　　HPF プログラミング

本章では，HPF を使用してプログラムを並列化する方法を説明します。HPF の機能は，データマッピング関連の指示文，処理の分担とデータ転送関連の指示文，不規則問題のための拡張指示文，および 外来手続に分類できます。

本章における構文は，次のように表記されています。

- 構文中の太字の構文要素は，その文字列をそのまま記述することを表しています。
- 構文中の<>で囲まれた構文要素は，いくつかの記述が可能であることを意味しており，直後に，どのような記述が可能であるかが示されています。
- 構文中のイタリックで記述された構文要素は，実体の名前 または 式を表しています。
- 構文中の[]は，その中の構文要素が省略可能であることを表しています。
- 構文中の,...は，直前の構文要素を，カンマで区切って繰返し記述できることを表しています。

4.1　データマッピング

本節では，データマッピング関連の指示文の使用方法を説明します。

4.1.1　DISTRIBUTE 指示文

HPF プログラムを実行する各プロセスは, HPF プログラム中では, 抽象プロセッサと呼ばれます。抽象プロセッサの個数は，実行するプロセス数と同一となります。

DISTRIBUTE 指示文によって，配列の各次元をどのように抽象プロセッサに分割配置するかを指定できます。DISTRIBUTE 指示文によって配列を分割配置することを分散すると呼び，指定された分割配置をその配列の分散と呼びます。

HPF コンパイラは，データのプロセスへの分割配置 および データの演算おけるアクセス方法を考慮して，最適な処理の分担の決定と必要なデータ転送の生成を行います。

DISTRIBUTE 指示文の構文は，次のとおりです。

プロセッサ構成(4.1.3 項参照)を指定する場合

!HPF$ DISTRIBUTE *a* **(** <分散形式>,… **) ONTO** *p*

または

!HPF$ DISTRIBUTE (<分散形式>,… **) ONTO** *p* :: *a*,…

- *a* は、配列またはテンプレート
- *p* は、プロセッサ構成の名前。
- <分散形式>は、*****, **BLOCK**[(<式>)], **GEN_BLOCK**(*map*), または **CYCLIC**[(<式>)]
  - *****は、その次元を分散しないことを指定します。
  - **BLOCK** は、その次元を均等に分散することを指定します。(<式>)により、分散幅を指定できます。分散幅を指定しない場合、分散幅は、次のように計算されます。
    (配列の対応次元の寸法-1)/(プロセッサ構成の対応次元の寸法)
  - **GEN_BLOCK** は、その次元を不均等に分散することを示します。(*map*)により、プロセッサ構成の対応次元の各要素に分散する配列要素の個数を指定します。1 次元配列 *map* には、プロセッサ構成の対応次元の各要素に分散する配列要素の個数をあらかじめ代入しておきます。
  - **CYCLIC** は、その次元をラウンドロビン方式で巡回的に抽象プロセッサに分散することを指定します。(<式>)により、分散幅を指定できます。分散幅を指定しない場合、分散幅は、1 です。

プロセッサ構成を指定しない場合

!HPF$ DISTRIBUTE *a* **(** <分散形式>,… **)**

または

!HPF$ DISTRIBUTE (<分散形式>,… **)** :: *a*,…

図 6　DISTRIBUTE 指示文

図 7 は，最も汎用的な分散形式である BLOCK 分散の例です。

```
      real a(11), b(11)
!HPF$ DISTRIBUTE (BLOCK) :: a, b
        :
      do i=1, 11
        b(i) = a(i) + 1
      enddo
```

図 7　1 次元配列の BLOCK 分散

例えば，4 つの抽象プロセッサ p(1), p(2), p(3), および p(4)で並列実行する場合，配列の各要素は，各抽象プロセッサに，図 8 のように分散されます。

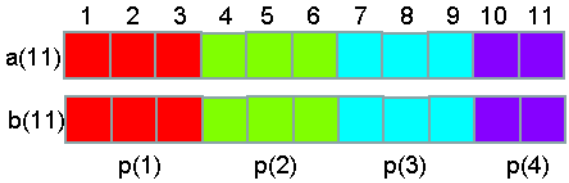

図 8　4 つの抽象プロセッサ上への 1 次元分散

この場合，代入文において，配列 a と b の対応する要素は， 同じ抽象プロセッサ上に分散されているので， HPF コンパイラは，抽象プロセッサに演算を均等に分割して， 図 9 のようにデータ転送なしで並列実行することができます。

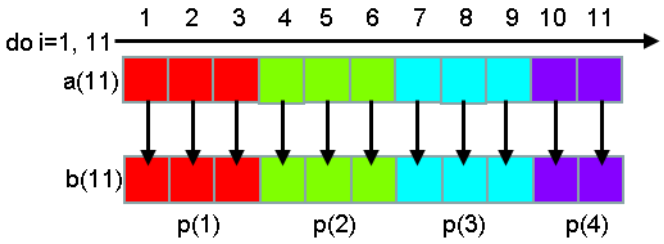

図 9　4 つの抽象プロセッサによるループの並列実行

図 10 は，2 次元配列 a の 2 次元目を BLOCK 分散した例です。

```
      real a(11,11)
!HPF$ DISTRIBUTE (*, BLOCK) :: a
```

図 10　2 次元配列の 1 次元 BLOCK 分散

例えば，4 つの抽象プロセッサ p(1), p(2), p(3), および p(4)で並列実行する場合，配列の各要素は，各抽象プロセッサに，図 11 のように分散されます。

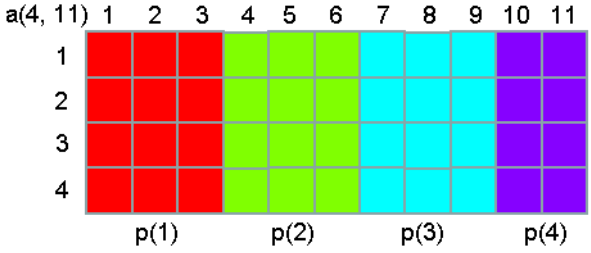

図 11　2 次元配列の 4 つの抽象プロセッサ上への 1 次元 BLOCK 分散

BLOCK 分散の分散幅は，図 12 のように明示的に指定することもできます。

```
      real a(11)
!HPF$ DISTRIBUTE (BLOCK(3)) :: a
```

図 12　分散幅の明示的な指定

ただし，どの抽象プロセッサにも分散されない要素が存在しては ならないため，図 12 のプログラムを 3 つの抽象プロセッサで実行することはできません。 配列要素 a(10)および a(11)が，どの抽象プロセッサにも分散されなくなってしまうからです。

図 15 のように，CYCLIC 分散を指定すると，配列の各要素を， ラウンドロビン方式で巡回的に抽象プロセッサ上に分散できます。

```
      real a(11)
!HPF$ DISTRIBUTE (CYCLIC) :: a
```

図 13　CYCLIC 分散

4 つの抽象プロセッサ p(1), p(2), p(3), および p(4)で並列実行すると， 配列の各要素は，図 14 のように分散されます。

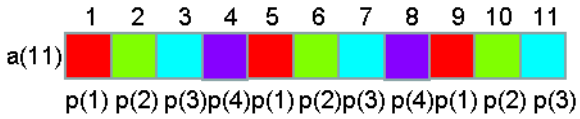

図 14　4 つの抽象プロセッサ上への CYCLIC 分散

図 15 のように，CYCLIC 分散の分散幅を明示的に指定することもできます。

```
      real a(11)
!HPF$ DISTRIBUTE (CYCLIC(2)) :: a
```

図 15　CYCLIC 分散幅の明示的な指定

4 つの抽象プロセッサ p(1), p(2), p(3), および p(4)で並列実行すると， 配列の各要素は，図 16 のように分散されます。

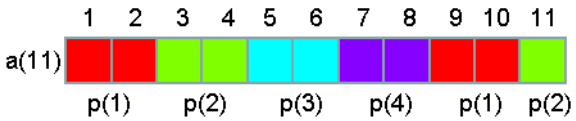

図 16　4 つの抽象プロセッサ上への CYCLIC(2)分散

一般化された BLOCK 分散である GEN_BLOCK 分散によって，図 17 のように，配列の均等でない大きさの連続的な切片を，抽象プロセッサ上に分割配置することができます。

```
      real a(13)
      integer map(4)
      data map/6,3,2,2/
!HPF$ DISTRIBUTE (GEN_BLOCK(map)) :: a
```

図 17　GEN_BLOCK 分散

ここで，GEN_BLOCK 分散に指定する 1 次元整数型配列をマッピング配列と呼びます。マッピング配列の大きさは，分散される抽象プロセッサの個数と同一でなければなりません。

4 つの抽象プロセッサ p(1), p(2), p(3), および p(4)で並列実行すると，配列の各要素は，図 18 のように分散されます。

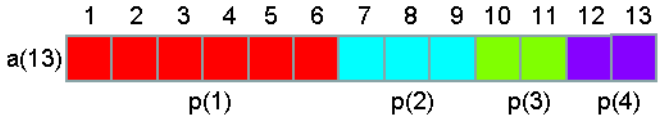

図 18　4 つの抽象プロセッサ上への GEN_BLOCK 分散

CYCLIC 分散や GEN_BLOCK 分散は，不均等な計算の負荷バランスを均等化するために使用できます。例えば，図 19 ように三角行列の和を計算する場合，

```
      real a(8,8), b(8,8), c(8,8)
        :
      do j=1, 13
        do i=1,j
          a(i,j) = b(i,j) + c(i,j)
        enddo
      endo
```

図 19　三角行列の和

各配列の 2 次元目を図 20 のように BLOCK 分散して，4 つの抽象プロセッサ p(1), p(2), p(3), および p(4)上で並列実行すると，図 21 のように，p(1), p(2), p(3), および p(4)が，それぞれ 3 要素，7 要素，11 要素，および 15 要素の代入を実行することになり，負荷バランスが悪くなります。

```
      real a(8,8), b(8,8), c(8,8)
!HPF$ DISTRIBUTE (*, BLOCK) :: a, b, c
```

図 20　2 次元目を BLOCK 分散

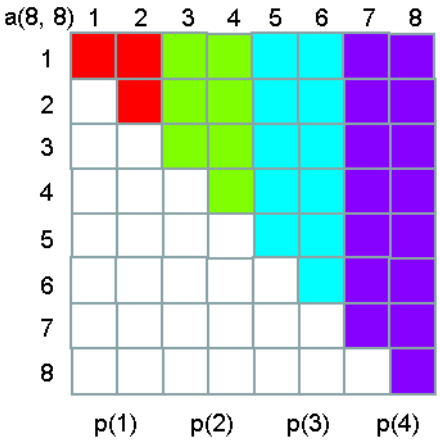

図 21　BLOCK 分散時の三角行列の和

各配列の 2 次元目を図 22 のように GEN_BLOCK 分散し，4 つの抽象プロセッサ p(1), p(2), p(3), および p(4)で並列実行すると，図 23 のように，p(1), p(2), p(3), および p(4)が，それぞれ 10 要素，11 要素，7 要素，および 8 要素の代入を実行することになり，負荷バランスが改善されます。

```
      real a(8,8), b(8,8), c(8,8)
      integer map(4)
      data map/4,2,1,1/
!HPF$ DISTRIBUTE (*, GEN_BLOCK(map)) :: a, b, c
```

図 22　2 次元目を GEN_BLOCK 分散

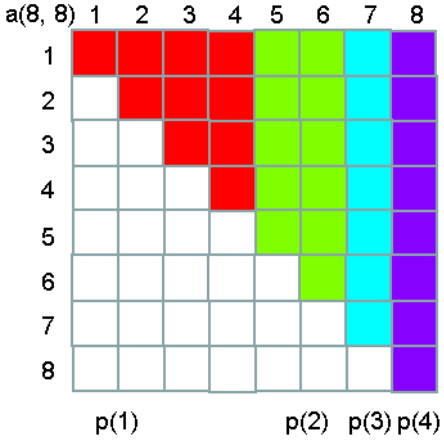

図 23　GEN_BLOCK 分散による三角行列の和

4.1.2　分散形式の選択

配列のアクセスパタンによって適切な分散形式は異なります。 多くの場合，各配列要素に対する処理量が概ね均等であるならば BLOCK 分散，不均等であるならば GEN_BLOCK 分散を選択するのが良いでしょう。 これら 2 つの分散形式は，並列化の粒度を大きくし易く，添字の連続した配列要素が

各抽象プロセッサ上に分散されるので，HPF コンパイラが効率の良い並列化を行い易いからです。さらに，後述するシフト転送等の定型化された効率のよいデータ転送パタンが適用できるので，高い実行性能を得やすい からです。

4.1.3　PROCESSORS 指示文

PROCESSORS 指示文により，抽象プロセッサの構成(プロセッサ構成)を宣言できます。配列形状のプロセッサ構成をプロセッサ配列といいます。プロセッサ配列の大きさは，実行するプロセス数と同一でなければなりません。

PROCESSORS 指示文の構文は，次のとおりです。

!HPF$ PROCESSORS *p* **(** **<>**,... **)**
または
!HPF$ PROCESSORS (**<>**,... **)** :: *p*,...

- *p* は、プロセッサ構成の名前
- **<>**は、プロセッサ構成の各次元の上下限。例えば、次の場合、
  !HPF$ PROCESSORS p(n1,n2)
  プロセッサ配列 p の大きさ n1*n2 が抽象プロセッサの個数であり、プロセッサ配列の次元数である 2 は、配列を分散する次元の個数と一致します。

図 24　PROCESSORS 指示文の構文

プロセッサ配列の形状は，プログラミングの都合にあわせて自由に宣言できます。プロセッサ配列の次元数は，データ配列を何次元分プロセスに分割配置するかに対応しており，さらに HPF プログラム中の多重ループが，いくつ並列化されるかに対応しています。

```
      real a(100,100)
!HPF$ PROCESSORS p(4)
!HPF$ DISTRIBUTE a(*, BLOCK) ONTO p
```

図 25　1 次元プロセッサ配列を使用する 1 次元分散(1 次元並列化)の例

```
      real a(100,100)
!HPF$ PROCESSORS p(2,2)
!HPF$ DISTRIBUTE a(BLOCK,BLOCK) ONTO p
```

図 26　2 次元プロセッサ配列を使用する 2 次元分散(2 次元並列化)の例

ただし，並列化の総数は，プロセス数と常に同一なので，1 次元のプロセッサ配列を使用した 1 つのループによる並列化が，多くの場合，並列化のオーバヘッドを小さくしやすくなります。

実行するプロセス数を実行時に決めたい場合， 組込み関数 NUMBER_OF_PROCESSORS()を使用します。 この組込み関数は，プロセス数を返します。

```
      real a(100,100)
!HPF$ PROCESSORS p(NUMBER_OF_PROCESSORS())
!HPF$ DISTRIBUTE a(*, BLOCK) ONTO p
```

図 27　組込み関数 NUMBER_OF_PROCESSORS()の使用

さらに，プロセッサ配列の宣言自体を省略して，図 28 のように記述しても， HPF コンパイラは，プロセス数と大きさが等しいプロセッサ配列への分割配置が指定されたものとして扱いますので，図 27 と全く同じ意味となります。従って，1 次元分散を行う場合，特にプロセッサ配列を宣言する必要はありません。

```
      real a(100,100)
!HPF$ DISTRIBUTE a(*, BLOCK)
```

図 28　PROCESSORS 指示文の省略

以下のように， 異なる形状のプロセッサ配列を混在させることも， それらの大きさが等しければ許されますが， 異なる形状のプロセッサ配列上に分割配置された配列データが混在すると，それらが同じ抽象プロセッサ上にあるかどうか判定，すなわち データ転送が必要かどうかの判定が困難になり，無駄なデータ転送が生成される場合があるので， 特別な理由がない限り，避けた方が良いでしょう。

```
      real a(100,100), b(100,100)
!HPF$ PROCESSORS p1(4), p2(2,2)
!HPF$ DISTRIBUTE a(*, BLOCK) ONTO p1
!HPF$ DISTRIBUTE b(BLOCK, BLOCK) ONTO p2
```

図 29　2 種類のプロセッサ配列の例

4.1.4　ALIGN 指示文

BLOCK 分散は，配列を均等に分散するので，図 30 のように，必要なデータの大きさ（下の例では n）が実行時に定まるような場合， 配列を大きめ（下の例では 11）に宣言しておいて，BLOCK 分散で分割配置すると，計算対象のデータが配置されない抽象プロセッサが発生して，不均等な作業負荷の原因となることに注意が必要です。

```
      real a(11), b(11)
!HPF$ DISTRIBUTE (BLOCK) :: a, b
      read(*,*)n
      do i=1, n
        b(i) = a(i) + 1
      enddo
```

図 30　必要なデータの大きさ n が実行時に決まる例

例えば n の値が 6 の場合，4 つの抽象プロセッサ p(1), p(2), p(3), および p(4)で並列実行すると，計算対象の要素は，図 8 のように p(1)および p(2)だけに分散され，p(3)および p(4)が遊んでしまうことになります。

必要な大きさが実行時に定まる場合には，以下のように，割付け配列や自動割付け配列により，動的に必要な大きさを 割り付けるのが良いでしょう。

```
      real, allocatable :: a(:)          ! 割付け配列
!HPF$ DISTRIBUTE (BLOCK) :: a
      read(*,*)n
      allocate(a(n))
        :
      call sub(a,n)
        :
      end

      subroutine sub(a,n)
      real a(n)                          ! 自動割付け配列
!HPF$ DISTRIBUTE (BLOCK) :: a
```

図 31　割付け配列・自動割付け配列の分散

割付け配列を分割配置する際に注意が必要なのは，配列の上下限がコンパイル時には決まらないので，各配列要素がどの抽象プロセッサに配置されるかは，一般に実行時までわからないということです。例えば図 32 の DO ループの場合，BLOCK 分散によりプロセッサ配列 p 上に分割配置されている 1 次元配列 a および b の上下限がそれぞれ a(1:10), b(1:11)だとすると，図 33 のように，a(6)は p(2) 上に，b(6) は p(1) 上にそれぞれ配置されるので，代入文 a(6)=b(6) の実行にデータ転送が必要となります。コンパイル時に宣言範囲が不明の場合，DISTRIBUTE 指示文だけでデータマッピングを指定すると，たとえ実際には宣言範囲が同じであっても，HPF コンパイラにはそのことがわからないので，データ転送が必要な場合も動作するよう効率の悪い実行コードを生成する可能性があります。

```
      real, allocatable :: a(:), b(:)          ! 割付け配列
!HPF$ PROCESSORS p(2)
!HPF$ DISTRIBUTE (BLOCK) ONTO p :: a, b
      read(*,*)n1, n2
      allocate(a(n1), b(n2))
      do i=1,10
        a(i) = b(i)
      enddo
```

図 32　割付け配列の BLOCK 分散

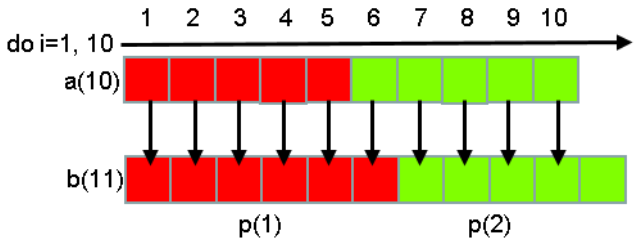

図 33　BLOCK 分散によりデータ転送が必要となる場合

このような場合に有効なのが ALIGN 指示文です。ALIGN 指示文を使うと，複数の配列の相対的位置関係（整列）を指定できます。ALIGN 指示文の構文は，図 34 のとおりです。

!HPF$ ALIGN *a* **(** <*i*>,… **) WITH** *t***(** <*f(i)*>,… **)**
　または
!HPF$ ALIGN (<*i*>,… **) WITH** *t***(** <*f(i)*>,… **) ::** *a*,…

- *a* は、配列
- *t* は、配列またはテンプレート
- <*i*> は，整数型スカラ変数または * 。* を指定した次元は整列されない。
- <*f(i)*> は，<*i*> の 1 次式 *s***<*i*>+*o*，または * 。ここで，*s* と *o* は整数型の式。
  - <*f(i)*> が <*i*> の 1 次式 *s***<*i*>+*o* の場合，配列 *a* の要素 <*i*>は，整列先 *t* の要素 *s***<*i*>+*o* に整列する。
  - <*f(i)*> が * の場合，配列 *a* 全体は，整列先 *t* の * が指定された次元に対応するプロセッサ配列の次元にそって複製される。（整列先 *t* の * が指定された次元の，任意の要素が配置されるすべての抽象プロセッサ上に配置される。）

図 34　ALIGN 指示文

例えば図 35 の ALIGN 指示文は，配列 b を基準配列として，配列要素 a(i) を b(i) と同一の抽象プロセッサに配置することを意味しています。この基準配列を整列先とよびます。整列先 b を DISTRIBUTE 指示文で分散すれば，配列 a のデータマッピングは，配列 b との相対的な位置関係

により自動的に決まります。ALIGN 指示文を利用すると，実行時に宣言範囲が決まる場合でも，配列どうしの添字の対応により，配列要素 b(i) と a(i) とが常に同一のプロセッサ上に配置されていることがコンパイル時にわかるので，ループを並列実行する際に，図 36 のように，データ転送が不要であることが判定可能になり，HPF コンパイラは効率の良い実行コードを生成できます。

```
      real, allocatable :: a(:), b(:)          ! 割付け配列
!HPF$ PROCESSORS p(2)
!HPF$ ALIGN a(i) WITH b(i)
!HPF$ DISTRIBUTE (BLOCK) ONTO p :: b
      read(*,*)n1, n2
      allocate(a(n1), b(n2))
      do i=1,10
        a(i) = b(i)
      enddo
```

図 35　ALIGN 指示文を使用したデータマッピング

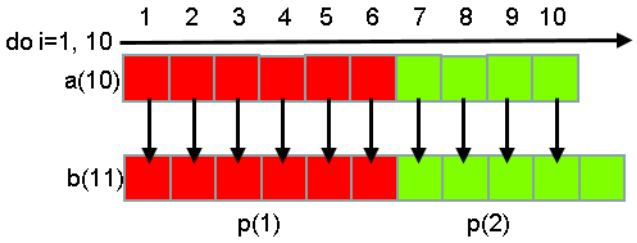

図 36　ALIGN 指示文の効果

同様に，形状引継ぎ配列やみかけ上宣言範囲の異なる自動割付け配列（図 37 参照）に対しても，ALIGN 指示文が有効です。

```
      :
      call sub(a,100, 100)
      end

      subroutine sub(a,n,m)
      real :: a(:)          ! 形状引継ぎ配列
      real :: b(n), c(m)    ! 自動割付け配列
```

図 37　形状引継ぎ配列 および 自動割付け配列

図 38 のように，宣言範囲が異なる配列が同一のループ中でアクセスされている場合も，ALIGN 指示文が有効です。宣言範囲が異なる配列を BLOCK 分散で均等に分割配置すると，図 39 のように，各抽象プロセッサに配置される範囲も少しずれてしまうため，ループの実行時にデータ転送が必要となります。そこで図 40 のように ALIGN 指示文を指定すれば，配列要素 a(i) と b(i) が，ループの同じ繰返しでアクセスされる c(i) と同一の抽象プロセッサに配置され，データ転送の発生を避けられ

ます。

```
real a(0:9), b(10), c(0:10)

do i=1,9
   c(i) = a(i) + b(i)
enddo
```

図 38　宣言範囲の異なる配列

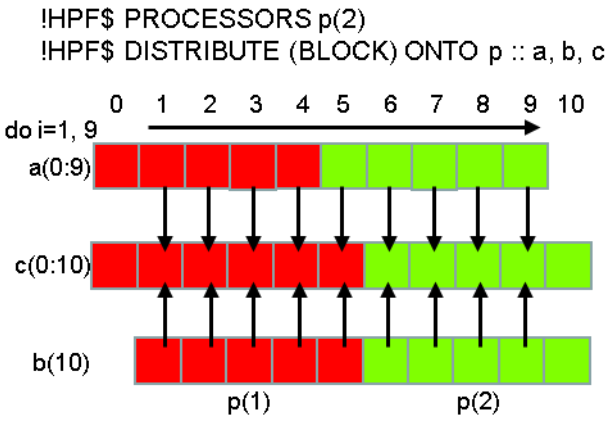

図 39　宣言範囲の異なる配列の BLOCK 分散

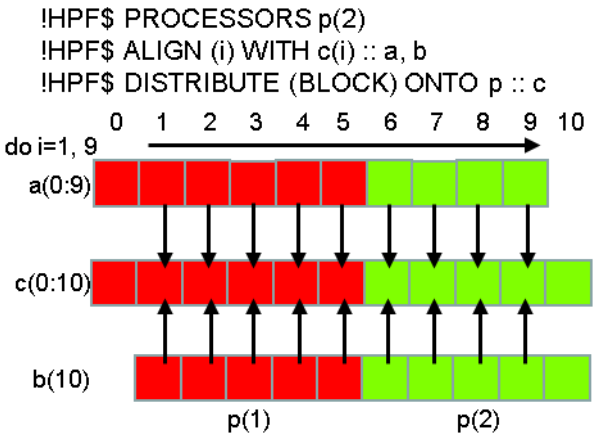

図 40　宣言範囲の異なる配列の整列

ここで ALIGN 指示文の整列先 c の宣言範囲（この場合，(0:10)）は，整列を指定する配列 a, b の（対応次元の）宣言範囲（この場合，それぞれ (0:9), (1:10)）を包含している必要があるということに注意してください。整列先に対して“はみ出す”要素があると，その要素は抽象プロセッサとの対応がとれないので，コンパイル時または実行時エラーとなるからです。

4.1.5　TEMPLATE 指示文

図 41 の例では，ループの同じ繰返しでアクセスされる配列要素 a(i) と b(i) とを整列したいとこ

ろですが，配列 a と b は宣言範囲が異なるので，どちらを整列先として ALIGN 指示文を指定しても，例えば図 42 のように“はみ出る”要素が発生してしまいます。図 40 の配列 c のような，両方の配列の宣言範囲を包含する配列を別に宣言すればよいのですが，データマッピングの指定のためだけにメモリを浪費してしまうことになります。

```
real a(0:9), b(10)

do i=1,9
   b(i) = a(i) + 1.0
enddo
```

図 41　宣言範囲が異なる場合

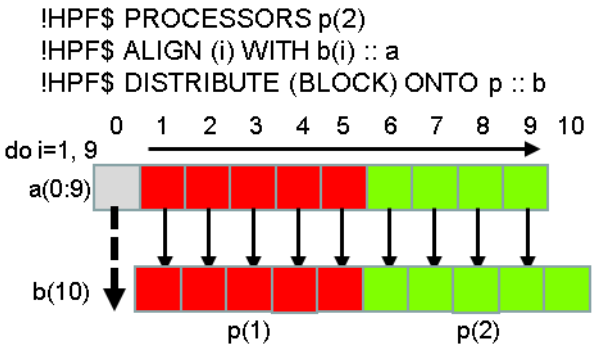

図 42　はみ出る配列要素が発生する ALIGN 指示文（文法違反）

このような場合に有効なのが TEMPLATE 指示文です。TEMPLATE 指示文を使うと，メモリを消費しない仮想配列（テンプレート）を宣言できます。

TEMPLATE 指示文の構文は，図 43 のとおりです。

!HPF$ TEMPLATE *t*(<>,…)
　または
!HPF$ TEMPLATE (<>,…) :: *t*,…

- *t* は、テンプレート
- <> は，テンプレートの各次元の上下限

図 43　TEMPLATE 指示文

図 44 のように,分割する次元に関して両方の配列の宣言範囲を包含するようなテンプレート t を宣言し，それを整列先として各配列に ALIGN 指示文を指定したうえで，テンプレート t を分散すれば，図 41 のループ実行時にデータ転送が発生しないよう配列 a と b の対応する要素を同一の抽象プロセッサに配置できます。

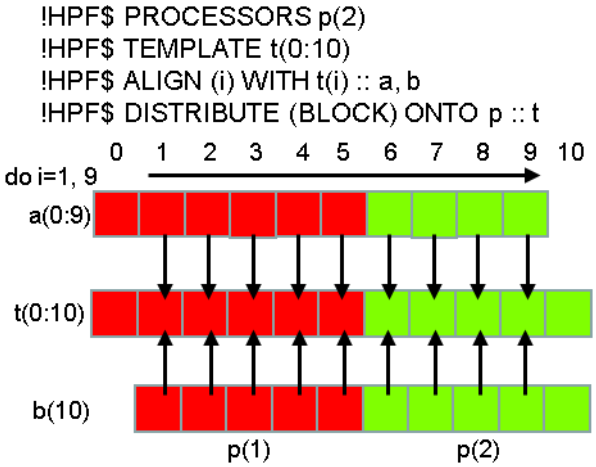

図 44　TEMPLATE 指示文を使用したデータマッピング

図 45 のループは，配列 a と b の宣言範囲は同一ですが，添字が一つずれているので，BLOCK 分散だけではやはりデータ転送が発生してしまいます。そこで，図 46 のように ALIGN 指示文で a(i+1) と b(i) とを整列させれば，データ転送なしで実行が可能になります。この例でも，a(i+1) と b(i) を直接整列させると“はみ出す”要素が発生してしまうので，両配列の宣言範囲を包含するテンプレート t を宣言し，ALIGN 指示文の整列先としています。

```
real a(10), b(10)

do i=1,9
  b(i) = a(i+1) + 1.0
enddo
```

図 45　添字が異なる場合

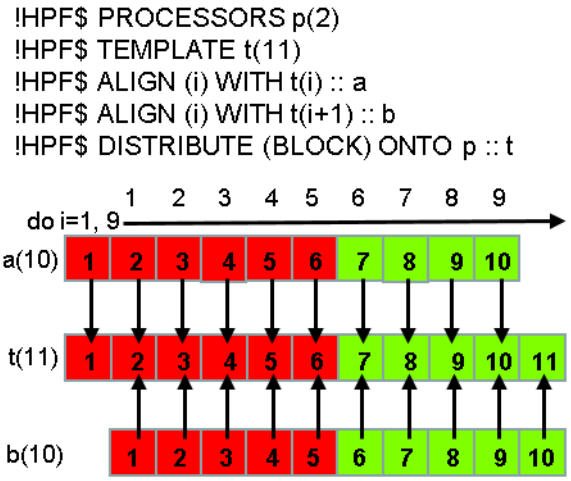

図 46　配列添字が異なる場合の TEMPLATE の使用

4.1.6　HPF のデータマッピングまとめ

HPF では，配列のデータマッピングは，DISTRIBUTE 指示文と ALIGN 指示文の組合せで指定することができます。一般には，図 47 のように，整列先とする（基準となる）配列またはテンプレートに対する相対的な位置関係を ALIGN 指示文で指定したうえで，その整列先に対して DISTRIBUTE 指示文を指定すれば，全ての配列のデータマッピングを決定できます。

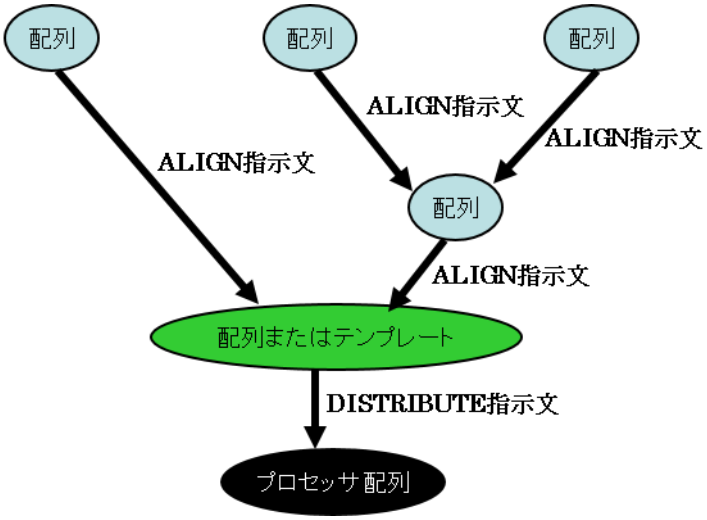

図 47　HPF のデータマッピング

スカラ変数 および DISTRIBUTE 指示文も ALIGN 指示文も指定されていない配列は，全ての抽象プロセッサ上に複製されます。読出すだけの配列は，複製しておくと，データ転送なしで全ての抽象プロセッサから読み出すことができるので，効率よく処理できます。

4.1.7　分割配置できない変数

HPF では，形状の異なる実引数と仮引数を Fortran の順序結合に基づいて結合する場合 および EQUIVALENCE 文や共通ブロックを通じて，形状の異なる変数同士を Fortran の記憶列結合に基づいて結合する場合，宣言部でそれらに SEQUENCE 指示文を指定する必要があります。SEQUENCE 指示文を指定した配列を分割配置することはできません。SEQUENCE 指示文の構文は，図 48 のとおりです。NOSEQUENE 指示文は，HPF コンパイラオプション-Msequence を指定した場合に，分割配置したい変数に対して指定します。

!HPF$ [NO] SEQUENCE　[[::] *s*,…]

- *s* は、配列名 または /共通ブロック名/。SEQUENCE 指示文において，*s*,… を省略した場合，全ての共通ブロック および 明示的に分割配置されていない全ての変数を指定したことになる。NOSEQUENCE 指示文において，*s*,…を省略した場合，全ての共通ブロック および 全ての変数を指定したことになる。

図 48　SEQUENCE 指示文

4.2　処理の分担とデータ転送

本節では，処理の分担とデータ転送を改善するための指示文の使用方法を説明します。

4.2.1　INDEPENDENT 指示文

INDEPENDENT 指示文は，図 49 のように，ループの繰返しにまたがる依存がないループに対して，並列実行可能であることを， プログラマが HPF コンパイラに明示するための指示文です。INDEPENDENT 指示文が指定されたループを，INDEPENDENT ループと呼びます。

```
!HPF$ INDEPENDENT
      do i=1,n
        a(i) = i
      enddo
```

図 49　並列実行可能なループへの INDEPENDENT 指示文の指定

ループの繰返しにまたがる依存には，図 50 のようなものがあり，このようなループは並列化できません。簡単にいうと，ループ中からループ外へ飛び出すループや，ループのある繰返しで確定されたデータを他の繰返し引用または確定するループは，並列化できません。

```
! 依存：配列要素 a(i)は，確定後，引用される
        do i=1,n
          a(i) = a(i) + a(i-1)
        enddo

! 逆依存：配列要素 a(i)は，引用後，確定される
        do i=1,n
          a(i) = a(i) + a(i+1)
        enddo

! 出力依存：スカラ変数 s は，確定後，確定される
        do i=1,n
          if(a(i) > 0.0) s = a(i)
        enddo

! 制御依存：ループの実行が，途中で終了する可能性がある
        do i=1,n
          if(a(i) > 0.0)goto 99
        enddo
99      continue
```

図 50　ループの繰返しにまたがる依存

INDEPENDENT 指示文の構文は，図 51 のとおりです。

完全に並列化可能なループの場合

!HPF$ INDEPENDENT [, **NEW(** *v*,… **)**]

- *v* は，変数名（NEW 変数）

集計計算を行う並列化可能なループの場合

!HPF$ INDEPENDENT [, **NEW(** *v*,… **)**], <REDUCTION 節>,…

- *v* は，変数名（NEW 変数）
- <REDUCTION 節>は，

  REDUCTION([<集計種別 1>：] *r*,… **)**

  または

  REDUCTION([<集計種別 2>：] *r* **/***p*,…**/**,… **)**

  - <集計種別 1>は，**+, *, .AND.**, **.OR.**, **.EQV.**, **.NEQV.**, **MAX**, **MIN**, **IAND**, **IOR**, または **IEOR**
  - *r* は，集計変数の名前
  - <集計種別 2>は，**FIRSTMAX**, **FIRSTMIN**, **LASTMAX**, または **LASTMIN**
  - *p* は，位置変数の名前
  - <集計種別 1>：を省略した場合，許される集計演算の形式は，

    r **=** *r* <op> <expr> または *r* **=** <expr> <op> *r*

    もしくは

    r **=** <f**(***r***,** <expr>**)**> または *r* **=** <f**(**<expr>**,** *r***)**>

    - *r* は，集計変数の名前
    - <op>は，集計演算子*****, **/**, **+**, **-**, **.AND.**, **.OR.**, **.EQV.**, または **.NEQV.**
    - <expr>は，集計変数を含まず，<op>よりも先に計算される式
    - <f()>は，集計関数 **MAX**, **MIN**, **IAND**, **IOR**, または **IEOR**

図 51　INDEPENDENT 指示文

HPF コンパイラは， INDEPENDENT 指示文の指定がなくても，コンパイル時に並列化可能であることがわかれば，自動的に INDEPENDENT 指示文が指定された DO ループと同様の並列化を行いますが,ループ中の配列の参照方法によっては,自動的には最適に並列化できない場合があります。並列化情報リストや診断メッセージで，本来は並列化できるにも関わらず並列化できないと判定されていたり，不要なデータ転送が生成されていたりすることがわかった場合，INDEPENDENT 指示文を指定することで， プログラムが実行性能を劇的に向上する場合もあります。

図 52 の場合，do j のループの左辺と右辺の要素は，重なることがなく，左辺の g(j,new)のデータ

マッピングに合わせて， do j のループの繰返しを抽象プロセッサへ割り当てると，隣接抽象プロセッサ間のデータ転送(シフト転送)だけで，効率の良い並列化が可能です。
しかし， HPF コンパイラは，変数 iold と inew の値が常に異なることを解析できないので， このようなループを 自動的には並列化できません

```
      subroutine sub(n, ncycles, g)
      real g(n+2,2)
!HPF$ DISTRIBUTE g(BLOCK,*)

       iold=1
       inew=2
       do it=1, ncycles
          do j = 2, n+1
              g(j,new) = g(j-1,iold) + g(j+1,iold) + g(j,iold)
          enddo
       enddo
       iold = 3 - iold
       inew = 3 - inew
       enddo
```

図 52　自動では並列化できないループの例

実際，HPF コンパイラオプション-Minfo を指定してコンパイルすると，次のような診断メッセージが出力されます。

```
7, Invariant assignments hoisted out of loop
8, Distributing inner loop; 2 new loops
   expensive communication: scalar communication (get_scalar)
   expensive communication: scalar communication (get_scalar)
```

このうち，expensive communication: scalar communication (get_scalar) という 2 つの診断メッセージは，オーバヘッドの大きなパタンのデータ転送が生成されたことを意味しています。本来必要なデータ転送は，オーバヘッドの少ないシフト転送だけのはずなので自動的にはうまく並列化ができなかったことが 分かります。このような場合，図 53 のように，do j のループに INDEPENDENT 指示文を指定します。

```
      subroutine sub(n, ncycles, g)
      real g(n+2,2)
!HPF$ DISTRIBUTE g(BLOCK,*)

       iold=1
       inew=2
       do it=1, ncycles
!HPF$    INDEPENDENT
         do j = 2, n+1
            g(j,new) = g(j-1,iold) + g(j+1,iold) + g(j,iold)
         enddo
       enddo
       iold = 3 - iold
       inew = 3 - inew
       enddo
```

図 53　INDEPENDENT 指示文の指定

すると，HPF コンパイラオプション-Minfo を指定してコンパイルした場合，次の診断メッセージが出力され，オーバヘッドの高いデータ転送なしで，うまく並列化されたことが分かります。

```
7, Invariant communication calls hoisted out of loop
9, Independent loop parallelized
```

4.2.2　NEW 節

図 54 のループは，スカラ変数 s が，ループの複数の繰返しで確定および引用されているので，本来は並列化できません。

```
      do i=1,n
        s = sqrt(a(i)**2 + b(i)**2)
        c(i) = s
      enddo
```

図 54　作業変数を含むループ

しかし，図 55 のように，NEW 節中に変数 s を指定した INDEPENDENT 指示文を指定すると，変数 s に対して繰返しごとに別々のメモリ領域を用意すれば，並列化可能であることを明示できます。NEW 節中に指定した変数は，NEW 変数と呼ばれます。

```
!HPF$ INDEPENDENT, NEW(s)
        do i=1,n
          s = sqrt(a(i)**2 + b(i)**2)
          c(i) = s
        enddo
```

図 55　NEW 節付きの INDEPENDENT 指示文

NEW 変数の値は，INDEPENDENT ループの実行後不定となることに注意してください。従って，ループ実行後，NEW 変数（この場合は，s）の値を確定することなくそのまま引用した場合， プログラムの動作は保証されません。

HPF コンパイラは，多くの場合，コンパイル時にスカラの作業変数は自動的に検出して NEW 変数として扱いますが，配列が作業変数として利用されている場合は自動検出できないので，NEW 節付きの INDEPENDENT 指示文を指定する必要があります。
例えば，図 56 の場合，配列 u および flux が作業変数として利用されており，繰返しごとに，確定されては引用されていますが，これらの配列に対して，各繰返しで別々のメモリ領域を使用すれば，配列 f の分散次元に対応する do k のループで，データ転送なしで並列化できます。

```
        subroutine rhs(f, u, n1, n2, n3)
        common /com/c1, c2, q
        dimension flux(2,n1), u(2,n1)
        dimension f(2, n1, n2, n3)
!HPF$ DISTRIBUTE F(*,*,*,BLOCK)

        do k=2, n3-1
          do j=2,n2-1
            do i=1,n1
              do m=1,2
                u(m, i) = c1 –c2
              enddo
              flux(1, i) = q * u(1,i)
              flux(2, i) = q * u(2,i)
            enddo
            do i=2, n1-1
              f(1,i,j,k) = f(1,i,j,k) * (flux(1,i+1) – flux(1, i-1))
              f(2,i,j,k) = f(2,i,j,k) * (flux(2,i+1) – flux(2, i-1))
            enddo
          enddo
        enddo
```

図 56　作業配列を含むループ

このコードを，HPF コンパイラオプション-Minfo を指定して翻訳すると，次の診断メッセージが出力されます。

```
  9, Distributing loop; 2 new loops
       1 FORALL   generated
       2 FORALLs generated
       no parallelism: replicated array, u
       no parallelism: replicated array, flux
       no parallelism: replicated array, flux
 10, Independent loop
 16, Independent loop
       2 FORALLs generated
```

診断メッセージ 10, Independent loop および 16, Independent loop から， 10 行目の do m のループおよび 16 行目の do i のループは，並列化可能だと判定されていることが分かりますが， 肝心の 7 行目の do k のループは，並列化可能と判定されていません。そこで，図 57 のように，.配列 u および flux，さらに内側ループの DO 変数 j,i,および m を NEW 節中に指定した INDEPENDENT 指示文を do k のループに指定します。

```
      subroutine rhs(f, u, n1, n2, n3)
      common /com/c1, c2, q
      dimension flux(2,n1), u(2,n1)
      dimension f(2, n1, n2, n3)
!HPF$ DISTRIBUTE F(*,*,*,BLOCK)

!HPF$ INDEPENDENT, NEW(u, flux, j, i, m)
      do k=2, n3-1
        do j=2,n2-1
          do i=1,n1
            do m=1,2
              u(m, i) = c1 –c2
            enddo
            flux(1, i) = q * u(1,i)
            flux(2, i) = q * u(2,i)
          enddo
          do i=2, n1-1
            f(1,i,j,k) = f(1,i,j,k) * (flux(1,i+1) – flux(1, i-1))
            f(2,i,j,k) = f(2,i,j,k) * (flux(2,i+1) – flux(2, i-1))
          enddo
        enddo
      enddo
```

図 57　作業配列を NEW 節に指定した INDEPENDENT 指示文

すると，HPF コンパイラオプション-Minfo を指定してコンパイルした場合の診断メッセージは，次のようになり，8 行目の do k のループが INDEPENDENT ループとして 並列化されたことが分かります。

```
  8, Independent loop parallelized
 11, Independent loop
 17, Independent loop
```

複数の INDEPENDENT ループが入れ子になっている場合，NEW 変数の NEW 節への指定場所は，その NEW 変数に定義を行う INDEPENDENT ループの内，一番内側です。 例えば，図 58 の場合，do i および do j の INDEPENDENT ループ中で代入される作業変数 s は，最内側の do j のループに対する INDEPENDENT 指示文の NEW 節中に記述します。

```
!HPF$ INDEPENDENT, NEW(j)
      do i=1, n
!HPF$   INDEPENDENT, NEW(s)
        do j=1,n
          s = sqrt(a(i,j)**2 + b(i,j)**2)
          c(i,j) = s
        enddo
      enddo
```

図 58　複数の INDEPENDENT ループで確定される NEW 変数

プログラムのループ中で確定される 任意のスカラ変数が，次項で説明する集計変数を除いて全て NEW 変数と見なしても良い場合， HPF コンパイラオプション-Mscalarnew を利用することができます

また，プログラムのループ中で確定される データマッピング指定のない任意の配列が， 集計変数を除いて全て NEW 変数と見なしても良い場合， HPF コンパイラオプション-Mnomapnew を利用することができます。

上記の例では，ループ中で定義される 全てのスカラ変数 k, i, j, および m ならびに データマッピング指定のない全ての配列 u および flux は，全て NEW 変数として扱えるため， NEW 節を省略した INDEPENDENT 指示文を指定した上で，HPF コンパイラオプション-Mscalarnew および-Mnomapnew を指定して翻訳すれば， HPF コンパイラは，これらの変数を自動的に NEW 変数として扱います。

4.2.3　REDUCTION 節

図 59 のように，1 つの変数（この場合は r)に，ループの各繰返しで 次々と同じ種類の演算が行われ，その値が蓄積されていく場合，ループの繰返し間での依存があることになるため，そのままでは，INDEPENDENT 指示文を指定することはできません。しかし，加法には結合律と交換律が成立するので， 各抽象プロセッサが，まず配列 a の所有範囲だけを一時領域に足し込んでおき（これを，ローカル集計演算 と言います），その後それぞれの計算結果をデータ転送して足し合わせる（これを，グローバル集計演算 と言います）ことにより，ほぼ並列にループを実行でできます。このような計算を

集計計算と呼び，この例の s のような変数を，集計変数と呼びます。

```
      real a(10)
!HPF$ PROCESSORS p(2)
!HPF$ DISTRIBUTE a(BLOCK) ONTO p

      r=0
      do i=1,10
        r = r + A(I)
      enddo
```

図 59　集計計算を行うループ

図 60 のように，集計変数を REDUCTION 節中に記述すると，集計計算を行うループに INDEPENDENT 指示文を指定できます。NEW 節中に集計変数を指定するのは，集計変数の値がループの繰返しまたいで蓄積されなければならず，さらにループ実行後に値が保存されていなければならないので間違いです。

```
      real a(10)
!HPF$ PROCESSORS p(2)
!HPF$ DISTRIBUTE a(BLOCK) ONTO p

      r=0
!HPF$ INDEPENDENT, REDUCTION(r)
      do i=1,10
        r = r + A(I)
      enddo
```

図 60　REDUCTION 節付きの INDEPENDENT 指示文

複数の INDEPENDENT ループが入れ子になっている場合，集計変数の REDUCTION 節への指定場所は，その変数に対する集計計算を行う INDEPENDENT ループの内，一番外側です。 例えば，図 61 の場合, do i および do j の INDEPENDENT ループ中で, 集計計算が行われる 集計変数 s は，最外側の do i のループに対する INDEPENDENT 指示文の REDUCTION 節中に記述します。

```
!HPF$ INDEPENDENT, NEW(j),REDUCTION(s)
      do i=1,n
!HPF$   INDEPENDENT
        do j=1,n
          s = s + a(i,j)
        enddo
      enddo
```

図 61　複数のループで実行する集計計算

4.2.4　手続呼出しを含むループの並列化

図 62 のように，手続の引用を含むループは，並列化可能かどうかがコンパイル時には分からないので，自動的に並列化することはできません。

```
      integer a(100,100)
!HPF$ DISTRIBUTE A(*,BLOCK)
        :
      do i=1,100
        call sub(a(:,i))
      enddo
        :
      end
      subroutine sub(a)
      integer a(100)
      do j=1,100
        a(j) = j
      enddo
      end
```

図 62　手続呼出しを含むループ

図 62 では，ループの各繰返しで，サブルーチン sub を呼び出すたびに，2 次元配列 a の各列を引数として渡して，値を確定しています。引数以外の変数の値が変わったり，入出力を行ったりしないので，実際にはループは並列化可能です。このような場合，HPF から Fortran の手続を呼び出せる外来手続機能を利用すると，ループの並列化が可能です。外来手続機能は，PROGRAM 文，FUNCTION 文，SUBROUTINE 文，または MODULE 文の先頭に，図 63 のような外来接頭辞を指定することにより，HPF 以外の言語 または グローバルモデル以外のモデルで記述された手続を，HPF 手続中から引用できるようにする機能です。グローバルモデル以外のモデルとしては，ローカルモデル および シリアルモデルが利用できます。ローカルモデルの手続は，MPI プログラムのように，各抽象プロセッサが独立に手続を実行します。シリアルモデルの手続は，1 つの抽象プロセッサだけが実行します。

EXTRINSIC（<言語>,<モデル>）

　または

EXTRINSIC（<外来種別キーワード>）

- <言語>は，**"HPF"** または **"Fortran"**
- <モデル>は，**"GLOBAL"**，**"LOCAL"**，または **"SERIAL"**。**"GLOBAL"**，**"LOCAL"**，および **"SERIAL"**は，それぞれグローバルモデル，ローカルモデル，および シリアルモデルを示します。
- <外来種別キーワード>は，**HPF**，**HPF_LOCAL**，**HPF_SERIAL**，**Fortran_LOCAL**，または **Fortran_SERIAL**。それぞれ，グローバルモデルの HPF，ローカルモデルの HPF，シリアルモデルの HPF，ローカルモデルの Fortran，シリアルモデルの Fortran を意味します。

図 63　外来接頭辞

図 62 のループは，次のような指定を追加すれば，HPF コンパイラにより並列化されます。

- 明示的引用仕様(interface block)を記述して，引用される手続が Fortran 言語のローカルモデルの手続であることを宣言する。
- 引用される手続の SUBROUTINE 文の先頭に外来接頭辞を指定して，手続が Fortran 言語のローカルモデルの手続であることを宣言する。
- ループに対して INDEPENDENT 指示文を指定する。

この場合，HPF コンパイラは，do i のループの各繰返しを，配列 a の所有範囲に合わせて，抽象プロセッサに割り当てて並列化します。

HPF コンパイラは，ローカルモデルの手続中でデータ転送が必要ないことを仮定します。従って，ローカルモデルの手続中の大域変数や仮引数にデータ転送が必要になる場合，プログラムの動作は保証されません。

```
      integer a(100,100)
!HPF$ DISTRIBUTE A(*,BLOCK)
      interface
        EXTRINSIC(Fortran_LOCAL) subroutine sub(a)
        integer a(100)
        intent(out) :: a
        end subroutine
      end interface
        :
!HPF$ INDEPENDENT
      do i=1,100
        call sub(a(:,i))
      enddo
        :
      end
      EXTRINSIC(Fortran_LOCAL) subroutine sub(a)
      integer a(100)
      intent(out) :: a
      do i=1,100
        a(i) = i
      enddo
      end
```

図 64　Fortran_LOCAL 手続の INDEPENDENT ループ中での引用

4.2.5　ON-HOME-LOCAL 指示構文 および 指示文

HPF コンパイラが DO ループを並列化する場合，分割配置された 1 つの配列を基準として，各抽象プロセッサが，その基準配列の所有部分だけをアクセスするように，ループの繰返しを各抽象プロセッサに割り当てます。このような基準配列をホーム配列 と言います。HPF コンパイラによるホーム配列の選択が最適ではない場合，無駄なデータ転送が発生します。図 65 の境界処理ループでは，DO 変数 i が，配列 a の分散されていない 1 次元目に対応しているので，全ての抽象プロセッサがループ全体を重複して実行します。一方，分散されている 2 次元目の添字は 1 なので，アクセスされるデータは最初の抽象プロセッサ上にしか配置されていないので，データ転送が必要となります。

```
      real a(100,100)
!HPF$ DISTRIBUTE a(*,BLOCK)
        :
      do i=1,99
        a(i,1) = a(i,1) + a(i+1,1)
      enddo
```

図 65　境界処理のループ

このような場合，図 66 のように ON-HOME-LOCAL 指示構文を指定して，部分配列 a(:,1)が配置されている抽象プロセッサだけでループ全体を実行すればデータ転送が不要であることを，HPF コン

パイラに教えると実行性能が向上します。

```
      real a(100,100)
!HPF$ DISTRIBUTE a(*,BLOCK)
            :
!HPF$ ON HOME(a(:,1)), LOCAL BEGIN
      do i=1,99
        a(i,1) = a(i,1) + a(i+1,1)
      enddo
!HPF$ END ON
```

図 66　境界処理ループ全体を囲む ON-HOME-LOCAL 指示構文

図 67 は，Compressed Row Storage (CRS)形式の疎行列 a に対する行列ベクトル積を行うループです。4 つ抽象プロセッサで並列実行する場合にデータ転送が発生しないよう，図 68 のように各配列を分散してしています。しかし，BLOCK 分散された配列と GEN_BLOCK 分散された配列とが同じループ中でアクセスされる場合，データ転送がないことを HPF コンパイラが自動的に判定することは困難です。

```
      real a(5), v(4), r(4)
      integer rst(5), cidx(5)
      integer, parameter :: m(4) = (/1,1,2,1/)
!HPF$ PROCESSORS p(4)
!HPF$ DISTRIBUTE r(BLOCK) ONTO p
!HPF$ DISTRIBUTE (GEN_BLOCK(m)) ONTO p :: a, cidx
            :
      do i=1,4
        r(i) = 0.0
        do j = rst(i), rst(i+1)-1
          r(i) = r(i) + a(j) * v(cidx(j))
        enddo
      enddo
```

図 67　疎行列の行列ベクトル積

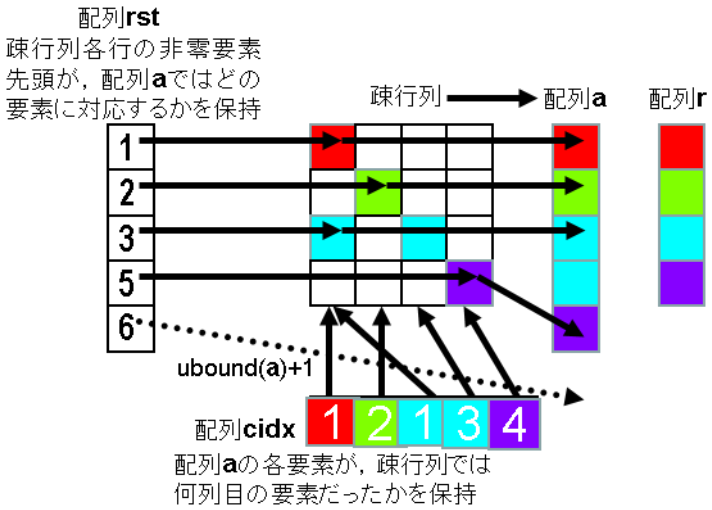

図 68　CRS 形式の疎行列の行列ベクトル積のデータマッピング

そこで，図 69 のように，do i の各繰返しを，配列 r(i)をホーム配列として実行すれば，配列 a および cidx に対するデータ転送が必要ないことを明示すると，効率よく並列実行できます。

```
        real a(5), v(4), r(4)
        integer rst(5), cidx(5)
        integer, parameter :: m(4) = (/1,1,2,1/)
!HPF$ PROCESSORS p(4)
!HPF$ DISTRIBUTE r(BLOCK) ONTO p
!HPF$ DISTRIBUTE (GEN_BLOCK(m)) ONTO p :: a, cidx
              :
        do i=1,4
!HPF$    ON HOME(r(i)), LOCAL(a, cidx) BEGIN
           r(i) = 0.0
           do j = rst(i), rst(i+1)-1
              r(i) = r(i) + a(j) * v(cidx(j))
           enddo
!HPF$    END ON
        enddo
```

図 69　疎行列の行列ベクトル積に対する ON-HOME-LOCAL 指示構文

ON-HOME-LOCAL 指示構文の対象が，1 つの実行文 または 実行構文の場合には，キーワード **BEGIN** および **END ON** を省略した ON-HOME-LOCAL 指示文も使えます。

ON-HOME-LOCAL 指示構文 および ON-HOME-LOCAL 指示文の構文は，図 70 のとおりです。

ON-HOME-LOCAL 指示構文

!HPF$ ON HOME(<部分配列>) [, LOCAL[(*v*,…)]] BEGIN

　　<実行文 または 実行構文の列>

!HPF$ END ON

- <部分配列>が配置されている抽象プロセッサが，<実行文 または 実行構文の列>を実行する
- v は，データ転送が不要な変数の名前。LOCAL だけを指定した場合，<実行文 または 実行構文の列>でアクセスされる全ての変数に対するデータ転送が不要であることを指定する

ON-HOME-LOCAL 指示文

!HPF$ ON HOME(<部分配列>) [, LOCAL[(*v*,…)]]

- <部分配列>が配置されている抽象プロセッサが，直後の実行文 または 実行構文を実行する。

図 70　ON-HOME-LOCAL 指示構文，指示文

4.2.6　SHADOW 指示文 および REFLECT 指示文

図 71 において，do i のループを左辺の b(i)をホーム配列として並列化した場合，抽象プロセッサの境界部分の計算では，図 72 のように，隣接する抽象プロセッサ上に配置されている配列 a の要素を引用するので，隣接抽象プロセッサ間でのデータ転送が必要です。

```
      real a(12), b(12)
!HPF$ PROCESSORS p(4)
!HPF$ DISTRIBUTE (*,BLOCK) ONTO p :: a, b
        :
      do i=2,11
        b(i) = a(i-1) + a(i) + a(i+1)
      enddo
```

図 71　隣接要素参照ループ

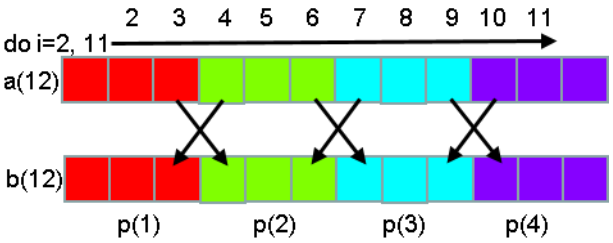

図 72　隣接抽象プロセッサ間のデータ引用

このようなデータ転送に備えて，隣接する抽象プロセッサから受け取ったデータを格納するためのバッファ領域を余分に割り付けておけば，図 73 のように，隣接する抽象プロセッサ間だけの効率の良いデータ転送を行えます。そして，ループ実行時にこのバッファ領域の値を引用すれば，図 74 のように，データ転送なしで効率よく並列実行できます。このようなバッファ領域をシャドウ領域，隣接する抽象プロセッサ間のデータ転送をシフト転送と呼びます。

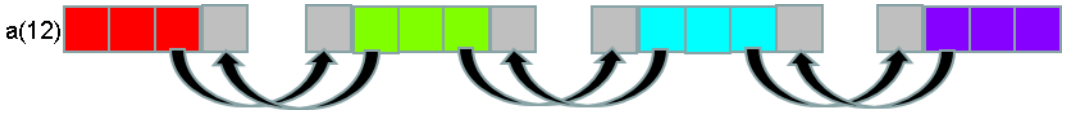

図 73　シフト転送

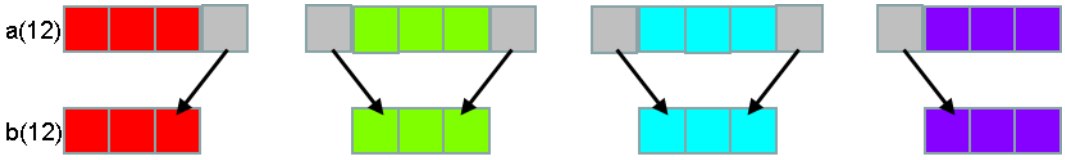

図 74　シャドウ領域の値を引用した並列実行

HPF コンパイラは，BLOCK 分散または GEN_BLOCK 分散された次元に対して，既定値では幅 4 のシャドウ領域を割り付け，自動的にシフト転送を行います。

しかし，図 75 の例では，隣接参照の幅 n が，シャドウ領域の幅に含まれるかどうかがコンパイル時には分からないので，自動的にシフト転送を行うことはできません。

```
        subroutine sub(a,b,n)
        real a(100), b(100)
!HPF$ PROCESSORS p(4)
!HPF$ DISTRIBUTE (BLOCK) ONTO p :: a, b
          :
        do i=2,99
          b(i) = a(i) + a(i+n)
        enddo
```

図 75　隣接参照の幅が実行時に決まる例

もし，プログラマが，n の値が.1 または-1 であることをわかっている場合は，次のように HPF 指示

文を挿入すれば，効率の良いシフト転送だけでループを並列実行できます。

1. シャドウ領域の幅を，SHADOW 指示文で明示的に宣言する。
2. ループ実行の前に，REFLECT 指示文で，シフト転送を行う。
3. ループは，b(i)をホーム配列とすればデータ転送なしで実行できることを，ON-HOME-LOCAL 指示文で明示する。

上記の指示文を挿入した HPF プログラムは，図 76 のようになります。

```
      subroutine sub(a,b,n)
      real a(100), b(100)
!HPF$ PROCESSORS p(4)
!HPF$ DISTRIBUTE (BLOCK) ONTO p :: a, b
!HPF$ SHADOW (1) :: a
        :
!HPF$ REFLECT a

      do i=2,99
!HPF$   ON HOME(b(i)), LOCAL
        b(i) = a(i) + a(i+n)
      enddo
```

図 76　SHADOW 指示文，REFLECT 指示文，および ON-HOME-LOCAL 指示文の指定

SHADOW 指示文の構文は，図 77 のとおりです。仮引数に SHADOW 指示文を指定する場合，対応する実引数とシャドウ幅が異なると，シャドウ幅を合わせるためのコピーが発生するので，可能であれば，対応する実引数にも同一の SHADOW 指示文を指定してください。

!HPF$ SHADOW *a*(<シャドウ幅>,…)

または

!HPF$ SHADOW (<シャドウ幅>,…) :: *a*,…

- *a* は，配列の名前
- <シャドウ幅>は，*n* または *l*: *u* で，各次元の下方向，上方向のシャドウ幅(定数値)を宣言する。*n* は，*n*：*n* と同じ意味。

図 77　SHADOW 指示文

REFLECT 指示文の構文は，図 78 のとおりです。

!HPF$ REFLECT [(<シャドウ幅>,…)] [::] *a,…*

- *a* は，配列の名前。宣言部で SHADOW 指示文を指定する必要がある。
- <シャドウ幅>,… を指定すると，シャドウ領域のうち，指定された幅だけにシフト転送を行う。これを部分 REFLECT 指示文と呼びます。

図 78　REFLECT 指示文

4.3　拡張組込み手続

本節では，HPF コンパイラが用意している拡張組込み手続の仕様を説明します。

4.3.1　計時手続

- **HPF_LOCAL_WCLOCK(ATIME)**
  - 機能

    プログラムを実行している各抽象プロセッサが，それぞれ実時間時計の値を返します。抽象プロセッサ間での同期が行われないので，測定値は，一般に抽象プロセッサ毎に異なる値を取ります。特定の計算区間における負荷バランスを知ることができます。
  - 分類

    サブルーチン
  - 引数

    ATIME

    倍精度実数型配列でなければなりません。INTENT(INOUT)引数とします。有効域の宣言部において，全ての次元で BLOCK 分散を指定した DISTRIBUTE 指示文に指定しなければなりません。形状は，分散先のプロセッサ配列と同一でなければなりません。

    配列 **ATIME** の各要素には，対応する抽象プロセッサの現在時刻が秒単位で設定されます。設定される値は非負とします。
  - 例示

```
        double precision t1(2), t2(2)
        integer a(100)
!HPF$ PROCESSORS p(2)
!HPF$ DISTRIBUTE (BLOCK) ONTO p :: t1, t2
            :
        call HPF_LOCAL_WCLOCK(t1)
        do i=1,100
          a(i) = i
        enddo
        call HPF_LOCAL_WCLOCK(t2)
```

t2(1) – t1(1)および t2(2)-t1(2)は，それぞれ抽象プロセッサ p(1)および p(2)が，ループの実行に要する秒単位の経過時間となります。

- **HPF_WCLOCK(TIME)**
  - 機能

    プログラムを実行している全抽象プロセッサが同期をとった後，実時間の計測を行い，その値を一致させるためのデータ転送を行います。測定値は，全ての抽象プロセッサ上で同一となりますが，同期およびデータ転送のオーバヘッドを伴うので，比較的大きな計算区間の計測に向いています。
  - 分類

    サブルーチン
  - 引数

    TIME

    倍精度実数型スカラでなければなりません。INTENT(OUT)引数とします。

    抽象プロセッサの現在時刻が秒単位で設定されます。設定される値は非負とします。
  - 例示

```
double precision t1, t2
integer a(100)
      :
call HPF_WCLOCK(t1)
call sub()
call HPF_WCLOCK(t2)
```

t2 – t1 は，サブルーチン sub の実行に要する秒単位の経過時間となります。

4.4　Fortranコードのクリーンナップ

HPF は Fortran 95 の拡張仕様なので，Fortran 95 プログラムは，基本的にはそのまま HPF プログラムとして実行できますが,次のような古い FORTRAN から引き継いだ機能を使っている場合は，修正が必要です。

- EQUIVALENCE 文や COMMON 文などによって，複数の変数が，同じメモリ領域を共有することがあります（記憶列結合)。
- 配列要素の並びの順序が決まっています(順序結合)。例えば,形状(2,3)の配列 a の要素は,a(1,1)，a(2,1)，a(1,2)，a(2,2)，a(1,3)，a(2,3) の順に並んでいます。

HPF では，配列は，分散メモリ上にばらばらに分割配置されるので，これらの性質は維持することができず，分割配置する変数には次のような制限が付きます。

- EQUIVALENCE 文中には指定できません。
- COMMON 文中に宣言する場合，型，形状，および データマッピングなどの属性が全ての出現で一致していなければなりません。
- 実引数と対応する仮引数の型および形状は一致していなければなりません。
- 実引数が配列要素(例えば a(1,2))の場合，対応する仮引数は，配列であってはなりません(アドレス渡し。図 79 参照)。つまり，実引数が配列要素の場合，仮引数はスカラ変数でなければなりません。
- 大きさ引継ぎ配列(a(n,*)のように，最終次元を*で宣言する仮配列引数。擬寸法配列とも呼ばれます)は，使用できません。

```
real a(n,n)
do i=1,n
   call sub(a(1,i),n)
enddo
end

subroutine sub(a,n)
real a(n)
```

図 79　アドレス渡し（HPF では許されません）

既存の Fortran プログラムを HPF で並列化する場合，まずはこれらの記述を次のように修正した後，HPF 指示文を挿入してください。

- 分割配置する配列に対する EQUIVALENCE 文は削除します。大きな配列を宣言し，それを部分的に使用しているような場合，割付け配列 または 自動割付け配列のように実行時に宣言範囲を決められる機能を使って，実際に利用する型および形状で配列を宣言するようにします。
- 共通ブロック変数は，データマッピングの指定も含めて，全ての出現で宣言を同一にします。インクルードファイルやモジュール中に COMMON 文を記述すると，書き忘れ・書き誤りが防げます。
- 実引数と対応する仮引数の次元数および各次元の寸法を一致させます。次のような修正が必要になる場合があります。
  - 図 79 のようなアドレス渡しは，図 80 のような部分配列実引数に修正して，配列を渡すことを明示します。
  - 図 81 のような大きさ引継ぎ配列(擬寸法配列)は，図 82 のような整合配列 または 形状引継ぎ配列に修正します。

```
real a(n,n)
do i=1,n
   call sub(a(:,i),n)
enddo
end

subroutine sub(a,n)
real a(n)
```

図 80　部分配列実引数

```
real a(n,n)
call sub(a,n)
end

subroutine sub(a,n)
real a(n,*)
```

図 81　大きさ引継ぎ配列

```
real a(n,n)
call sub(a,n)
end

subroutine sub(a,n)
real a(n,n)
```

図 82　整合配列

既存の Fortran プログラムには，このような制限事項に直面して，HPF で並列化するのが難しかったり，ひどく手間がかかったりするコードもありますが，並列化しなくてもよい手続については，次のいずれかの方法でコンパイル・リンクすれば，プログラムの修正は必要ありません。

- 分割配置された配列や入出力がない手続なら，HPF 手続としてそのまま HPF コンパイラでコンパイルします。
- 外来手続機能を使って，Fortran 手続として HPF コンパイラでコンパイルします。この場合，引用元の HPF 手続中には，Fortran 手続の引用であることを 外来接頭辞で明示するため引用仕様宣言(interface block)などを記述します。外来手続機能については, 4.2.4 項を参照してください。
- Fortran コンパイラを使ってオブジェクトやアーカイブにして，HPF プログラムとリンクします。この方法は，既存の Fortran 用のライブラリなどを HPF プログラムから呼び出す場合にも使用できます。

第5章　　チューニングとデバッグ

本章では，HPF プログラムのチューニングおよびデバッグ方法について説明します。

5.1　チューニング

5.1.1　並列化情報リスト

HPF コンパイラオプション-Mlist2 を指定してしてコンパイルすると, HPF コンパイラによる並列化とデータ転送の情報を含む 並列化情報リストファイルが生成されます。並列化情報リストファイルのサフィックスは，.lst です。

例えば，図 83 の HPF プログラム対しては，図 84 のような並列化情報リストが生成されます。並列化情報リスト中の各記号の意味は，表 6 のとおりです。

```
      real :: a(100,100) = 0
!HPF$ DISTRIBUTE a(*,block)

      do i=1,99
        do j=1,100
          a(j,i) = a(j,i) + a(j,i-1)
        enddo
      enddo

      do j = 1,100
         do i = 1,100
            x = max(x,a(i,j))
         end do
      end do

      write(*,*)x
```

図 83　HPF プログラム例

```
(     1)                                real :: a(100,100) = 0
(     2)                             !HPF$ DISTRIBUTE a(*,block)
(     3)
(     4) <S>----------              do i=1,99
          COMM: SFT [a] [LINO: 5 in src.hpf]
(     5) <N>----------                 do j=1,100
(     6)   |                                a(j,i) = a(j,i) + a(j,i-1)
(     7)   +----------                 enddo
(     8)                               enddo
(     9)
          COMM: RED [x] [LINO: 10 in src.hpf]
          HOME: a(:,j)
(    10) <P>----------                do j = 1,100
(    11)   |<I>----------                  do i = 1,100
(    12)   |                                    x = max(x,a(i,j))
(    13)   |                               end do
(    14)   +----------                end do
(    15)
(    16)                               write(*,*)x
(    17)                               end
```

図 84　並列化情報リストの例

表 6　並列化情報リスト中の記号

記号	意味
(　1)	HPF ソースファイルの行番号。
COMM: SFT [a] **[LINO:** 5 **in** src.hpf]	HPF コンパイラによりデータ転送が生成されたことを示します。 次の形式で表示されます。 **COMM:** データ転送の種類 [対象変数名] **[LINO:**行番号] データ転送の種類は，次のとおりです。 **RED**：リダクション **SFT**：シフト **CPY**：配列のコピー **G/S**　：収集/拡散 **SCL**：スカラ変数に対するデータ転送 次のように，**COMM:**を含む行を抽出することにより，生成されたデータ転送の一覧が得られ， 無駄なデータ転送の有無をチェックできます。 %>grep "COMM:" src.lst

	上記のデータ転送のうち，マーク **RED** および **SFT** は，通常はあまり問題とはなりませんが，マーク **CPY** は発生しないように改善できれば，プログラムの高速化につながります。マーク **G/S** および **SCL** は，極めてオーバヘッドが高い場合が多いので，発生しないように改善すべきです。
<S>	並列化できないと判定されたループであることを示します。 次のように，<S>を含む行を抽出することにより，自動では並列化可能と判定されなかったループの一覧が得られます。 %>grep "<S>" src.lst 並列化されるべきループが，並列化可能と判定されていない場合， INDEPENDENT 指示文を指定すると，並列化される場合があります。
<N>	並列化できると判定されたが，並列化されなかったループであることを示します。このようなループは，付随してデータ転送が生成されていない限り，性能低下の要因とはなりませんが，ループ中に出現する配列のデータマッピングを変更すると，並列化できる場合があります。 以下のように，<N>を含む行を抽出することにより， 並列化されなかった並列化可能なループの一覧が得られます。 %>grep "<N>" src.lst
<P>	HPF コンパイラによって並列化されたループであることを示します。
<I>	HPF コンパイラによって並列化可能と判定されたループであることを示します。ループがリダクションの依存を含む場合， <I>の代わりに <R>が表示されます。
HOME: a(:,j)	直後のループネストのホーム配列(並列化の基準配列)を示します。

HPF コンパイラオプション-Mlist3 を指定してコンパイルすると， HPF コンパイラオプション-Mlist2 により出力される情報に加えて， HPF コンパイラにより並列化された中間コードイメージを

出力できます。

例えば，図 83 の 2 番目の do ループネストに対して 出力される中間コードイメージは，図 85 のようになります。 -Mlist3 オプションにより， より詳細な並列化状況を参照できます。

```
(      9)
           COMM: RED [x] [LINO: 10 in src.hpf]
           HOME: a(:,j)
(     10) <P>----------            do j = 1,100
(     11)   |<I>----------              do i = 1,100
(     12)   |                                 x = max(x,a(i,j))
(     13)   |                             end do
(     14)   +----------            end do
          .
          .          x$ind = x
          .          j$indl = a$sd(84)
          .          j$indu = a$sd(85)
          .          pghpf_saved_local_mode = pghpf_local_mode
          .          pghpf_local_mode = 1
          .!NEC$nosync
          .!NEC$shortloop
          .          do j = j$indl, j$indu
          .!NEC$nosync
          .              do i = 1, 100
          .                  x$ind = max(x$ind,a(i,j))
          .              enddo
          .          enddo
          .          pghpf_local_mode = pghpf_saved_local_mode
          .          call pghpf_global_maxval(x$ind,a,125_8,pghpf_type(27),a$sd,
          .         +pghpf_type(26))
          .!         call .reduce_maxval(x$ind,a,125_8)
          .          x = x$ind
          .
```

図 85　詳細並列化情報リストの例

なお，配列構文に対しては，並列化情報の記号は出力されません。

HPF コンパイラオプション-Mlist2 オプションまたは-Mlist3 および NEC Fortran コンパイラの編集リスト出力オプション(-report-format または –report-all)を同時に指定すると，図 86 のように，HPF コンパイラによる並列化情報マークの右隣に， NEC Fortran コンパイラによるループ最適化情報も出力されます。(ただし，HPF コンパイラオプション-Mftn を指定していない場合に限ります。)

```
(     1)                                real :: a(100,100) = 0
(     2)                          !hpf$ distribute a(*,block)
(     3)
(     4) <S>+------------          do i=1,99
           COMM: SFT [a] [LINO: 5 in src.hpf]
(     5) <N>V------------            do j=1,100
(     6)   |                              a(j,i) = a(j,i) + a(j,i-1)
(     7)   +--------------           enddo
(     8)                               enddo
(     9)
           COMM: RED [x] [LINO: 10 in src.hpf]
           HOME: a(:,j)
(    10) <P>P------------           do j = 1,100
(    11)   |<I>V----------             do i = 1,100
(    12)   |                               x = max(x,a(i,j))
(    13)   |                           end do
(    14)   +--------------         end do
(    15)
(    16)                               write(*,*)x
(    17)                               end
```

図 86　ループ最適化情報付き並列化情報リスト

最適化情報の記号の意味は，NEC Fortran コンパイラの編集リストと同じです。例えば，P は共有並列化されたループ， V はベクトル化されたループを意味しています。 さらに，インライン展開(I)や部分ベクトル化(S)の情報も， 対応するソースコード行開始位置の左側に出力されます。 詳細は，「Fortran コンパイラ ユーザーズガイド」を参照して下さい。 なお，HPF コンパイラによる分散並列化時にループが分割され,分割された各ループの NEC Fortran コンパイラによる最適化状況が様々な場合， 記号”M”が出力されます。

5.1.2　診断メッセージ

HPF コンパイラオプション-Minfo を指定してコンパイルすると,診断メッセージが出力されます。特に注意すべき診断メッセージは，次のとおりです。

- expensive communication
  オーバヘッドの大きなデータ転送が生成されたことを意味しています。

- Array "配列名" not aligned with home array; array copied

配列のデータマッピングが,ループ並列化の基準配列のデータマッピングと整合していないので,テンポラリ領域にコピーされたことを示しています。 多くの場合，データ転送を伴います。

- communication is generated: array copy
  配列がテンポラリ領域にコピーされたことを示しています。多くの場合,データ転送を伴います。

5.1.3　FTRACE リージョン指定機能の使用法

FTRACE リージョン指定機能を HPF から利用する場合，図 87 のようにサブルーチン FTRACE_REGION_BEGIN 及び FTRACE_REGION_END が Fortran_LOCAL 外来手続であることを明示的引用仕様中で宣言する必要があります。

```
interface
   extrinsic(Fortran_LOCAL) subroutine ftrace_region_begin(label)
   character(*) label
   end subroutine
   extrinsic(Fortran_LOCAL) subroutine ftrace_region_end(label)
   character(*) label
   end subroutine
end interface
```

図 87　リージョン指定機能を利用するための明示的引用仕様

性能プロファイラ FTRACE の詳細については,「PROGINF/FTRACE ユーザーズガイド」を参照してください。

5.1.4　チューニング例

本項では，HPF プログラムの代表的なチューニング例を紹介します。

- 作業配列を含むループの並列化
  図 88 の例では， 作業配列 tmp が，do k のループ中で繰返し代入されているので， 自動的に並列化することはできません。

```
      integer tmp(100),a(100,100)
!HPF$ DISTRIBUTE a(*,BLOCK)
        :
      do k = 2, nz - 1
        do j = 2, ny - 1
          do i = 1, 100
            tmp(i) = i
          enddo
          a(j,k) = tmp(i) + tmp(i+1)
        enddo
      enddo
      write(*,*)a
      end
```

図 88　作業配列を含むループ

図 89 のように,作業配列 tmp を NEW 節中に記述した INDEPENDENT 指示文を指定すると,並列化が可能になります。

```
      integer tmp(100),a(100,100)
!HPF$ DISTRIBUTE a(*,BLOCK)
        :
!HPF$ INDEPENDENT, NEW(tmp,i,j)
      do k = 2, nz - 1
        do j = 2, ny - 1
          do i = 1, 100
            tmp(i) = i
          enddo
          a(j,k) = tmp(i) + tmp(i+1)
        enddo
      enddo
      write(*,*)a
      end
```

図 89　作業配列を NEW 節中に指定した INDEPENDENT 指示文

- ループ分割

図 90 のループは，確定される左辺の配列 a, b の分散次元である 2 次元目の添字にずれがあるので，どちらかにデータ転送が必要です。確定される配列に対してデータ転送が発生する場合，一旦テンポラリ領域にデータを受信してから,データ領域にコピーする,といった処理が行われるので,オーバヘッドが大きくなりがちです。

```
      real a(10,10), b(10,10), c(10,10)
!HPF$ DISTRIBUTE (*,BLOCK) :: a, b, c
        :
      do j=1,9
        do i=1,99
          a(i+1,j) = -c(i+1,j+1)
          b(i,j+1) = c(i+1,j+1)
        enddo
      enddo
```

図 90　左辺の添字にずれがある場合

そこで，図 91 のようにループを 2 つに分割して，各ループの左辺の配列の分散次元の添字を 1 種類にすれば，1 番目のループは，右辺の配列 c に対するシフト転送だけ，2 番めのループは，データ転送なしで効率のよい並列実行が可能です。

```
      real a(10,10), b(10,10), c(10,10)
!HPF$ DISTRIBUTE (*,BLOCK) :: a, b, c
        :
      do j=1,9
        do i=1,9
          a(i+1,j) = -c(i+1,j+1)
        enddo
      enddo
      do j=1,9
        do i=1,9
          b(i,j+1) = c(i+1,j+1)
        enddo
      enddo
```

図 91　ループ分割

- 境界処理のデータ転送削減

  図 92 のように，分割配置された次元の両端だけを参照する 境界処理 のループは，指示文の指定がないと，全ての抽象プロセッサが実行に参加して， 効率の悪いデータ転送が発生します。

```
      double precision a(100,100)
!HPF$ PROCESSORS p(2)
!HPF$ DISTRIBUTE a(*,BLOCK) ONTO p

      do i=1,100
        a(i,1) = a(i,2)
        a(i,100) = a(i,99)
      enddo
```

図 92　境界処理のループ

図 93 のように，ON-HOME-LOCAL 指示文を指定して，境界の要素が分割配置されている抽象プロセッサだけが実行すれば，データ転送の発生を抑制できます。。

```
      double precision a(100,100)
!HPF$ PROCESSORS p(2)
!HPF$ DISTRIBUTE a(*,BLOCK) ONTO p

      do i=1,100
!HPF$   ON HOME(a(:,1)), LOCAL
        a(i,1) = a(i,2)
!HPF$   ON HOME(a(:,100)), LOCAL
        a(i,100) = a(i,99)
      enddo
```

図 93　境界処理に対する ON-HOME-LOCAL 指示文の指定

- 境界処理部分の分割

  図 94 のように，１つのループ中で，IF 文によって一部分で境界処理を行うと，ループの並列化を阻害したり，効率の悪いデータ転送が発生したりする恐れがあります。

```
      parameter(n=100)
       real a(n,n),b(n,n)
!HPF$ DISTRIBUTE (*,BLOCK) :: a,b

      do j=1,n
        if(j.eq.n)then
          do i=1,n
            a(i,j) = 0.9
          enddo
        else
          do i=1,n
            a(i,j) = b(i,j) + b(i,j+1)
          enddo
        endif
      enddo
```

図 94　ループの一部が境界処理の例

図 95 のように，境界処理部分は，別のループとして記述し， ON-HOME-LOCAL 指示構文を指定すると，効率の良い処理が可能です。

```
      parameter(n=100)
      real a(n,n),b(n,n)
!HPF$ DISTRIBUTE (*,BLOCK) :: a,b

       do j=1,n-1
         do i=1,n
           a(i,j) = b(i,j) + b(i,j+1)
         enddo
       enddo

       j=n
!HPF$ ON HOME(a(:,j)), NEW(i), LOCAL(a) BEGIN
       do i=1,n
         a(i,j) = 0.9
       enddo
!HPF$ END ON
```

図 95　境界処理部分を分割したループ

- 境界処理部分の添字指定

  図 96 のように，分割配置された次元に対する境界処理の添字を定数で記述すると， DO 変数と添字との対応がとれず， 効率の悪いデータ転送が発生する恐れがあります。

```
      parameter(n=100)
      real a(n,n),b(n,n)
!HPF$ DISTRIBUTE (*,BLOCK) :: a,c

      do j=1,n
        if(j.eq.2)then
          do i=1,n
            a(i,1) = a(i,1) - b(i)*c(i,1)
          enddo
        endif
      enddo
```

図 96　添字を定数で記述した境界処理ループ

図 97 のように，添字には DO 変数の一次式を指定してください。

```
!HPF$ DISTRIBUTE (*,BLOCK) :: a,c
      do j=1,n
        if(j.eq.2)then
          do i=1,n
            a(i,j-1) = a(i,j-1) - b(i)*c(i,j-1)
          enddo
        endif
      enddo
```

図 97　添字を DO 変数の 1 次式で記述した境界処理ループ

- 異なるデータマッピングをもつ実引数

  図 98 のように，同一の手続が，さまざまなデータマッピングをもつ実引数と共に呼び出されると， いずれかの手続呼出し時にデータ転送が発生し，効率の悪いコードになる恐れがあります。

```
      double precision a(100,100),b(100,100)
!HPF$DISTRIBUTE a(*,BLOCK)
      call sub(a)
      call sub(b)
      end
```

図 98　さまざまなデータマッピングをもつ実引数

この様な場合，図 99 のように呼ばれ側手続のコピーを作成し，実引数のデータマッピングのそれぞれに対応して，実引数と同一のデータマッピングを仮引数に指定すると，実行性能が向上します。 このような最適化を手続クローニングといいます。

```
      double precision a(100,100),b(100,100)
!HPF$DISTRIBUTE a(*,BLOCK)
      call sub1(a)
      call sub2(b)
      end

      subroutine sub1(a)
      double precision a(100,100)
!HPF$DISTRIBUTE a(*,BLOCK)
        :
      end

      subroutine sub2(b)
      double precision b(100,100)
        :
      end
```

図 99　実引数のデータマッピングに対応した手続のコピー

- 仮引数のデータマッピング

  図 100 の例では，分割配置された実引数と結合する仮引数が分割配置されていないため，手続呼出し時に実引数と仮引数のデータマッピングを一致させるためデータ転送が発生します。

```
        program main
        real a(100,100)
!HPF$ DISTRIBUTE a(*,BLOCK)
        call sub(a)
        end

        subroutine sub(a)
        real a(100,100)！ マッピング指定無し
```

図 100　仮引数のデータマッピングが実引数と異なる場合

HPF 実行時オプション-hpf -commmsg を指定して実行すると，手続呼出し時にデータ転送が発生した場合，次のような診断メッセージが発行されるので，手続境界でのデータ転送の発生をチェックできます。

"a": Communication occurs at procedure boundary PROG=sub ELN=7 Called from main ELN=4

このような場合，図 101 のように，実引数と同一のデータマッピングを仮引数に指定すると，実行性能が向上します。

```
        program main
        real a(100,100)
!HPF$ DISTRIBUTE a(*,BLOCK)
        call sub(a)
        end

        subroutine sub(a)
        real a(100,100)
!HPF$ DISTRIBUTE a(*,BLOCK)
```

図 101　仮引数へのデータマッピングの指定

- 入出力の記述方法

  図 102 のように，1 要素毎に入出力を行うと，入出力の効率が低下します。

```
        write(13,*) (a(i), b(i), i=1, n)
```

図 102　1 要素ずつの効率の悪い入出力文

特にデータサイズが大きい場合，図 103 のように，配列全体を書式無しで一括して入出力するようにして下さい。。

```
write(13,*) a, b
```

図 103　配列全体を一括で入出力する文

- Fortran を利用した入力の高速化

  入力時には，まず 1 つのプロセスがデータを読み込んでから，読み込んだデータが配置されているプロセスにデータ転送を行います，そのため，図 104 のように分割配置していない配列に対して 1 要素毎の入力を行うと，1 つのプロセスが 1 要素ずつ入力した後，全プロセスにデータ転送を行う動作を繰り返すので，極端に性能が低下します。

```
real a(n,n,n)
  :
open(10,file=’data’)
do j=1,n
  do i=1,n
    read(10,*) (a(i,j,k), k=1,n)
  enddo
enddo
close(10)
```

図 104　1 要素ずつの効率の悪い入力文（分割配置されていないデータ）

このような場合，図 105 のように入力部分を Fortran のサブルーチンに切り出し，全てのプロセスがそれぞれ Fortran のサブルーチンを実行する Fortran_LOCAL 外来手続を利用して HPF からその入力用 Fortran サブルーチンを呼び出すと(図 106 参照)，全プロセスがそれぞれ入力を行うことにより，データ転送が必要なくなるので，性能が大幅に向上します。

```
subroutine read_all_fortran(a,n,n,n)
real a(n,n,n)
integer n,n,n
open(10,file=’data’)
do j=1,n
  do i=1,n
    read(10,*) (a(i,j,k), k=1,n)
  enddo
enddo
close(10)
end subroutine
```

図 105　入力部分だけを切り出した Fortrans サブルーチン

```
real a(n,n,n)
interface
   extrinsic(Fortran_LOCAL) subroutine read_all_fortran(a,n,n,n)
   read a(n,n,n)
   integer n,n,n
   end subroutine
end interface
  :
call read_all_fortran(a,n,n,n)
```

図 106　Fortran_LOCAL 外来手続を利用した入力文の高速化

一方，図 107 のように，入力対象の配列が分割配置されている場合には，データ転送の発生自体は避けられませんが，図 108 のように，やはり入力部分を Fortran のサブルーチンに切り出し，1 つのプロセスだけが Fortran のサブルーチンを実行する Fortran_SERIAL 外来手続を利用して HPF からその入力用 Fortran サブルーチンを呼び出すと(図 109 参照)，1 つのプロセスが入力を実行した後，各プロセスへのデータ転送がまとめて行われるので，大幅な性能向上が可能です。

```
      real a(n,n,n)
!HPF$ DISTRIBUTE a(*,*,BLOCK)
        :
      open(10,file=’data’)
      do j=1,n
        do i=1,n
          read(10,*) (a(i,j,k), k=1,n)
        enddo
      enddo
      close(10)
```

図 107　1 要素ずつの効率の悪い入力文（分割配置されたデータ）

```
      subroutine read_one_fortran(a,n,n,n)
      real a(n,n,n)
      integer n,n,n
      open(10,file=’data’)
      do j=1,n
        do i=1,n
          read(10,*) (a(i,j,k), k=1,n)
        enddo
      enddo
      close(10)
      end subroutine
```

図 108　入力部分だけを切り出した Fortrans サブルーチン

```
      real a(n,n,n)
!HPF$ DISTRIBUTE a(*,*,BLOCK)
      interface
         extrinsic(Fortran_SERIAL) subroutine read_one_fortran(a,n,n,n)
         read a(n,n,n)
         integer n,n,n
         end subroutine
      end interface
         :
      call read_one_fortran(a,n,n,n)
```

図 109　Fortran_SERIAL 外来手続を利用した入力文の高速化

Fortran の外来手続機能を使用する場合，図 110 のように，まず Fortran サブルーチンを，-c オプション付きで Fortran コンパイラでコンパイルして Fortran オブジェクトファイルを生成し，HPF プログラムのコンパイル時に,その Fortran オブジェクトファイルをリンクしてください。

```
%> nfort -c read_all_fortran.f
%> ve-hpf hpfsourcefile.hpf read_all_fortran.o -o hpfexecfile
```

図 110　Fortran サブルーチンを呼び出す HPF プログラムのコンパイル

- Fortran を利用した出力の高速化

  出力時には，まず 1 つのプロセスにデータ転送を行ってから，そのプロセスがデータの出力を行います。そのため，図 111 のように，分割配置された配列に対して 1 要素毎の出力を行うと，出力用プロセスに 1 要素ずつデータ転送を行っては出力を実行する動作を繰り返すので，極端に性能が低下します。

```
      real a(n,n,n)
!HPF$ DISTRIBUTE a(*,*,BLOCK)
        :
      open(10,file=’data’)
      do j=1,n
        do i=1,n
          write(10,*) (a(i,j,k), k=1,n)
        enddo
      enddo
      close(10)
```

図 111　1 要素ずつの効率の悪い出力文

このような場合，図 112 のように出力部分を Fortran のサブルーチンに切り出し，1 つのプロセスだけが Fortran のサブルーチンを実行する Fortran_SERIAL 外来手続を利用して HPF からその出力用 Fortran サブルーチンを呼び出すと(図 113 参照)，最初に 1 つのプロセスにデータを集めてから出力を行うことにより，性能が大幅に向上します。

```
      subroutine write_one_fortran(a,n,n,n)
      real a(n,n,n)
      integer n,n,n
      open(10,file=’data’)
      do j=1,n
        do i=1,n
          write(10,*) (a(i,j,k), k=1,n)
        enddo
      enddo
      close(10)
      end subroutine
```

図 112　出力部分だけを切り出した Fortran サブルーチン

```
      real a(n,n,n)
!HPF$ DISTRIBUTE a(*,*,BLOCK)
      interface
        extrinsic(Fortran_SERIAL) subroutine write_one_fortran(a,n,n,n)
        read a(n,n,n)
        integer n,n,n
        end subroutine
      end interface
         :
      call write_one_fortran(a,n,n,n)
```

図 113　Fortran_SERIAL 外来手続を利用した出力文の高速化

- 配列に対する集計計算ループのループ順序変更

  図 114 は，do k のループが，配列 a に対する集計計算を行うループです。

```
      double precision w(100,100,100),a(100,100)
!HPF$ DISTRIBUTE w(*,*,block)

      do k=1,100
        do j=1,100
          do i=1,100
            a(i,j) = a(i,j) + w(i,j,k)
          enddo
        enddo
      enddo
```

図 114　配列のリダクションループ

コンパイル時に，NEC Fortran コンパイラのコンパイラオプション-mparallel を指定して自動共有並列化を行う場合，ループの順序は，図 115 のように，リダクションを行わない完全に並列なループを最外側に記述した方が共有並列化の効率が良くなります。この場合，HPF コンパイラは， 配列 w の分散次元に対応する do k のループで分散並列化を行い，Fortran コンパイラは，最外側の do j のループで共有並列化を行います。

```
      double precision w(100,100,100),a(100,100)
!hpf$ DISTRIBUTE w(*,*,BLOCK)

      do j=1,100
        do k=1,100
          do i=1,100
            a(i,j) = a(i,j) + w(i,j,k)
          enddo
        enddo
      enddo
```

図 115　完全に並列なループ do j を最外側に移動させる

5.2　簡便なHPFプログラム作成方法

HPF コンパイラオプション-Mautodist を指定して，逐次 Fortran プログラムをコンパイルすると，全ての配列の最終次元を BLOCK 分散した HPF プログラムとしてコンパイルできます。さらに，サブオプション**=all**[*:b*] および **=rank***n*[*:b*]を使用すると，より詳細に配列のデータマッピングを指定できます。HPF コンパイラオプション-Mlist2 オプションを同時に指定すると，指定したデータマッピングの結果，各ループが並列化されたかどうか および どの場所でどのようなデータ転送が生成されたかを，並列化情報リストでチェックできます。

本節では，これらのオプションを利用して，図 116，図 117，および 図 118 の Fortran プログラム sample.F を HPF で並列化する手順を説明します。

```
module param
parameter(n=1023,maxiter=10)
end module
```

図 116　例題プログラム: モジュール

```
      program sample
      use param
      double precision a(n,n),b(n,n),c(n,n),sum,ap
      integer idxx(n),idxy(n),ix,iy,i,j,iter
      data ap/0.0d0/

      do i=1,n
        idxx(i) = n - i + 1
        idxy(i) = n - i + 1
      enddo
      do j=2,n-1
        do i=1,n
          b(i,j) = 1.0d0
          c(i,j) = 1.0d0
        enddo
      enddo
      call bound(b)
      call bound(c)

      do iter=1,maxiter
! main loop
        do j=2,n-1
          do i=2,n-1
            ix = idxx(i)
            iy = idxy(j)
            a(i,j)=(b(i,j)+b(i-1,j)+b(i+1,j)                                    &
     &     +b(i,j-1)+b(i,j+1))*0.2d0*c(ix,iy)+ap
          enddo
        enddo
        do i=1,n
          a(1,i) = a(2,i)
          a(n,i) = a(n-1,i)
        enddo
        call bound(a)
        do j=1,n
          do i=1,n
            ix = idxx(i)
            b(ix,j)=a(i,j)*c(i,j)
            ap = ap + a(i,j)
          enddo
        enddo
      enddo

      write(*,*)ap
      end
```

図 117　例題プログラム: 主プログラム

```
subroutine bound(dummy)
use param
double precision dummy(n,n)
do i=1,n
   dummy(i,1) = dummy(i,2)
   dummy(i,n) = dummy(i,n-1)
enddo
end
```

図 118　例題プログラム: サブルーチン bound

まずは，HPF コンパイラオプション-Mautodist および –Mlist2 を指定してコンパイルします。すると,全ての配列の最終次元を BLOCK 分散した HPF プログラムに対する並列化情報リスト sample.lst が生成されます。

主プログラムに対応する並列化情報は，図 119 のようになります。データ転送が生成されたことを示す”COMM:”というマークの行に着目すると，26 行目 および 40 行目に多くのデータ転送が生成されており，このままでは効率が悪いことがわかります。効率の悪いままで実行すると，場合によっては，実行速度が，元のプログラムの数百倍・数千倍遅くなる場合もあるので，決してこのままで実行してはいけません。以下では，どのように HPF プログラムを改善するか，1 つ 1 つ解説していきます。

```
(    11) <I>----------          do i=1,n
(    12)                          idxx(i) = n - i + 1
(    13)                          idxy(i) = n - i + 1
(    14)                        enddo
(    15) <I>----------          do j=2,n-1
(    16) <I>----------            do i=1,n
(    17)                            b(i,j) = 1.0d0
(    18)                            c(i,j) = 1.0d0
(    19)                          enddo
(    20)                        enddo
(    21)                        call bound(b)
(    22)                        call bound(c)
(    23)
(    24) <S>----------          do iter=1,maxiter
(    25)                      ! main loop
          COMM: SFT [b] [LINO: 26 in sample.F]
          COMM: CPY [idxx] [LINO: 26 in sample.F]
          COMM: G/S [c] [LINO: 26 in sample.F]
          HOME: idxy(j)
(    26) <P>----------            do j=2,n-1
(    27)   |<I>----------            do i=2,n-1
(    28)   |                          ix = idxx(i)
(    29)   |                          iy = idxy(j)
(    30)   |                          a(i,j)=(b(i,j)+b(i-1,j)+b(i+1,j)
(    31)   |                       &     +b(i,j-1)+b(i,j+1))*0.2d0*c(ix,iy)+ap
(    32)   |                        enddo
(    33)   +----------            enddo
          HOME: a(:,i)
(    34) <P>----------            do i=1,n
(    35)   |                          a(1,i) = a(2,i)
(    36)   |                          a(n,i) = a(n-1,i)
(    37)   +----------            enddo
(    38)                            call bound(a)
(    39) <S>----------            do j=1,n
          COMM: CPY [idxx] [LINO: 40 in sample.F]
          COMM: CPY [a] [LINO: 40 in sample.F]
          COMM: SCL [c] [LINO: 40 in sample.F]
          COMM: SCL [a] [LINO: 40 in sample.F]
(    40) <S>----------              do i=1,n
          COMM: RED [ap] [LINO: 41 in sample.F]
(    41)                              ix = idxx(i)
(    42)                              b(ix,j)=a(i,j)*c(i,j)
(    43)                              ap = ap + a(i,j)
(    44)                            enddo
(    45)                          enddo
(    46)                        enddo
(    47)
(    48)                        write(*,*)ap
(    49)                        end
```

図 119　主プログラムの並列化情報リスト

26 行目のデータ転送情報は，図 120 のように表示されており，最終次元である 1 次元目で BLOCK 分散された配列 idxy(j)を基準に並列化された do j のループに対して生成されていることが，マーク”HOME: idxy(j)” および “<P>”からわかります。(マーク”<I>”がついている 27 行目の do i のループは，並列化可能であると自動判定されていますが，並列化はされていません。)

```
          COMM: SFT [b] [LINO: 26 in sample.F]
          COMM: CPY [idxx] [LINO: 26 in sample.F]
          COMM: G/S [c] [LINO: 26 in sample.F]
          HOME: idxy(j)
(    26) <P>----------              do j=2,n-1
(    27)   |<I>----------              do i=2,n-1
(    28)   |                              ix = idxx(i)
(    29)   |                              iy = idxy(j)
(    30)   |                              a(i,j)=(b(i,j)+b(i-1,j)+b(i+1,j)
(    31)   |                       &     +b(i,j-1)+b(i,j+1))*0.2d0*c(ix,iy)+ap
```

図 120　26 行目のデータ転送情報

3 つのデータ転送のうち，最初の”COMM: SFT [b]”は，比較的効率の良いシフト転送なので，通常は問題ありません。2 番めの”COMM: CPY [idxx]”は，最終次元である 1 次元目で BLOCK 分散されている配列 idxx が,並列化されている do j のループに対応しない,添字 i により引用されている(idxx(i))ために発生しています。並列化されるループの DO 変数と異なる添字でアクセスされている次元は，分散しないほうがよいので，配列 idxx に対しては，全ての次元を分散しないことを示す図 121 のような DISTRIBUTE 指示文を挿入します。

```
!HPF$ DISTRIBUTE (*) :: idxx
```

図 121　1 次元配列 idxx を分散しない DISTRIBUTE 指示文

3 番めの”COMM: G/S [c]”は，最終次元である 2 次元目で BLOCK 分散されている配列 c が，do j で並列化されているループ中で，間接添字 ix および iy により引用されている(c(ix,iy))ために発生しています。並列化されるループの DO 変数と添字が異なるので，配列 c もやはり全ての次元を分散しないように，図 122 のような DISTRIBUTE 指示文を挿入します。

```
!HPF$ DISTRIBUTE (*,*) :: c
```

図 122　2 次元配列 c を分散しない DISTRIBUTE 指示文

40 行目のデータ転送情報は, 図 123 のように表示されており, 並列化できない”<S>”と判定された do j ループおよび do i ループの間に生成されていることがわかります。

```
(    39) <S>----------              do j=1,n
         COMM: CPY [idxx] [LINO: 40 in sample.F]
         COMM: CPY [a] [LINO: 40 in sample.F]
         COMM: SCL [c] [LINO: 40 in sample.F]
         COMM: SCL [a] [LINO: 40 in sample.F]
(    40) <S>----------                do i=1,n
         COMM: RED [ap] [LINO: 41 in sample.F]
(    41)                                  ix = idxx(i)
(    42)                                  b(ix,j)=a(i,j)*c(i,j)
(    43)                                  ap = ap + a(i,j)
```

図 123　40 行目のデータ転送情報

do j のループは，実際にはスカラ変数 ap に対する集計計算(総和)を行う並列化可能なループですが，HPF コンパイラは，自動的に並列化可能であることを判定できなかったので，REDUCTION 節付きの INDEPENDENT 指示文を図 124 のように挿入します。作業変数 ix や内側の DO ループの DO 変数に対する NEW 節も指定しておきます。

```
!HPF$ INDEPENDENT, NEW(ix,i), REDUCTION(ap)
          do j=1,n
            do i=1,n
              ix = idxx(i)
              b(ix,j)=a(i,j)*c(i,j)
              ap = ap + a(i,j)
            enddo
          enddo
```

図 124　REDUCTION 節付き INDEPENDENT 指示文

ここで，HPF コンパイラオプション-Mautodist および –Mlist2 を指定して再びコンパイルします。すると, 主プログラムに対する並列化情報リスト sample.lst は, 図 125 のようになります。今度は，効率のよいシフト転送”COMM: SFT [b]” および スカラ変数に対するリダクション転送”COMM: RED [ap]”だけで効率よく並列化できている(マーク”<P>”)ことがわかります。

```
(    10)                        !HPF$ DISTRIBUTE (*) :: idxx
(    11)                        !HPF$ DISTRIBUTE (*,*) :: c
(    12)
(    13) <I>----------          do i=1,n
(    14)                              idxx(i) = n - i + 1
(    15)                              idxy(i) = n - i + 1
(    16)                            enddo
(    17) <I>----------          do j=2,n-1
(    18) <I>----------            do i=1,n
(    19)                                b(i,j) = 1.0d0
(    20)                                c(i,j) = 1.0d0
(    21)                              enddo
(    22)                            enddo
(    23)                            call bound(b)
(    24)                            call bound(c)
(    25)
(    26) <S>----------          do iter=1,maxiter
(    27)                        ! main loop
          COMM: SFT [b] [LINO: 28 in sample.F]
          HOME: idxy(j)
(    28) <P>----------            do j=2,n-1
(    29)   |<I>----------              do i=2,n-1
(    30)   |                              ix = idxx(i)
(    31)   |                              iy = idxy(j)
(    32)   |                              a(i,j)=(b(i,j)+b(i-1,j)+b(i+1,j)
(    33)   |                       &     +b(i,j-1)+b(i,j+1))*0.2d0*c(ix,iy)+ap
(    34)   |                              enddo
(    35)   +----------            enddo
          HOME: a(:,i)
(    36) <P>----------            do i=1,n
(    37)   |                            a(1,i) = a(2,i)
(    38)   |                            a(n,i) = a(n-1,i)
(    39)   +----------            enddo
(    40)                          call bound(a)
(    41)                  !HPF$ INDEPENDENT, NEW(i,ix), REDUCTION(ap)
          COMM: RED [ap] [LINO: 42 in sample.F]
          HOME: b(:,j)
(    42) <P>----------            do j=1,n
(    43)   |<S>----------              do i=1,n
(    44)   |                              ix = idxx(i)
(    45)   |                              b(ix,j)=a(i,j)*c(i,j)
(    46)   |                              ap = ap + a(i,j)
(    47)   |                            enddo
(    48)   +----------            enddo
(    49)                            enddo
(    50)
(    51)                            write(*,*)ap
(    52)                            end
```

図 125　HPF 指示文挿入後の並列化情報リスト

次に，並列化情報リストではチェックできない手続呼出し時のデータ転送をチェックします。主プログラムで 3 回引用されている手続 bound には，実引数として，配列 b，c，および a が渡されています。配列 a および b は，HPF コンパイラオプション-Mautodist により，最終次元である 2 次元目で分散されますが, 配列 c は, 分散されないよう DISTRIBUTE 指示文を追加したので, このままでは, 手続 bound の呼出しのどれかで, データ転送が発生してしまいます。このような手続呼出し時のデータ転送を削減するため, 実引数と対応する仮引数が全て同じデータマッピングをもつよう, 図 126 のように，手続のコピーを作成します(手続クローニング)。

```
      subroutine bound(dummy)
      use param
      double precision dummy(n,n)！ オプション–Mautodist により最終次元を分散する
      do i=1,n
        dummy(i,1) = dummy(i,2)
        dummy(i,n) = dummy(i,n-1)
      enddo
      end

      subroutine bound_nodist(dummy)
      use param
      double precision dummy(n,n)
!HPF$ DISTRIBUTE (*,*) :: dummy     ！ 分散しない
      do i=1,n
        dummy(i,1) = dummy(i,2)
        dummy(i,n) = dummy(i,n-1)
      enddo
      end
```

図 126　手続のコピーの作成(手続クローニング)

そして，分散しない実引数による手続 bound の引用(24 行目)を，コピーした手続 bound_nodist の引用に置き換えます。

```
call bound(c)
```

↓

```
call bound_nobound(c)
```

ここで，HPF コンパイラオプション-Mautodist および –Mlist2 を指定してコンパイルすると，サブルーチン bound および bound_nodist に対する並列化情報リスト sample.lst は，それぞれ図 127 および 図 128 のようになります。

```
(    54)                          subroutine bound(dummy)
(    55)                          use param
(    56)                          double precision dummy(n,n)
          COMM: SFT [dummy] [LINO: 57 in sample.F]
          COMM: SFT [dummy] [LINO: 57 in sample.F]
(    57) <N>----------            do i=1,n
(    58)   |                          dummy(i,1) = dummy(i,2)
(    59)   |                          dummy(i,n) = dummy(i,n-1)
(    60)   +----------           enddo
(    61)                          end
```

図 127　サブルーチン bound の並列化情報リスト

```
(    62)                          subroutine bound_nodist(dummy)
(    63)                          use param
(    64)                          double precision dummy(n,n)
(    65)                        !HPF$ DISTRIBUTE (*,*) :: dummy
(    66) <N>----------            do i=1,n
(    67)   |                          dummy(i,1) = dummy(i,2)
(    68)   |                          dummy(i,n) = dummy(i,n-1)
(    69)   +----------            enddo
(    70)                          end
```

図 128　サブルーチン bound_nodist の並列化情報リスト

生成されているデータ転送は，手続 bound の 57 行目に対する，効率のよいシフト転送だけなので，このままでも悪くはありません。しかし，57 行目のループは，HPF コンパイラオプション-Mautodist によって，2 次元目で BLOCK 分散された配列 dummy に対して，2 次元目で境界処理を行っているので，両端の要素が配置されている両端の抽象プロセッサだけが実行するよう図 129 のように ON-HOME-LOCAL 指示文を指定すれば，データ転送なしで実行できます。

```
        subroutine bound(dummy)
        use param
        double precision dummy(n,n)！オプション–Mautodist により最終次元を分散する
        do i=1,n
!HPF$   ON HOME(dummy(:,1)), LOCAL
          dummy(i,1) = dummy(i,2)
!HPF$   ON HOME(dummy(:,n)), LOCAL
          dummy(i,n) = dummy(i,n-1)
        enddo
        end
```

図 129　境界処理への ON-HOME-LOCAL 指示文の指定

最後に，HPF コンパイラオプション-Mautodist および –Mhpfout を指定してコンパイルすると，完成した HPF プログラム sample.hpf.src が生成されます。

5.3　デバッグ

本節では，HPF プログラムでよく出現するバグとその検出および修正方法を説明します。

HPF プログラムは，Fortran コンパイラによってコンパイルすると，逐次 Fortran プログラムとして実行できます。 並列実行に先立って，逐次 Fortran プログラムとして正しく書かれていることを確認してください。

Fortran プログラムとして正常に動作しても， HPF では動作しない場合，以下のような原因が考えられます。

5.3.1　実引数と仮引数の不一致

HPF においては，手続呼出し時の実引数と仮引数の形状や型などは原則として一致している必要があるので， 既存の FORTRAN77 プログラムで良く見られる以下のような記述は，原則として許されません。

- 配列要素実引数と配列仮引数

```
real a(100,100),b(100,100)
do i=1,100
   call sub(a(1,i),b(1,i))   ! 配列要素実引数
enddo
end

subroutine sub(a,b)
real a(100),b(:)              ! 仮配列引数
```

図 130　配列要素実引数と配列仮引数

図 130 のような記述が検出された場合，実行時に次のエラーメッセージが発行され 異常終了します。

- 仮引数が形状引継ぎ配列でない場合

  ″a″: Nonsequential dummy array is associated with array element or scalar actual. PROG=sub ELN=8

- 仮引数が形状引継ぎ配列の場合

"b": Assumed-shape dummy array is associated with array element or scalar actual. PROG=sub ELN=8

配列の一部分を引数とする場合，図 131 のように 部分配列を使用してください。

```
real(10) a(100,100),b(100,100)
do i=1,100
   call sub(a(:,i),b(:,i)) ! 部分配列実引数
enddo
end

subroutine sub(a,b)
real a(100),b(:)
```

図 131　部分配列実引数

- 実引数と仮引数の形状不一致

```
real a(10000),b(10000)
n = 100
call sub(a,b,n)
end

subroutine sub(a,b,n)
real a(n,n),b(n)
```

図 132　実引数と仮引数の形状不一致

実引数の形状と対応する仮引数の形状は，同一でなければなりません。

図 132 のような記述が検出された場合，実行時に次のエラーメッセージが発行され 異常終了します。

- 実引数と仮引数の次元数が異なる場合

  "a": Dummy argument rank differs from actual. PROG=sub ELN=7

- ある次元の寸法が異なる場合

  "b": Dummy array shape differs from actual in dim 1. PROG=sub ELN=7

実行時に配列の大きさを決めたい場合，図 133 のように割付け配列を使用してください。

```
real, allocatable :: a(:,:) ! 割付け配列

n = 100
allocate(a(n,n))
```

図 133　割付け配列の使用

1 つの手続内だけで使用される配列は，図 134 のように 自動割付け配列を使用すると良いでしょう。

```
      subroutine sub(n)
      real :: a(n,n)      ! 自動割付け配列
!HPF$ DISTRIBUTE (*,BLOCK) :: a
```

図 134　自動割付け配列の使用

副プログラムの最初の呼出し時に大きさを決定し，以後の呼出しでは，その領域をそのまま使いたい場合，図 135 のように，SAVE 属性付きの割付け配列を宣言して，最初の呼出し時だけ割り付けるようにしてください。。

```
      subroutine sub(n)
      integer :: iflag = 0

      real, save, allocatable :: a(:,:) ! SAVE 属性付き割付け配列

!HPF$ DISTRIBUTE a(*,BLOCK)
      if(iflag.eq.0)then
        allocate(a(n,n))
        iflag = 1
      endif
```

図 135　最初の呼出し時の割付け

5.3.2　共通ブロック変数の不一致

HPFにおいては，共通ブロック中の変数の個数，各変数の型，形状，およびデータマッピングは，原則として手続間で一致している必要があります。以下のような記述は許されていません。

- 共通ブロック中の変数の個数が異なる

```
      subroutine sub1()
      common /com/a(100,100),b(100,100)
!HPF$ DISTRIBUTE (*, BLOCK) :: a,b
          :
      end

      subroutine sub2()
      common /com/a(100,100)　！ 配列 b の宣言がない。
!HPF$ DISTRIBUTE (*,BLOCK) :: a
```

図 136　共通ブロック変数の個数が異なる共通ブロック

- 共通ブロック変数のデータマッピングの不一致

```
      subroutine sub1()
      common /com/a(100,100)
!HPF$ DISTRIBUTE (*,BLOCK) :: a
          :
      end

      subroutine sub2()
      common /com/a(100,100)　　！ マッピング指定漏れ
```

図 137　共通ブロック変数のデータマッピングの不一致

HPFコンパイラオプション-Mcommonchkを指定してコンパイルすると，実行時に共通ブロック変数の宣言をチェックし，手続間での不一致を検出した場合，以下のようなエラーメッセージを発行して異常終了します。

- 共通ブロック中の変数の個数が異なる場合

  Inconsistency detected in the number of components of common block between sub1 and sub2 : /com/ PROG=sub2

- 共通ブロック変数のデータマッピングが異なる場合

```
Inconsistency detected in the number of explicitly mapped arrays of
common block between sub1 and sub2 : /com/ PROG=sub2
```

本オプションは，1 つの実行ファイルを構成する全ての手続に対して指定しなければならないことに注意してください。

また，本オプションおよびHPFコンパイラオプション-Mnoentryまたは-Mnoerrlineを同時に使用することはできません。同時に指定された場合，後に指定されたオプションの機能だけが有効となります。

5.3.3　配列の宣言範囲外参照

HPF プログラムでは，図 138 のように， 配列の宣言範囲外を参照することはできません。

```
        program main
        real a(100,100)
!HPF$ DISTRIBUTE a(BLOCK, *)
        do i=1,10000
          a(i,1) = i
        enddo
```

図 138　配列の宣言範囲外参照

HPF コンパイラオプション-Msubchk を指定してコンパイルすると，実行時に宣言範囲外参照の有無をチェックし，検出した場合，次のようなエラーメッセージを発行します。

"a" is accessed out of declared bounds along 1st dim. PROG=main ELN=5

宣言範囲外参照検出用のコードは，ベクトル化や並列化を可能な限り阻害しないように挿入されますが，それでもなお性能が低下する場合があることに注意してください。

また，本オプションおよび HPF コンパイラオプション-Mnoentry または-Mnoerrline を同時に使用することはできません。同時に指定された場合，後に指定されたオプションの 機能だけが有効となります

5.3.4　INDEPENDENT 指示文の誤指定

図 139 の do ループは，前の繰返しで確定された変数 l の値が引用されているので，ループの繰返し間での依存があり，並列性することはできません。

並列化できないループに INDEPENDENT 指示文を指定すると，結果不正の原因となります。

```
      l=0
!HPF$ INDEPENDENT,NEW(I,J)　！ 誤り
      do i=1,n
        do j=1,n
          l = l+1
          a(j,i) = l
        enddo
      enddo
```

図 139　並列化できないループへの INDEPENDENT 指示文の指定

HPFコンパイラオプション-Mnoindependentを指定すると，プログラマが指定したINDEPENDENT指示文を全て無視して，HPFコンパイラの自動並列化だけで並列化を行います。これにより結果が正常になった場合，INDEPENDENT指示文の指定に誤りがある可能性があります。

並列化情報リスト等を参照して，自動的には並列化可能と判定されなかったdoループをチェックすると，INDEPENDENT指示文の誤指定の発見に役立ちます。

付録 A　HPF 指示文 および HPF 指示構文

A.1　宣言部に指定する指示文

A.1.1 DISTRIBUTE指示文

プロセッサ構成を指定する場合

!HPF$ DISTRIBUTE *a* **(** <分散形式>,… **) ONTO** *p*

または

!HPF$ DISTRIBUTE (<分散形式>,… **) ONTO** *p* **::** *a*,…

- *a* は、配列またはテンプレート
- *p* は、プロセッサ構成の名前
- <分散形式>は、*****, **BLOCK**[(<式>)], **GEN_BLOCK(***map***)**, または **CYCLIC**[(<式>)]
  - *****は、その次元を分散しないことを指定します。
  - **BLOCK** は、その次元を均等に分散することを指定します。(<式>)により、分散幅を指定できます。分散幅を指定しない場合、分散幅は、次のように計算されます。
    (配列またはテンプレートの対応次元の寸法-1)/(プロセッサ構成の対応次元の寸法)
  - **GEN_BLOCK** は、その次元を不均等に分散することを示します。(*map*)により、プロセッサ配列の対応次元の各要素に分散する配列要素の個数を指定します。1 次元配列 *map* には、プロセッサ構成の対応次元の各要素に分散する配列要素の個数をあらかじめ代入しておきます。
  - **CYCLIC** は、その次元をラウンドロビン方式で巡回的に抽象プロセッサに分散することを指定します。(<式>)により、分散幅を指定できます。分散幅を指定しない場合、分散幅は、1 です。

プロセッサ構成を指定しない場合

!HPF$ DISTRIBUTE *a* **(** <分散形式>,… **)**

または

!HPF$ DISTRIBUTE (<分散形式>,… **) ::** *a*,…

A.1.2 TEMPLATE指示文

!HPF$ TEMPLATE *t* **(** <> ,… **)**

または

!HPF$ TEMPLATE (<>,… **) ::** *t*,…

- *t* は、テンプレート
- <> は，テンプレートの各次元の上下限

A.1.3 PROCESSORS指示文

!HPF$ PROCESSORS *p* **(** <>,… **)**
または
!HPF$ PROCESSORS (<>,… **) ::** *p*,…

- *p* は、プロセッサ構成の名前
- <>は、プロセッサ構成の各次元の上下限。例えば、次の場合、
  !HPF$ PROCESSORS p(n1,n2)
  プロセッサ配列 p の大きさ n1*n2 が抽象プロセッサの個数であり、プロセッサ配列の次元数である 2 は、配列を分散する次元の個数と一致します。

A.1.4 ALIGN指示文

!HPF$ ALIGN *a* **(** <*i*>,… **) WITH** *t***(** <*f(i)*>,… **)**
または
!HPF$ ALIGN (<*i*>,… **) WITH** *t***(** <*f(i)*>,… **) ::** *a*,…

- *a* は、配列
- *t* は、配列またはテンプレート
- <*i*> は，整数型スカラ変数または ***** 。***** を指定した次元は整列されない。
- **<***f(i)***>** は，<*i*> の 1 次式 *s******<*i*>**+***o*，または ***** 。ここで，*s* と *o* は整数型の式。
  - <*f(i)*> が <*i*> の 1 次式 *s******<*i*>**+***o* の場合，配列 *a* の要素 <*i*>は，整列先 *t* の要素 *s******<*i*>**+***o* に整列する。
  - <*f(i)*> が ***** の場合, 配列 *a* 全体は, 整列先 *t* の ***** が指定された次元に対応するプロセッサ配列の次元にそって複製される。(整列先 *t* の ***** が指定された次元の，任意の要素が配置されるすべての抽象プロセッサ上に配置される。)

A.1.5 SHADOW指示文

!HPF$ SHADOW *a***(** <シャドウ幅>,… **)**
　　または
!HPF$ SHADOW (<シャドウ幅>,… **) ::** *a*,…

- *a* は，配列の名前
- <シャドウ幅>は，*n* または *l***:** *u* で，各次元の下方向，上方向のシャドウ幅(定数値)を宣言する。*n* は，*n***:** *n* と同じ意味。

A.1.6 SEQUENCE指示文

!HPF$ [NO] SEQUENCE　[[::] *s*,…]

- *s* は、配列名 または /共通ブロック名/。SEQUENCE 指示文において，*s*,… を省略した場合，全ての共通ブロック および 明示的に分割配置されていない全ての変数を指定したことになる。NOSEQUENCE 指示文において，*s*,…を省略した場合，全ての共通ブロック および 全ての変数を指定したことになる。

A.2　実行部に指定する指示文

A.2.1 INDEPENDENT指示文

完全に並列化可能なループの場合

!HPF$ INDEPENDENT [, **NEW(** *v*,… **)**]

- v は，変数名

集計計算を行う並列化可能なループの場合

!HPF$ INDEPENDENT [, **NEW(** *v*,… **)**], <REDUCTION 節>,…

- v は，変数名
- <REDUCTION 節>は,

  REDUCTION([<集計種別 1> :] *r*,… **)**

  または

  REDUCTION([<集計種別 2> :] *r* **/***p*,…**/**,… **)**
  - <集計種別 1>は，**+, *, .AND.**, **.OR.**, **.EQV.**, **.NEQV.**, **MAX**, **MIN**, **IAND**, **IOR**, または **IEOR**
  - *r* は，集計変数の名前
  - <集計種別 2>は，**FIRSTMAX**, **FIRSTMIN**, **LASTMAX**, または **LASTMIN**
  - *p* は，位置変数の名前
  - <集計種別 1>： を省略した場合，許される集計演算の形式は，

    r **=** *r* <op> <expr>　または　*r* **=** <expr> <op> *r*

    もしくは

    r **=** <f(*r*, <expr>)>　または　*r* **=** <f(<expr>, *r*)>
    - *r* は，集計変数の名前
    - <op>は，集計演算子*****, /, **+**, -, **.AND.**, **.OR.**, **.EQV.**, または **.NEQV.**
    - <expr>は，集計変数を含まず，<op>よりも先に計算される式
    - <f()>は，集計関数 **MAX**, **MIN**, **IAND**, **IOR**, または **IEOR**

A.2.2 ON-HOME-LOCAL指示構文 および 指示文

ON-HOME-LOCAL 指示構文

!HPF$ ON HOME(<部分配列> **) [, LOCAL[(** *v,…* **)]] BEGIN**

<実行文 または 実行構文の列>

!HPF$ END ON

- <部分配列>が配置されている抽象プロセッサが，<実行文 または 実行構文の列>を実行する
- v は，データ転送が不要な変数の名前。LOCAL だけを指定した場合，<実行文 または 実行構文の列>でアクセスされる全ての変数に対するデータ転送が不要であることを指定する

ON-HOME-LOCAL 指示文

!HPF$ ON HOME(<部分配列> **) [, LOCAL[(** *v,…* **)]]**

- <部分配列>が配置されている抽象プロセッサが，直後の実行文 または 実行構文を実行する。

A.2.3 REFLECT指示文

!HPF$ REFLECT [(<シャドウ幅>,…)] [::] *a,…*

- *a* は，配列の名前。宣言部で SHADOW 指示文を指定する必要がある。
- <シャドウ幅>,… を指定すると，シャドウ領域のうち，指定された幅だけにシフト転送を行う。これを部分 REFLECT 指示文と呼びます。

A.3　その他の構文

A.3.1 EXTRINSC接頭辞

EXTRINSIC（<言語>，<モデル>）

　または

EXTRINSIC（<外来種別キーワード>）

- <言語>は，**”HPF”** または **“Fortran”**
- <モデル>は，**”GLOBAL”**，**“LOCAL”**，または **”SERIAL”**。**”GLOBAL”**，**“LOCAL”**，および **”SERIAL”**は，それぞれグローバルモデル，ローカルモデル，および シリアルモデルを示します。
- <外来種別キーワード>は，**HPF**，**HPF_LOCAL**，**HPF_SERIAL**，**Fortran_LOCAL**，または **Fortran_SERIAL**。それぞれ，グローバルモデルの HPF，ローカルモデルの HPF，シリアルモデルの HPF，ローカルモデルの Fortran，シリアルモデルの Fortran を意味します。

付録 B　よく尋ねられる質問

B.1　データマッピング

- DISTRIBUTE 指示文も ALIGN 指示文も指定していないデータは，どのように分割配置されますか。
  - 全ての抽象プロセッサ上に複製されます。

B.2　データ転送

- 割付け配列または形状引継ぎ配列に対して，不要なデータ転送が発生します。
  - 割付け配列および形状引継ぎ配列のデータマッピングは，ALIGN 指示文を使用してください。データマッピングの指定方法については，4.1.4 項を参照してください。

B.3　実行性能とメモリ使用量

- 実行時のメモリ使用量が多すぎます。
  - 次のような原因が考えられます。
    - ✧ HPF コンパイラは，高速なシフト転送を行うため， BLOCK 分散 または GEN_BLOCK 分散により分割配置された次元の両端に， 既定値では，それぞれ幅 4 のシャドウ領域を確保します。シフト転送が必要ない場合， 各配列の SHADOW 幅を 0 に設定すると，メモリ使用量を削減できます。 SHADOW 幅を 0 に設定するには， 各配列に対して SHADOW 指示文を指定して，SHADOW 幅を明示的に 0 と指定するか，HPF コンパイラオプション-Moverlap=size:*n* を指定して，次のようにコンパイルしてください。

```
%> ve-hpf –Moverlap=size:0 source.hpf
```

    - ✧ DATA 文などで，大きな配列を初期化している場合，配列全体分のメモリ領域が，各抽象プロセッサ上に確保されます。 実行時の最初に初期化をすれば，メモリ使用量が削減できます。

    - ✧ ループの実行や手続呼出し時にデータ転送が必要となる場合，データ転送対象の配列全体分のメモリ領域が，各抽象プロセッサ上に確保される場合があります。 以下のような方法で，並列化可能なループを増やしたり，データ転送を削減したりすることにより，

使用メモリ量を削減できます。なお，データ転送の発生箇所は，並列化情報リストまたは診断メッセージから抽出することができます。

- INDEPENDENT 指示文(+REDUCTION 節)を指定して，並列化可能なループを増やす。
- ON-HOME-LOCAL 指示文を指定して，無駄なデータ転送を減らす。
- 配列のデータマッピングやループの記述方法を修正して，無駄なデータ転送を減らす。
- 実引数と仮引数のデータマッピングを可能な限り同一にする。

➢ DISTRIBUTE 指示文も ALIGN 指示文も指定されていないデータは，全ての抽象プロセッサ上に複製されます。可能であれば，大きな配列は分割配置するようにしてください。

➢ 分散並列化と自動共有並列化を併用するハイブリッド並列化の場合，局所変数は各スレッド上にそれぞれ割り付けられます。そのため大きな局所配列があると，共有並列処理用のメモリ使用量が大きくなります。大域変数のメモリ領域は，既定値では全スレッドで共有されるので，局所配列を大域配列に修正すると，使用メモリ量を削減できます。

- Fotran コンパイラオプション-mparallel を指定してコンパイルすると，大幅に性能が低下します。
  ➢ Fortran コンパイラオプション-mparallel を指定すると，HPF による分散並列化および Fortran コンパイラによる自動共有並列化の 2 レベルの並列化が行われます。
  その結果，プログラムの並列度は，最大で，HPF の抽象プロセッサ数と共有並列スレッド数の積となります。 VE のコア数がこの最大並列度よりも少ない場合， 競合のため，実行性能が著しく低下します。HPF プログラム実行時の実行プロセッサ数の指定 および 実行時の環境変数 OMP_NUM_THREADS または VE_OMP_NUM_THREADS による共有並列スレッド数の指定により，最大並列度が VE のコア数を越えないようにしてください。

- 並列化情報リストや診断メッセージを見る限り，主要ループは並列化されているし重いデータ転送もでていないのに何故か性能がでません。
  ➢ 次のような原因が考えられます。
  ✧ 実引数と仮引数のデータマッピング(DISTRIBUTE 指示文, ALIGN 指示文，および SHADOW 指示文の指定)が異なる場合，手続の呼出し時と戻り時に， 実引数と仮引数間でデータ転送を伴うコピーが発生します。
  これはコンパイル時には検出できませんので，実行時オプション-hpf-commmsg を指定して実行し，手続呼出し時・戻り時にデータ転送が発生していないかチェックしてください。

```
%> mpirun –np 4 ./a.out –hpf -commmsg
```

次の警告メッセージが出力された場合，

"仮引数名": Communication occurs at procedure boundary PROG="手続名" ELN="行番号"

実引数と仮引数との間でデータ転送を伴うコピーが発生しているので，警告メッセージ中の手続名や仮引数名を参考にして，仮引数と対応する実引数のデータマッピングが同一になるよう修正してください。

✧ HPF コンパイラにより分散並列化されたループは，ループ長が抽象プロセッサ数分の 1 になるだけでなく，ループの始値と終値が変数に置き換えられるので，逐次実行時とはベクトル化対象のループが変わってしまい，ベクトル長が短くなって性能が出ないことがあります。FTRACE 情報を参照して，想定よりベクトル長が短い場合，並列化されループ長が短くなったループがベクトル化されていないかチェックしてください。そして，必要であれば，Fotran コンパイラ指示行 NOVECTOR などを指定して，ベクトル化対象ループを変更してください。

✧ HPF コンパイラでコンパイルする場合，何度も呼び出される手続が，インライン展開できなくなって性能が低下する場合があります。

- インライン展開された手続が配列引数を持たない場合，HPF コンパイラオプション -Mnoentry を指定してコンパイルすると，インライン展開が可能になる場合があります。
- インライン展開したい手続が配列引数をもつ場合，手動でインライン展開する必要があります。

B.4　その他

- INDEPENDENT 指示文を指定すると結果が不正となります。
  - ➢ 次のような原因が考えられます。INDEPENDENT 指示文誤指定の見つけ方については，5.3.4 項も参照してください。

    ✧ 並列化できないループに INDEPENDENT 指示文を指定すると，結果が不正となります。例えば，以下の例では，l の値は，直前の繰返しで確定された l の値を引用しているので，INDEPENDENT 指示文を指定することはできません。

```
        l=0
!HPF$   INDEPENDENT,NEW(I,J)  ! 誤り
        do i=1,n
          do j=1,n
            l = l+1
            a(j,i) = l
          enddo
        enddo
```

次のようにプログラムを修正すれば, 並列化が可能となり, INDEPENDENT指示文も指定できます。

```
!HPF$   INDEPENDENT,NEW(I,J)
        do i=1,n
          do j=1,n
            a(j,i) = 1+n*(i-1)*(j-1)
          enddo
        enddo
```

✧　集計計算を行っているループにREDUCTION節なしのINDEPENDENT指示文を指定すると，実行結果が不正となります。以下の例では，do iのループにおいて，配列aに対して配列bの値を足し込んでいく総和の集計計算を行っていますので，INDEPENDENT指示文だけでは結果が不正になります。

```
!HPF$   INDEPENDENT,NEW(i)  ! 誤り
        do j=1,n
          do i=1,n
            a(i) = a(i) + b(i,j)l
          enddo
        enddo
```

次のように, 配列aに対するREDUCTION節を追加するか, INDEPENDENT指示文を削除すれば，結果が正しくなります。

```
!HPF$   INDEPENDENT,NEW(i),REDUCTION(a)
        do j=1,n
          do i=1,n
            a(i) = a(i) + b(i,j)l
          enddo
        enddo
```

付録 C　発行履歴

C.1　発行履歴一覧表

2020 年	9 月	初版
2022 年	7 月	第 2 版

C.2　追加・変更点詳細

初版

第 2 版

NEC Fortranコンパイラオプション（2.3.2）

NEC Fortranコンパイラ実行時環境変数（3.2.1）

FTRACEリージョン指定機能の使用法（5.1.3）

入出力の高速化について（5.1.4）

索　引

R

S

T

V

あ

い

お

か

き

く

け

さ

し

せ

そ

ち

て

は

ふ

へ

ほ

ま

め

る

ろ

わ

SX-Aurora TSUBASA システムソフトウェア

SX-Aurora TSUBASA
NEC HPF ユーザーズガイド

2022年　7月 第2版

日本電気株式会社

東京都港区芝五丁目７番１号

TEL(03)3454-1111（大代表）