XcalableMP

仕様書

Version 0.5J

超並列プログラミング言語検討委員会

2009 年 8 月

Contents:

1 はじめに

この文書は、分散メモリシステムを対象に指示文ベースで並列プログラミングを行うためのプログラミングモデル xcalableMP について、記述するものである。

1.1 目的と適用範囲

1.2 設計方針

XcalableMP の設計方針は、以下とおり。

① 基本的に分散メモリ並列システムを対象とする。
② 既存のプログラムからの移行、あるいは教育のコストを考えて、指示文によりベース言語（C 言語と Fortran 言語）を指示文により拡張する。極力、ベース言語の仕様は変更しない。
③ グローバルビューにおいては、データ並列プログラミングモデルとワークシェアによって、典型的な並列化をサポートし、OpenMP のように既存のコードをなるべく変更せずに並列化を可能とする。
④ 個別のノードを意識してプログラミングするローカルビューとして、CAF(Coarray Fortran)の元にした CAF-like PGAS (Partitioned Global Address Space)機能を提供する。これにより、メッセージ通信とリモートメモリ操作を用いることができる。
⑤ 通信や同期は明示的に行う。これにより、すべての通信・同期操作は指示文で指定されたときに行われるために、性能チューニングの工夫がわかりやすくできるようになる。
⑥ 柔軟性と拡張性のために、基本的な実行モデルは MPI が呼び出せるように設定し、これにより、より複雑で性能チューニングされた MPI ライブラリを利用可能にする。
⑦ マルチコアあるいは SMP クラスタでは、OpenMP とのハイブリッドプログラミングができるようにする。ノード内部では、OpenMP を用いてスレッドプログラミングすることができるようにする。

Features of XcalableMP are summarized as follows:

- **Language extensions** for familiar languages C and Fortran, which can reduce code-rewriting and educational costs.
- XcalableMP supports typical parallelization based on the **data parallel paradigm** and work sharing under "**global view**", and enables parallelizing the original sequential code using minimal modification with simple **directives**, like OpenMP.

- XcalableMP also includes CAF-like PGAS (Partitioned Global Address Space) feature as "**local view**" programming.
- **Explicit communication and synchronization**. All actions are taken by directives for being “easy-to-understand” in performance tuning.
- For flexibility and extensibility, the execution model allows **combining with explicit MPI coding** for more complicated and tuned parallel codes and libraries.
- For multi-core and SMP clusters, **OpenMP directives can be combined** into XcalableMP for thread programming inside each node as a hybrid programming model.

XcalableMP is being designed based on the experiences of HPF, Fujitsu XPF (VPP Fortran) and OpenMPD.

2 言語と実行モデルの概要

2.1 ハードウエアモデル と基本実行モデル

XcalableMP の並列プログラムが実行される対象は、分散メモリ型のプラットフォーム（Fig-1）とする。すなわち、ひとつのコア、もしくはメモリを共有する複数のコアからなる計算機（ノードと呼ぶ）が、ネットワークにより結合されているプラットフォームを対象とする。各ノードは直接参照・更新できる固有のメモリ空間を持ち、他のノードのメモリ空間はコストを伴う通信を介して参照・更新することができる。

基本的な実行モデルは、分散メモリ型 SPMD(Single Program Multiple Data)である。プログラムは各ノードで同じプログラムがメインプログラムから同時に実行が開始される。指示文により、各ノードのプログラムは通信・同期を行う。基本的に、指示文がない限り通信・同期が行われない（通信・同期の明示）。この場合、プログラムは各ノードにおいて、重複実行されることになる。

各ノードで実行されるプログラムはプログラムの実行開始時は、1つのスレッドのみとする。ノードの中では必要に応じて、OpenMP などを用いることに複数スレッドを使うハイブリッドのプログラミングをすることが可能である。これについては、???を参照のこと。

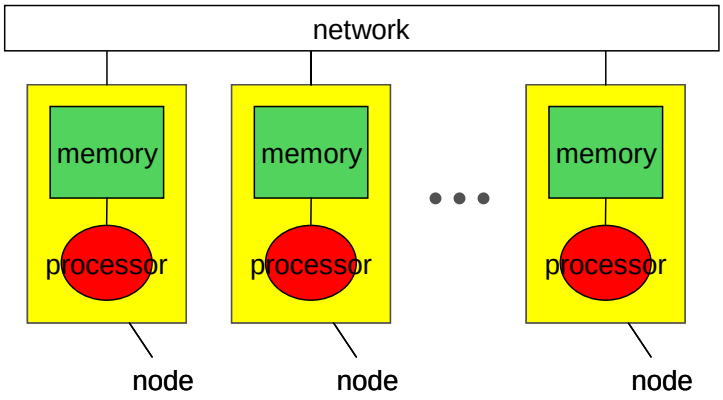

Fig- 1 Hardware Model

通常のプログラム内で宣言されたデータは、それぞれのノード内のメモリに配置され、ノードで実行されるプログラムからアクセスされる。XcalableMP では、ノード番号を指定することによって、他のノードに配置されたデータをアクセスすることができる。これによる、プログラミングをローカルビューによるプログラムという。

これに対して、XcalableMP では、ノード間に跨る配列データを宣言し、これらの処理を各ノードで分担させて行うことができる。これをグローバルビューによるプログラミングと呼ぶ。

2.1.1 グローバルビューのプログラミング

グローバルビューは、データ並列プログラミングモデルによって、典型的な並列化をサポートし、OpenMP のように既存のコードをなるべく変更せずに並列化を可能とする機能である。

プログラマは、まず共有されるデータについてデータの分散を指定し、データを各ノードに分散配置する。loop 構文を用いることによって、ループなどの実行をデータが割り当てられているノードに分割して実行する。配列の隣接のデータなどの通信を行うなど、必要なデータの一貫性を保持して計算を進めることができる。プログラマは適当な指示文を用いて、必要なデータはノードにあるように通信を行わなくてはならない。

グローバルビューは、逐次プログラムのイメージから出発して、データをノードに分散させ、それに応じた計算の並列化を考えていくプログラミングスタイルに適している（Fig-3）。グローバルビューではノード数に関わらず同じ結果を出すプログラムを作成することができる。

グローバルビューのプログラミングのために、以下の3つを明示的に記述する手段が提供される。これらの大部分は指示行の形式をもっているので、ベース言語（C または Fortran）のコンパイラで翻訳した場合にはそのまま逐次プログラムとして同じ意味の実行を行えることが多い。

- データマッピング… データの分散を指定する。
- 計算の並列化 … データの分散に従って、ループを分割する。ただし、HPF では処理系に

与えるヒントであるのに対して、XcalableMP では処理系の振る舞いを強制的に指示する。

- 通信･同期 … グローバルビューでのノード間の通信を明示するための書式をもつ。XcalableMP では、ノード間の通信はすべて明示的に指示される。処理系は、明示的に通信を指示されていない実行文については、通信が発生しないことを保障する。

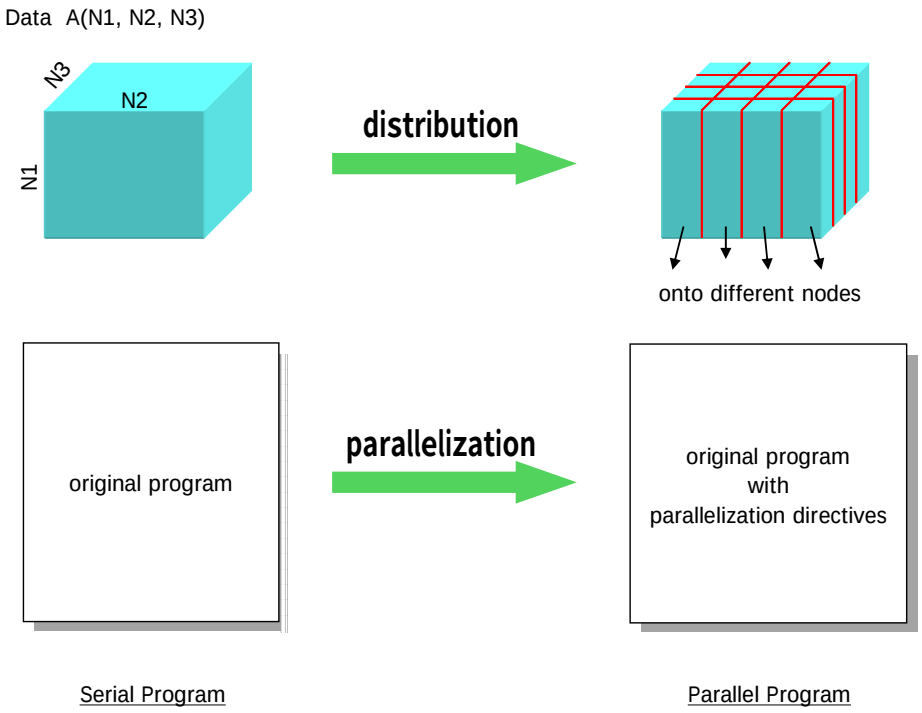

Fig- 2 グローバルビューを使ったプログラムの並列化

2.1.2 ローカルビューによるプログラミング

ローカルビューは、各ノードの振る舞いを記述するプログラミングスタイルに適している（Fig-4）。MPI プログラムはローカルビューであるため、XcalableMP のローカルビューとの interoperability は高い。

ローカルビューのプログラミングのために、ベース言語からの拡張仕様と、指示文の形式の仕様が提供される。拡張仕様の代表的なものは coarray 記法である。例えば、ノード n に配置された配列要素 A(i,j,k)を参照するには、Fortran では A(i,j,k)[n]と表記する。そのデータの引用が必要な文脈ではそのデータを他ノードから受け取る通信が発生し、そのデータの更新が必要な文脈では他ノードのそのデータへの書き込みを起こす通信が発生する。

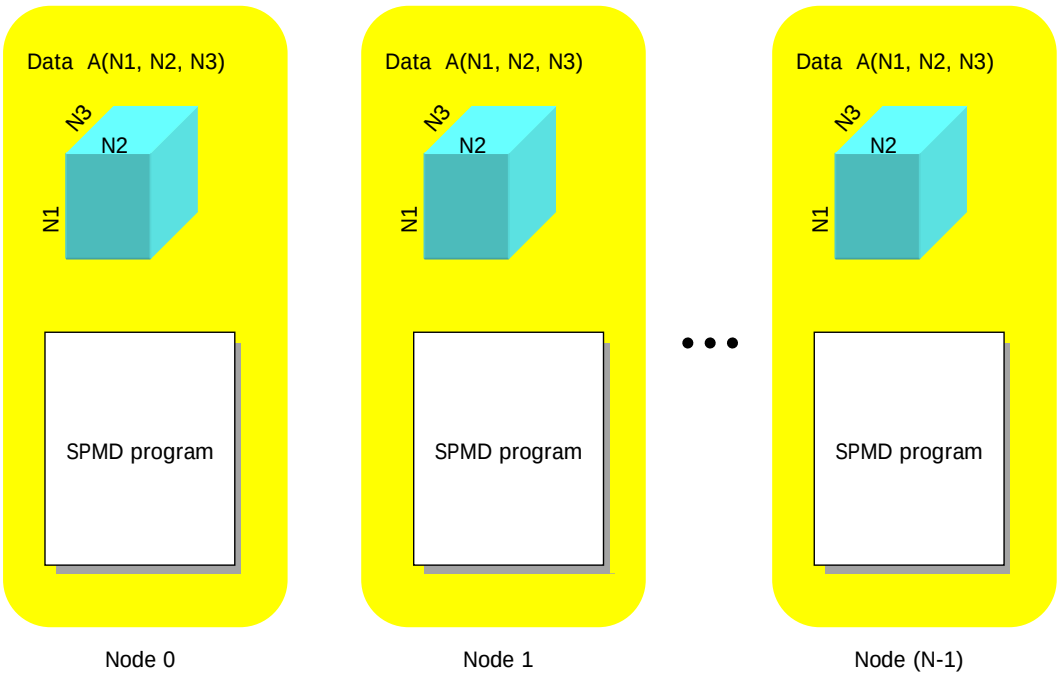

Fig- 3 ローカルビューのプログラミング

2.1.3 共通概念とビューの連携

ノードは、グローバルビューではデータの分散先や計算負荷の分散先として使用され、ローカルビューでは coarray 記法の中に現れてデータを指定するために使用される。

グローバルビューとローカルビューについては、アプリケーションの性質によって使い分けるのがよい。Fig-2 に示す例は、グローバルビューの変数 G とローカルビューの変数 L の割付けのイメージを示している。

ローカルビューの変数は、宣言された形状のデータすべてのノードに割り付けられるので、データを参照するには、変数名と各次元の添え字に加えて、ノード番号を指定しなければならない。ノード番号を省略した場合には、自ノードのデータであると見なされる。

グローバルビューの変数は、データは仮想的な共有空間に置かれているかのように見えるので、原則として配列要素の特定にノード番号を必要としない。ただし、グローバルビューであっても、XcalableMP では他ノードのデータの参照はすべて明示的な記述を必要とするので、そのデータが自ノードに配置されているか否かは利用者が意識しなければならない。

XcalableMP では、グローバルビューで記述されたデータをローカルデータとして、アクセスする指示文を提供する。即ち、一つのデータに対してグローバルビューとローカルビューの2つの名前を付けて、それらを使い分けることによって、同じデータを両方のビューで使用することができる（**エラー! 参照元が見つかりません。**節）。この機能により、グローバルビューとローカルビューが自

然に混在したプログラムを記述することができる。

```
      dimension L(50, 40)                                    ! local view (default)
      dimension G(80, 100)                                   ! global view
!$xmp template, distribute (block) onto (0:4) ::  T(100)
!$xmp align with T :: G
```

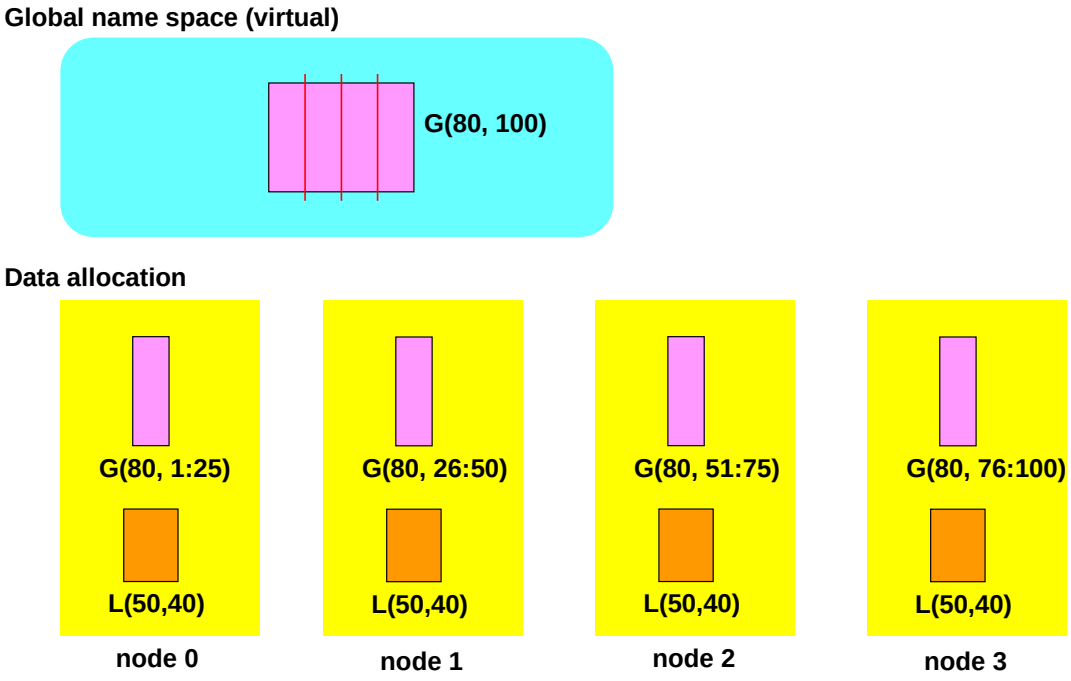

Fig- 4 グローバルビューとローカルビューのデータの例

2.2 タスクと実行モデル

プログラムの実行開始時に決まるノードの集合を、全体ノード(entire nodes)と呼ぶ。全体ノードを複数のタスクに振り分けて、それぞれのタスクで並列実行を行うプログラムを記述することもできる（Fig-5）。全体ノードのサブセットであり、タスクを一緒に実行するノードの集合を、そのタスクの実行ノードと呼ぶ。タスク並列化の指示がない場合には、プログラム全体が1つのタスクであり、ノードの全体が実行ノードである。

タスクの実行は、デフォルトでは冗長実行（SPMD 実行）である。Loop 指示構文または Array 指示構文による指示があれば、その範囲で並列実行が行われる。

タスクは、Task 指示構文によって定義される。Task 指示構文の中で呼び出されたサブプログラム（サブルーチンまたは関数）は、静的にまたは呼び出し時に実行ノードが決定されるタスクとして実行される。

ノードを跨ぐデータの参照･定義には、通信の指示文を用いる。ローカルビューでは coarray 記法によって行うこともできる。MPI ライブラリを直接呼び出して行うこともできる。ノード番号で指定されるノード群を、共有メモリ空間またはタスク間の通信チャネルとみなしてプログラミングすることも可能である（Fig-6）。これらの通信はすべて明示的な指示で行われ、指示のない限り処理系が自動的に生成することはない。

データは特に指定されてない限り、ローカルのノードのメモリに重複して配置される。

2.3 言語仕様の体系

XcalableMP 言語仕様は、Fortran と C をベース言語とし、ベース言語に対して以下の拡張または修正を加えたものである。

- 指示行（directive）
- 最小限の言語拡張
- 問合せ関数（組込み手続）
- 最小限必要な言語の変更と制約

2.4 用語

■ ノード node

XcalableMP の計算機モデルにおける、固有のメモリと CPU（複数のコアがあってもよい）からなる計算を行う。XcalableMP では、各ノードで1つのスレッドから実行が始まる。

■ ノード番号 node number

全体ノードに属するすべてのノードに一意に割り振られる番号
ノード番号は 1 以上で全体ノードの数以下の値となる。
ノード番号と MPI の rank 番号との対応付けはシステムが決定する。
全体ノードにマップした coarray のイメージ index は、ノード番号と一致する。

■ ノード集合 node sets
ノードの集合。

■ 全体ノード集合 entire nodes set
■ 全体ノード
プログラム全体の実行に参加するすべてのノード。特にその全体を意図する場合には全体ノード集合と呼ぶことがある。全体ノードはプログラムの実行開始時に確定する。

■ 実行ノード集合 execution/executing nodes set
■ 実行ノード execution nodes
プログラム中の一定の範囲（プログラム単位、実行ブロック）について、実行を行うノード。プログラム全体を実行する実行ノードは全体ノードである。タスクの実行ノードはそのタスクに指定されたノード群であり、それは静的に決定される場合と、実行時に動的に決定される場合がある。

■ ノード配列 node array
ノードから成る多次元配列。ノード配列は属性として、名前、形状、それぞれの配列要素のノード番号をもつ。異なる名前をもつノード配列が同じノードを含むこともある。ノード配列の要素の並び順は、デフォルトでは処理系が決定する。

■ 実行ノード配列 execution node array
実行ノードから成る仮想的な配列

■ 名前付きノード配列 named node array
Nodes 指示文によって名前を定義されたノード配列

■ タスク task
単一または複数のノードで協調して一連の動作を行うプログラム実行のこと。プログラムの文脈では、単一または複数のノードで実行されるプログラムの範囲を示す。タスクは入れ子の構造をもつことができ、内側のタスクの実行ノードは外側のタスクの実行ノードのサブセットとなる。

■ テンプレート template

インデックスの集合を表現する仮想的な配列。ここでインデックスとは、配列の添え字およびループの反復の総称。テンプレートは属性として、名前、次元数、各次元の下限値と上限値、各次元の分散の属性をもつ。

■ 重複実行 duplicated execution

同じプログラムの部分を異なるノードで実行すること。それぞれのノードの内部状態が同じ場合には、ローカルな実行する時には同じ結果となる。

■ データマッピング Data mapping

データをノードに分散配置すること。

■ ワークマッピング Work mapping

実行する部分をノードに割りあてること。」

■ マップ、マッピング map, mapping

① 分散変数またはテンプレートのノードへの対応付け
Distribute 指示文の onto 節で指示されたノード配列に分散配置される。

② coarray のノードへの対応付け
型宣言文や DIMENSION 文などの on 節で指示されたノード集合（なければその手続の実行ノード集合）に重複配置される。

■ イメージインデックス image index

coarray の複製一つ一つに一意に与えられる 1 以上の番号
全体ノードにマップされた coarray のイメージインデックスは、ノード番号と一致する。

■ ローカル

手続きの完了がそれを実行しているプロセスのみに依存する場合。 このような操作は、別のユーザープロセスとの通信を必要としない。

■ ノンローカル

操作を完了するために、別のプロセスでのなんらかの MPI 手続きの実行が 必要かもしれない場合。このような操作は、別のユーザープロセスとの 通信を必要とするかもしれない。

■ グローバルデータ global data

分散された配列。

■ ローカルデータ local data

グローバルデータ以外のデータ。

■ 集団的 collective

あるプロセスグループ中のすべてのプロセスが、その手続きを起動しなければ ならない場合。

■ 分散 distribution

distribute 指示文によって、テンプレートの各要素を、ノード集合の特定の要素に割り当てること、または、テンプレートの特定の次元の各インデックスを、ノード集合の特定の次元の特定のインデックスに割り当てること。または、そのような割り当ての状態。

広義には、整列先のテンプレートを介して、配列の各要素を、ノード集合の特定の要素に割り当てること、または、配列の特定の次元の各インデックスを、ノード集合の特定の次元の特定のインデックスに割り当てることを指す。

■ 整列 alignment

align 指示文によって、配列の各要素を、テンプレートの特定の要素に対応付けること、または、配列の特定の次元の各インデックスを、テンプレートの特定の次元の特定のインデックスに対応付けること。または、そのような対応付けの状態。

■ シャドウ shadow

分散された配列において、分散境界をはさんで隣接する要素(反映元)の値を一時的に保持するために用いられる領域。

■ ローカルエイリアス

分散された配列に関し、各ノードに割り当てられている要素のみから成る配列、または、**local_alias** 指示文によりそのような配列に付けられる名前。ローカルエイリアスは、通常のローカルデータとして定義および参照できる。

3 指示文

本章では、指示文の説明に以下の表記を用いる。

xxx …	*xxx* または *xxx xxx* または *xxx xxx xxx* または･･･
	…の直前が閉括弧の場合、開括弧から閉括弧までを *xxx* と見なす
xxx , …	*xxx* または *xxx, xxx* または *xxx, xxx, xxx* または･･･
xxx	*xxx* は省略可能
[F]	その行の内容はベース言語が Fortran の場合に限る
[C]	その行の内容はベース言語が C の場合に限る

指示文は、指示行の形式の中で記述される。指示行は、Fortran では !$xmp などで始まる行であり、C では#pragma xmp で始まる行である。Fortran の固定形式／自由形式のルールや継続行のルールは、OpenMP と HPF に準じる。

指示文は宣言指示文と実行指示文に分かれる。実行指示文には、単独指示文（stand-alone directive）と、指示構文（directive construct）を構成する指示文がある。単純指示文には、Barrier 指示行など、その行単独でひとつの機能を表現する。指示構文は以下の形式をもち、関連付けられたベース言語との組合せでひとつの機能を表現する。

[F]

!$xmp *directive-name clause …*
statement
…
!$xmp end *directive-name*

ただし、Fortran の Loop 指示構文の場合、end を省略できる。

!$xmp *directive-name clause …*
do-loop-construct

[C]

#pragma xmp *directive-name clause …*
statement

C の *statement* は 複文 *compound-statement*（中括弧で囲まれた *statement* の並び）であってもよい。

3.1 ノード宣言指示文

3.1.1 Nodes 指示文

[Summary]

Nodes 指示文は、ノード配列に名前を与え、形状と大きさを宣言する。同時に、そのノード配列についてのいくつかの属性を宣言することができる。

[Syntax]

[F]

!$xmp nodes *[* **(** *map-type* **)***]* *nodes-name* **(** *nodes-size [, ...]* **)**

!$xmp nodes *[* **(** *map-type* **)***]* *nodes-name* **(** *nodes-size [, ...]* **) = ***

!$xmp nodes *[* **(** *map-type* **)***]* *nodes-name* **(** *nodes-size [, ...]* **) =** *nodes-ref*

[C]

#pragma xmp nodes *[* **(** *map-type* **)***]* *nodes-name* **(** *nodes-size [, ...]* **)**

#pragma xmp nodes *[* **(** *map-type* **)***]* *nodes-name* **(** *nodes-size [, ...]* **) = ***

#pragma xmp nodes *[* **(** *map-type* **)***]* *nodes-name* **(** *nodes-size [, ...]* **) =** *nodes-ref*

[共通]

nodes-size は、正の整定数式または *

map-type は、**regular**

[Restriction]

- ノード名 *nodes-name* は、class(1)に属す名前であり、他の class(1)の名前と衝突してはならない。
- メインプログラムとモジュールでは、 第2の書式を記述することはできない。
- *nodes-size* は、最後の次元にだけ“*”を記述することができる。
- *nodes-ref* で参照されるノード名は、*nodes-name* または *nodes-name* を直接または間接に参照して定義された名前であってはならない。
- *nodes-size* が“*”を含まないとき、*nodes-size* の総積は、継承元ノードのノード数と一致しなければならない。ここで継承元ノードとは、第1の書式では全体ノード(用語)、第2の書式では呼出し元の実行ノード(用語)、第3の書式では *nodes-ref* である。
- *nodes-ref* に含まれる *nodes-subscript* は、“*”であってはならない。

[Description]

第1の書式では全体ノード、第2の書式では実行ノード、第3の書式では *nodes-ref* で表されるノードについて、指定する次元数および各次元の大きさ *nodes-size* でノード配列(用語)を構成し、

その名前 *nodes-name* を宣言する。

map-type の指定がないとき、ノード配列のノードの並び順は処理系に依存する。処理系はできるだけ計算機のネットワークトポロジに適したマッピングを行うことが望ましい。

map-type に regular が指定されたとき、ノード配列のノードの並び順は、Fortran の配列の並び順に従う。すなわち、第1の書式では、Fortran の配列の並び順に沿ったノード番号（用語）となる。第2および第3の書式では、順序結合（sequence association）に沿って継承元とノード配列のノードが対応付けられる。

[Examples]

メインプログラムでの第1および第3の書式の使用例。ノード配列pの宣言によってノード数 16 が明示されているので、このプログラムは 16 ノードで実行されなければならない。regular の指定がないので、r(1)と p(3)が同じノードである保証はないし、z(1,1)がノード番号 1 である保証はない。

```
      program main

!$xmp nodes p(16)
!$xmp nodes q(4,*)
!$xmp nodes r(8)=p(3:10)
!$xmp nodes z(2,3)=(1:6)
      ...
      end program
```

```
main()
{
#pragma xmp nodes p(16)
#pragma xmp nodes q(4,*)
#pragma xmp nodes r(8)=p(3:10)
#pragma xmp nodes z(2,3)=(1:6)
   ...
}
```

regular オプションの使用例。ノード配列pは regular の宣言がないので、p(1),p(2),…の順にノード番号 1,2,…が対応付く保証はないが、q は regular の宣言があるので q(1,1),q(2,1),q(3,1),q(4,1),q(1,2),…の順にノード番号 1,2,3,4,5,…となる。r は regular 宣言があるので、r(1)と p(3)、r(2)と p(4)、…がそれぞれ同じノードであることが保証される。z は regular 宣言があるので、z(1,1),z(2,1),z(1,2),z(2,2),z(1,3),z(2,3)の順にノード番号 1,2,3,4,5,6 が対応する。

```
      program main

!$xmp nodes p(16)
!$xmp nodes(regular) q(4,*)
!$xmp nodes(regular) r(8)=p(3:10)
!$xmp nodes(regular) z(2,3)=(1:6)
      ...
      end program
```

サブプログラムでの使用例。ノード配列が第2の書式なので、関数 foo を呼び出す実行ノードは 16 ノードでなければならない。q は第1の書式であり、全体ノードの2次元形状を表現する。r は実行ノード配列 p のサブセット、x は全体ノード q のサブセットとなる。

```
      function foo()

!$xmp nodes p(16)=*
!$xmp nodes q(4,*)
!$xmp nodes r(8)=p(3:10)
!$xmp nodes x(2,3)=q(1:2,1:3)
      ...
      end function
```

3.1.2 ノード参照

[Summary]

ノード配列またはノード番号の表記を使って、ノードの並び(配列)を表現する。

[Syntax]

ノード参照 *nodes-ref* には、ノード番号参照 *node-number-ref* と、名前付きノード参照 *named-nodes-ref* がある。

- *nodes-ref*

  node-number-ref / *named-nodes-ref*

- *node-number-ref*

  node-number / **(**[*node-number*] **:** [*node-number*] [**:** *int-expr*] **)**

  node-number は正の整数式

- *named-nodes-ref*

nodes-name [**(** *nodes-subscript* [, ...] **)**]

- *nodes-subscript*

  int-expr / *triplet* / *****

[Description]

ノード番号参照は、1 つのノード番号、またはノード番号の並びを参照する。

名前付きノード参照は、ノード名で指定されるノード配列、またはその部分配列を参照する。添字三つ組（*triplet*）を使った部分配列の表現は、Fortran の部分配列と同様である。

nodes-subscript として “*” をもつ名前付きノード参照は、実行指示行の on 節の中でだけ現れることができ、実行ノードによって異なる実行ノード配列を参照する。すなわち、第 k 添字が “*” である名前付きノード参照

$p(s_1, \ldots, s_{k-1}, *, s_{k+1}, \ldots, s_n)$　　ただし s_i は*以外の *nodes-subscript*

は、ノード

$p(j_1, \ldots, j_{k-1}, j_k, j_{k+1}, \ldots, j_n)$　　ただし j_i は整数

では

$p(s_1, \ldots, s_{k-1}, j_k, s_{k+1}, \ldots, s_n)$

と評価され、第 k 添字が同じ j_k であるノードと共に実行ノード配列を構成する。複数の *nodes-subscript* が “*” である場合も同様である。

[Examples]

- ノード参照は以下のように使用される。
  - Distribute 指示文で、分散先のノード配列として

    例: `!$xmp distribute a(block) onto` **`p`**

  - Nodes 指示文で、ターゲットのノードの並びとして

    例: `!$xmp subnodes r(2,2,4) =` **`p(1:4,1:4)`**

    例: `!$xmp subnodes r(2,2,4) =` **`(1:16)`**

  - Task 指示文で、タスクを担当する実行ノードを指定するため

    例: `!$xmp task on` **`p(1:4,1:4)`**

    例: `!$xmp task on` **`(1:16)`**

    例: `!$xmp task on` **`p(:,*)`**

    例: `!$xmp task on` **`m`**

  - Loop 指示文で、反復を担当する実行ノードを指定するため

    例: `!$xmp loop (i) on` **`p(lb(i):lb(i+1)-1)`**

- Barrier 指示文、Reduction 指示文と Bcast 指示文で、実行ノードを指定するため
  例: `!$xmp barrier on` **`p(5:8)`**
  例: `!$xmp reduction (+:a) on` **`p(*,:)`**
  例: `!$xmp bcast b from p(k) on` **`p(:)`**

- Bcast 指示文で、通信元の指定のため
  例: `!$xmp bcast b from` **`p(k)`** `on p(:)`

3.1.3 ノードのプログラム例

- communicator の生成を行えばよい。子 communicator の生成では親実行ノード全員の参加が必要となるので、一般には連続する Task 構文は同時には実行できない。
- Tasks 構文がある場合、Tasks 構文の入口で、すべての子実行ノード群を評価し、それぞれに対応する子 communicator を生成しておく。

[Example]

```
      subroutine caller
!$xmp nodes p(1000)
      real a(100,100)

!$xmp tasks
!$xmp  task on p(1:500)
       call task1(a)
!$xmp  end task
!$xmp  task on p(501:800)
       call task1(a)
!$xmp  end task
!$xmp  task on p(801:1000)
       call task1(a)
!$xmp  end task
!$xmp end tasks

      end do
```

```
      subroutine task1(a)
!$xmp nodes q(*)
      real a(100,100)


      end subroutine
```

3.2 template とデータマッピング宣言指示文

3.2.1 Template 指示文

[Summary]
テンプレートを宣言する。

[Syntax]
[F]
!$xmp template *template-name*(*template-spec [, template-spec]…*)

ここで、template-spec は次のいずれかである。

[int-expr :] int-expr
:

[C]
#pragma xmp template *template-name*(*template-spec [, template-spec]…*)

ここで、template-spec は次のいずれかである。

[int-expr :] int-expr
:

- 制約: *template-spec* の並びは、全て「*[int-expr :] int-expr*」であるか、または全て「**:**」でなければならない。

[Description]

template-spec の並びで指定される形状のテンプレートを宣言する。

template-spec の並びが全て「**:**」である場合、テンプレートは初期不定である。そのようなテンプレートは、実行時に **template_fix** 指示文によって確定されるまで参照できない。

template-spec として「*int-expr*」が指定されたとき、当該次元の暗黙の下限値は 1 である。

3.2.2 Distribute 指示文

[Summary]

テンプレートの分散を指定する。

[Syntax]
[F]

```
!$xmp distribute template-name(dist-format [, dist-format]…)
                              onto nodes-name
```

ここで、*dist-format* は次のいずれかである。

```
*
block
cyclic[(int-expr)]
gblock({*|int-array})
```

[C]

```
#pragma xmp distribute template-name(dist-format [, dist-format]…)
                              onto nodes-name
```

ここで、*dist-format* は次のいずれかである。

```
*
block
cyclic[(int-expr)]
gblock({*|int-array})
```

- 制約:「*****」でない *dist-format* の数は、*nodes-name* で指定されるノード集合の次元数と等しくなければならない。
- 制約: **gblock** に続く括弧内に現れる *int-array* は整数型の一次元配列であり、かつその大きさは、*nodes-name* で指定されるノード集合の対応する次元の大きさと等しくなければならない。
- 制約: **gblock** に続く括弧内に現れる *int-array* の値は非負でなければならない。
- 制約: **gblock** に続く括弧内に現れる *int-array* の値の総和は、*template-name* で指定されるテンプレートの対応する次元の大きさと等しいか、または、より大きくなければならない。

[Description]
指定した分散フォーマットに従い、テンプレートをノード集合へ分散する。このとき、「*****」でない *dist-format* が指定されている次元は、**onto** 節の *nodes-name* で指定されているノード集

合の次元に、左から右の順で対応する。

*dist-format*として一つ以上の「**gblock(*)**」が指定されている場合、テンプレートの分散は初期不定である。そのようなテンプレートは、実行時に **template_fix** 指示文によって確定されるまで参照できない。

各々の *dist-format* の意味は次の通りである。

- *****
  テンプレートの当該次元は分散されない。
- **block**
  テンプレートの当該次元は、連続する要素から成る均一なサイズのブロックに分割され、各ブロックは、対応するノード集合の次元のインデックスへ順に割り当てられる。
  テンプレートの当該次元のサイズ d が、対応するノード集合の次元のサイズ p で割り切れない場合、テンプレートの当該次元は、連続する `ceil(d/p)` 個の要素から成る `d/ceil(d/p)` 個のブロックと、`d%ceil(d/p)` 個の要素から成る `1` 個のブロックに分割され、各ブロックは、対応するノード集合の次元のインデックスへ順に割り当てられる。このとき、`k = p - d/ceil(d/p) - 1 > 0` であれば、最後の `k` 個のインデックスに割り当てられるブロックが存在しないことに注意されたい。
- **cyclic**
  cyclic(1)と等価な意味を持つ。
- **cyclic(n)**
  テンプレートの当該次元は、連続する **n** 要素ずつから成る複数のブロックに分割され、各ブロックは、対応するノード集合の次元のインデックスへラウンドロビン形式で割り当てられる。
- **gblock(m)**
  テンプレートの当該次元は、連続する要素から成るブロックに分割され、各ブロックは、対応するノード集合の次元のインデックスへ順に割り当てられる。ここで、ノード集合の `i` 番目のインデックスに割り当てられるブロックのサイズは、マッピング配列 **m** の `i` 番目の要素**で**与えられる。

分散フォーマットは、テンプレートの各次元のインデックスを、対応するノード集合の次元のインデックスへ割り当てる(分散する)方法を指定する。その結果、各テンプレート要素を割り当てるべきノード要素が決まる。

- 例 1

```
!$xmp nodes p(4)
!$xmp template t(64)
!$xmp distribute t(block) onto p
```

テンプレート **t** はブロック分散され、各ノード要素に次のように割り当てられる。

p(1)	**t(1:16)**
p(2)	**t(17:32)**
p(3)	**t(33:48)**
p(4)	**t(49:64)**

- 例 2

```
!$xmp nodes p(4)
!$xmp template t(64)
!$xmp distribute t(cyclic(8)) onto p
```

テンプレート **t** はブロックサイズ 8 でサイクリック分散され、各ノード要素に次のように割り当てられる。

p(1)	**t(1:8)** **t(33:40)**
p(2)	**t(9:16)** **t(41:48)**
p(3)	**t(17:24)** **t(49:56)**
p(4)	**t(25:32)** **t(57:64)**

- 例 3

```
!$xmp nodes p(8,5)
!$xmp template t(64,64,64)
!$xmp distribute t(*,cyclic,block) onto
```

テンプレート **t** の一次元目は分散されない。二次元目はノード **p** の一次元目へサイクリック分散され、三次元目はノード **p** の二次元目へブロック分散される。各ノード要素への割り当ては、次のようになる。

p(1,1)	**t(1:64, 1:57:8, 1:13)**
p(2,1)	**t(1:64, 2:58:8, 1:13)**
…	…
p(8,5)	**t(1:64, 8:64:8, 53:64)**

ここで、**t** の三次元目のサイズ 64 は、**p** の二次元目のサイズ 5 で割り切れないため、**t** の三次元目の分散が不均等になっていることに注意されたい（**p(1,1)**では **t** の三次元目のサイズが 13 であるのに対し、**p(8,5)**では 12 になっている)。

3.2.3 Align 指示文

[Summary]
配列の整列を指定する。

[Syntax]
[F]

```
!$xmp align array-name(align-source [, align-source]…)
      with template-name(align-subscript [, align-subscript]…)
```

ここで、*align-source* は次のいずれかである。

```
scalar-int-variable
*
```

align-subscript は次のいずれかである。

```
scalar-int-variable {+|-} int-expr
*
```

[C]

```
#pragma xmp align array-name[align-source][[align-source]]…
      with template-name(align-subscript [, align-subscript]…)
```

ここで、*align-source* は次のいずれかである。

```
scalar-int-variable
*
```

align-subscript は次のいずれかである。

scalar-int-variable [{+|-} int-expr]
*

align-source に現れる *scalar-int-variable* を「整列ダミー変数」と呼ぶ。

- 制約: *align-subscript* **の並びの中で**、同一の整列ダミー変数は高々1 回だけ現れること**ができる。**
- 制約: *align-subscript* **に、**整列ダミー変数は高々1 回だけ現れることができる。
- 制約: *align-subscript* の *int-expr* の中に、整列ダミー変数が現れてはならない。

[Description]
array-name で指定される配列を、*template-name* で指定されるテンプレートへ整列させる。すなわち、*align-source* の並びで表される配列要素の各々を、*align-subscript* の並びで表されるテンプレート要素へそれぞれ整列させる。

ここで、*align-source* および *align-subscript* は、次のように解釈される。

- 第一の形式の *align-source* と *align-subscript* は、それぞれ整列ダミー変数とその (制限された) 式を表している。整列ダミー変数の値域は、配列の当該次元における全ての有効なインデックスの集合である。
- 第二の形式の *align-source* と *align-subscript* は、当該指示文の他のどこにも現れない(整列ダミー変数でない)ダミー変数を表す。その値域は、配列またはテンプレートの当該次元における全ての有効なインデックスの集合である。
  - 第二の形式の *align-source* により、配列の当該次元を「縮退」させることができる。この結果、当該次元のインデックスは、整列先のテンプレート要素を決定するのに影響しない。
  - 第二の形式の *align-subscript* により、配列を「複製」することができる。すなわち、配列の各要素は複製され、テンプレートの当該次元における全てのインデックスに整列される。

- 例 1

```
!$XMP align a(i) with t(i)
```

配列要素 **a(i)**は、テンプレート要素 **t(i)**に整列される。

- 例 2

```
!$xmp align a(*,j) with t(j)
```

部分配列 **a(:,j)**は、テンプレート要素 **t(j)**に整列される。**a** の一次元目は縮退している。

- 例 3

```
!$xmp align a(j) with t(*,j)
```

配列要素 **a(j)**は複製され、**t(1,j)**～**t(10,j)**の各テンプレート要素へ整列される (**t** の一次元目の上下限をそれぞれ1および 10 とする)。

- 例 4

```
!$xmp template t(n1,n2)
      real a(m1,m2)
!$xmp align a(*,j) with t(*,j)
```

部分配列 **a(:,j)**は、**t(1,j)**～**t(n1,j)**の各テンプレート要素へ整列される。

a の一次元目の「*****」をダミー変数 **i** に、**t** の一次元目の「*****」をダミー変数 k に、それぞれ置き換えると、この **align** 指示文が指示する整列は次のように解釈できる。

```
{a(i,j) → t(k,j) | (i,j,k) ∈ (1:n1,1:n2,1:m1)}
```

- コメント:「データモデル」の章を設けて、「整列」という概念を説明する必要がある。

3.2.4 Shadow **指示文**

[Summary]
配列にシャドウを付加する。

[Syntax]
[F]
!$xmp shadow *array-name*(*shadow-width [, shadow-width]…*)

ここで、*shadow-width* は次のいずれかである。

int-expr
int-expr **:** *int-expr*

[C]
#pragma xmp shadow *array-name***[***shadow-width***][[***shadow-width***]]…**

ここで、*shadow-width* は次のいずれかである。

int-expr
int-expr **:** *int-expr*

- 制約: *shadow-width* に指定される整数式は、0 以上の値を持たなければならない。

[Description]
array-name で指定される配列の各次元に付加するシャドウの幅を指定する。*shadow-width* が「*int-expr* **:** *int-expr*」であるとき、ローカル配列の当該次元の上端と下端に、それぞれ指定される幅のシャドウが付加される。*shadow-width* が「*int-expr*」であるとき、ローカル配列の当該次元の上端と下端に、同じ幅のシャドウが付加される。

シャドウ領域に保持されるデータを「シャドウ実体」と呼ぶ。各シャドウ実体は、それぞれ特定の配列要素を表現している。あるシャドウ実体は、同じ配列要素を表現するデータ実体に対応すると考えられる。そのようなデータ実体を、シャドウ実体の「反映元」と呼ぶ。

3.2.5 Template_fix 指示文

[Summary]
実行指示文。テンプレートを確定する。

[Syntax]
[F]
!$xmp template_fix *template_name*[**(***template-spec [, template-spec]…***)**]
*[***,distribute (***dist-format [, dist-format]…***)**]

ここで、*template-spec* は次の通りである。

[int-expr **:***] int-expr*

ここで、*dist-format* は次のいずれかである。

*****
block
cyclic*[***(***int-expr***)**]
gblock(*int-array***)**

[C]

#pragma xmp template_fix ???

- 制約: **template_fix** 指示文が実行される時点において、*template-name* で指定されるテンプレートは不定でなければならない。
- 制約: **template_fix** 指示文に現れる *dist-format* の並びと、*template-name* で指定されるテンプレートに対する **distribute** 指示文に現れる *dist-format* の並びは、**gblock** に続く括弧内を除いて一致していなければならない。
- 制約: *template-spec* の並びと **distributed** 指示節のうち、少なくとも一つが指定されなければならない。
- 制約: **template_fix** 指示文は、実行部にのみ現れることができる。

[Description]

template_fix 指示文は、実行指示文である。各次元のサイズまたは分散フォーマット指定することにより、初期不定であるテンプレートを実行時に確定する。

初期不定であるテンプレートに整列している配列は割付け配列でなければならず、当該テンプレートが **template fix** 指示文によって確定されるまでは割付けることはできない。当該テンプレートを on 節に指定する実行指示文は、当該テンプレートが template_fix 指示文によって確定される前に実行されてはならない。一つ目の制約により、あるテンプレートに対して、複数回の **template fix** 指示文を実行することはできない。

distribute 指示節に指定する *dist-format* の意味は、**distribute** 指示文のそれに準ずる.

- 例

```
!$XMP template :: t(:)
!$XMP distribute (gblock(*)) :: t

      real , allocatable :: a(:)
!$XMP align (i) with t(i) :: a

      …

      N = …; M(…) = …

!$XMP template_fix t(N), distribute (gblock(M))

      allocate (a(N))
```

形状が「**(:)**」であり、分散フォーマットが「**gblock(*)**」なので、テンプレート **t** は初期不定である。また、割付け配列 **a** は **t** に整列している。

t のサイズ **N** とマッピング配列 **M** を定義した後、**template_fix** 指示文によって **t** を確定する。**t** を確定した後、**a** を割付けることができる。

3.3 ワークマッピング構文

3.3.1 Task 指示構文

[Summary]

タスク並列実行のための1つのタスクを表現する。

[Syntax]

[F]

!$xmp task on { *nodes-ref* | *template-ref* }

block

!$xmp end task

[C]

#pragma xmp task on { *nodes-ref* | *template-ref* }

block

[Restriction]

- *nodes-ref* または *template-ref* によって指定される実行ノードの集合は、構文の外側を冗長実行する実行ノードの集合（以降、親実行ノード）に包含されていなければならない。

[Description]

nodes-ref または *template-ref* によって、ブロック（*block*）を実行する実行ノードを指定する。親実行ノードのうち、ブロックの実行ノードはブロックを冗長実行し、それ以外のノードはブロックの実行を行わず次の実行文へ進む。

nodes-ref と *template-ref* は、Task 構文が静的に Tasks 構文で囲まれていないとき、Task 構文の入口で評価される。Tasks 構文で囲まれているとき、Tasks 構文の入口で評価される。評価の結果は、親実行ノードの全てで同じでなければならない。

3.3.2 Tasks 指示構文

[Summary]

複数の Task 指示構文をまとめて、同時実行（タスク並列実行）させる。

[Syntax]

[F]

```
!$xmp tasks [ nowait ]
        task-directive-construct
        ...
!$xmp end tasks
```

[C]

```
#pragma xmp tasks [ nowait ]
    {
        task-directive-construct
        ...
    }
```

[Description]

直下の Task 構文（*task-directive-construct*）（以下、子 Task 構文）について、その入口と出口で暗黙の同期処理を行わないことを保障する。すなわち、連続する2つの子 Task 構文の実行では、それらの実行ノード群に重なりがない限り、同時実行されることが期待できる。

すべての子Task構文のTask指示文の *nodes-ref* と *template-ref*は、親実行ノードがTasks構文に出会ったときに評価されて確定する。

Tasks 構文の入口では暗黙の同期処理は行われない。

Tasks 構文の出口では、nowait が指定されたとき暗黙の同期処理は行われない。nowait が

指定されないとき、子 Task 構文が発行したすべての Gmove 片側通信が完了していることを互いに確認するため、親実行ノード群に対して暗黙の同期が行われる。

【備考】

- nowait はバリア同期だけではなく、片側通信の完了の待ち合わせの意味を持つ。つまり、nowait 節を指定した場合には、構文実行後に明示的なバリア同期を実行しても、片側通信が完了していることは保証されない。
- 構文入口での同期には明示的な Barrier 指示行を使えばよい。

【MPI による実装】

- Tasks 構文がない場合、Task 構文の入口で子実行ノード群を評価し、それに従って子 communicator の生成を行えばよい。子 communicator の生成では親実行ノード全員の参加が必要となるので、一般には連続する Task 構文は同時には実行できない。
- Tasks 構文がある場合、Tasks 構文の入口で、すべての子実行ノード群を評価し、それぞれに対応する子 communicator を生成しておく。

[Example]

```
      subroutine caller
!$xmp nodes p(1000)
      real a(100,100)

!$xmp tasks
!$xmp  task on p(1:500)
        call task1(a)
!$xmp  end task
!$xmp  task on p(501:800)
        call task1(a)
!$xmp  end task
!$xmp  task on p(801:1000)
        call task1(a)
!$xmp  end task
!$xmp end tasks

      end do
```

```
      subroutine task1(a)
!$xmp nodes q(*)
      real a(100,100)



      end subroutine
```

3.3.3 Loop 指示構文

[Summary]
必要なら集計の詳細を示し, ループの負荷分散を指示する.

[Syntax]
[F]
!$xmp loop *[(loop-index [, loop-index] …)]* **on** *on-ref [reduction-ref]*

[C]
#pragma xmp loop *[(loop-index [, loop-index] …)]* **on** *on-ref [reduction-ref]*

[共通]
ここで, on-ref は, 次のいずれかである.

template-ref
nodes-ref

ここで, reduction-ref は, 以下で定義される.

reduction （ *reduction-kind* : *reduction-spec [, …]* ）

reduction-kind は, 次のいずれかである.

+
*****
MAX
MIN
FIRSTMAX
FIRSTMIN
LASTMAX
LASTMIN

reduction-spec は, 以下で定義される.

reduction-variable [/ location-variable [, …] /]

- 制約：loop-index は, 直後のループおよびそのループに内包されるループについてのル

ープインデックスでなければならない.

- 省略：loop-index の並びが省略された場合は，直後のループのループインデックスが指定されたものと見なす.
- 制約：template-ref に現れる template-spec は，「*」か「:」か loop-index でなければならない. ただし，loop-index は，直後のループまたはそのループを外包しているループについてのループインデックスでなければならない.
- 制約：nodes-ref は，loop-index の値毎に異なったノード集合を参照しなければならない. また，ノード集合は，それぞれ異なったノードから構成されていなければならない. つまり，あるノードは，二つ以上のノード集合に含まれてはならない.
- 制約：Loop 構文が多重で使われる場合，on-ref は包含関係を満足していなければならない.
- 制約：loop-index に対応するループのパラメータ（初期値，終値，増分）はループ内で不変でなければならず，実行ノード集合内で一致していなければならない.
- 制約：reduction-ref は，直後のループおよびそのループに内包されるループについての集計演算でなければならない.
- 制約：reduction-kind が，FIRST(LAST)MAX(MIN) のいずれかである場合のみ，reduction-spec は，一つ以上の location-variable を持たなければならない.
- 制約：location-variable が指定された場合は，直後のループおよびそのループに内包されるループにおいて，その値を確定していなければならない.
- 制約：reduction-variable は，その変数自体の更新を除いて，中間的な値がループ中で参照されてはならない.
- 制約：reduction-variable および location-variable は，内包するループに対する reduction-ref に現れてはならない.

[Description]

インデックス付きループ文の前に置き，直後のループおよびそのループに内包されるループについて，コンパイラに負荷分散を指示する．ノード毎のループ実行範囲をループの外で決定するため，効率の良い負荷分散が期待できる.

負荷分散を指定するループインデックスについて集計演算を行っている場合，その詳細を示すことで並列実行を行っても全体として正しい集計演算結果を得ることができる.

on-ref が template-ref の場合，template の分散に従って loop-index に対応するノード集合が決定され，そのノード集合を実行ノード集合とする．このため，この構文が評価されるときには，参照される template は確定していなければならない．template-spec が「*」の場合，その次元は縮退していると見なし，割り当てに関して無視される．template-spec が「:」の場合，

その次元の全てのテンプレート要素に対応するノードを割り当ての対象とする.

on-ref が nodes-ref の場合, それぞれの loop-index の値毎に参照されたノード集合を実行ノード集合としてタスクを生成し, その loop-index の値について実行する.

reduction-kind として,「+」,「*」,「max」または「min」が指定された reduction-ref を持つ template-ref 指定の Loop 構文は, Loop 構文から reduction-ref を削除し, template-ref における「:」を「*」に, そのループのループインデックスをそのループの実行範囲に置き換え,「*」およびその他の変数はそのままにした template-ref を持つ Reduction 構文をループ直後に置いたプログラムと等価である. 即ち,

```
!$xmp loop (j) on t(:,j) reducion(...)
     do j = js, je
        ...
        do i = 1, N
           ...
        end do
       ...
     end do
```

は, 以下の Reduction 構文と等価である.

```
!$xmp loop (j) on t(:,j)
     do j = js, je
        ...
        do i = 1, N
           ...
        end do
        ...
     end do
!$xmp reducion(...) on t(*,js:je)
```

- 例 1

```
!$xmp distribute t(block) onto p
!$xmp align (i) with t(i) :: a, b
      ...
!$xmp loop on t(i)
      do i = 1, N
        a(i) = 1.0
        b(i) = a(i)
      end do
```

ループは, テンプレート t の分散に従って実行するノードを決定し, 負荷分散される. この例は, 以下の Task 構文と等価であるが, ノード毎のループ実行範囲をループ外で決定するため, より高速な実行が期待できる.

```
!$xmp distribute t(block) onto p
!$xmp align (i) with t(i) :: a, b
      ...
      do i = 1, N
!$xmp task on t(i)
        a(i) = 1.0
        b(i) = a(i)
!$xmp end task
      end do
```

- 例 2

```
!$xmp distribute t(*,block) onto p
!$xmp align (*,j) with t(*,j) :: a, b
      ...
!$xmp loop (j) on t(*,j)
      do j = 1, M
        do i = 1, N
          a(i,j) = 1.0
          b(i,j) = a(i,j)
        end do
      end do
```

テンプレート t の 1 次元目は縮退しているため, ループは 2 次元目についてのみ負荷分散される.
この例は, 以下の Task 構文と等価である.

```
!$xmp distribute t(*,block) onto p
!$xmp align (*,j) with t(*,j) :: a, b
      ...
      do j = 1, M
!$xmp task on t(*,j)
        do i = 1, N
          a(i,j) = 1.0
          b(i,j) = a(i,j)
        end do
!$xmp end task
      end do
```

- 例 3

```
!$xmp distribute t(block,block) onto p
!$xmp align (i,j) with t(i,j) :: a, b
      ...
!$xmp loop (i,j) on t(i,j)
      do j = 1, M
        ...
        do i = 1, N
          a(i,j) = 1.0
          b(i,j) = a(i,j)
        end do
        ...
      end do
```

多次元ループに対しては, loop-index の並びを用いて記述することができる. この例は, 以下の多重 Loop 構文と等価であるが, ノード毎のループ実行範囲を多重ループの外で決定するため, より高速な実行が期待できる. なお, template-ref が, 包含関係を満足していることに注意されたい.

```
!$xmp distribute t(block,block) onto p
!$xmp align (i,j) with t(i,j) :: a, b
      ...
!$xmp loop (j) on t(:,j)
      do j = 1, M
        ...
!$xmp loop (i) on t(i,j)
        do i = 1, N
          a(i,j) = 1.0
          b(i,j) = a(i,j)
        end do
        ...
      end do
```

- 例4

```
!$xmp nodes p(10,3)
      ...
!$xmp loop on p(:,i)
      do i = 1, 3
         call subtask ( i )
      end do
```

i の値に従って p(:,i)で参照されるそれぞれ異なった 10 個のノードから実行ノード集合を構成し, それぞれのループの繰り返しを別のタスクとして実行する. この例は, 以下のループを用いた Task 構文または静的な Tasks/Task 構文と等価である.

```
!$xmp nodes p(10,3)
      ...
      do i = 1, 3
!$xmp task on p(:,i)
        call subtask ( i )
!$xmp end task
      end do
      ...
!$xmp tasks
!$xmp task on p(:,1)
      call subtask ( 1 )
!$xmp end task
!$xmp task on p(:,2)
      call subtask ( 2 )
!$xmp end task
!$xmp task on p(:,3)
      call subtask ( 3 )
!$xmp end task
!$xmp end tasks
```

- 例5

```
      ...
      lb(1)  = 1
      iub(1) = 10
      lb(2)  = 11
      iub(2) = 25
      lb(3)  = 26
      iub(3) = 50
!$xmp loop (i) on p(lb(i):iub(i))
      do i = 1, 3
        call subtask ( i )
      end do
```

i の値に従って異なったノード数の実行ノード集合を構成し, 不均等な負荷分散を実現する. この例は, 以下のループを用いた Task 構文または静的な Tasks/Task 構文と等価である.

```
      do i = 1, 3
!$xmp task on p(lb(i):iub(i))
        call subtask ( i )
!$xmp end task
      end do
      ...
!$xmp tasks
!$xmp task on p(1:10)
      call subtask ( 1 )
!$xmp end task
!$xmp task on p(11:25)
      call subtask ( 2 )
!$xmp end task
!$xmp task on p(26:50)
      call subtask ( 3 )
!$xmp end task
!$xmp end tasks
```

■ 例6

```
      ...
      s = 0.0
!$xmp loop (i) on t(i) reduction(+:s)
      do i = 1, N
        s = s + a(i)
      end do
```

それぞれのノードのスカラー変数 s に正しい集計演算結果が格納される. reduction-ref がない場合, s はそれぞれのノードの部分和を保持するだけで, 正しい集計演算結果にはならない. この例は, 以下の Reduction 構文と等価である.

```
      ...
      s = 0.0
!$xmp loop (i) on t(i)
      do i = 1, N
        s = s + a(i)
      end do
!$xmp reduction(+:s) on t(1:n)
```

- 例7

```
      ...
     amax = -1.0e30
     ip = -1
     jp = -1
!$xmp loop (i,j) on t(i,j) reduction(firstmax:amax/ip,jp/)
      do j = 1, M
         do i = 1, N
            if( a(i,j) .gt. amax ) then
               amax = a(i,j)
               ip = i
               jp = j
            end if
         end do
      end do
```

それぞれのノードのスカラー変数 amax に配列 a(i,j)の最大値が格納される. reduction-ref がない場合, amax はそれぞれのノード内の部分最大値を保持するだけで, 正しい集計演算結果にはならない. また, 最大値が複数あった場合, このプログラムを逐次実行すると最大値を与えるインデックスは, 最初に出現したインデックスになるため, firstmax を指定している. なお, この例は, Reduction 構文では表現できない.

3.3.4 Array 指示構文

[Summary]
直後の配列代入文について要素毎の負荷分散を指示する.

[Syntax]
[F]
!$xmp array on *template-ref*

[C]
未サポート

- 制約:`template-spec` は, 直後の配列代入文と形状適合しなければならない.
- 制約:この構文は集団実行を前提としているので, 特定のノードのみが実行するようガードさ

れた領域で記述してはならない.

- 省略:`template-spec`の要素範囲が省略された場合は, 全範囲を指定したものと見なす.

[Description]

Fortran では, 配列に対する代入をループで記述する替わりに配列代入文を利用できる. この配列が分散配列である場合, コンパイラにノード毎に代入の実行範囲が異なることを通知しなければならない.

- 例 1

```
!$xmp distribute t(block) onto p
!$xmp align (i) with t(i) :: a
      ...
!$xmp array on t(1:N)
      a(1:N) = 1.0
```

この例は, 以下の Loop 構文と等価である.

```
!$xmp distribute t(block) onto p
!$xmp align (i) with t(i) :: a
      ...
!$xmp loop on t(i)
      do i = 1, N
        a(i) = 1.0
      end do
```

- 例2

```
!$xmp template t(100,20)
!$xmp distribute t(block,block) onto p
      dimension a(100,20)
!$xmp align a(i,j) with t(i,j)
      ...
!$xmp array on t
      a = 1.0
```

この例は, 以下の Loop 構文と等価である.

```
!$xmp template t(100,20)
!$xmp distribute t(block,block) onto p
      dimension a(100,20)
!$xmp align a(i,j) with t(i,j)
      ...
!$xmp loop (i,j) on t(i,j)
      do j = 1, 20
         do i = 1, 100
            a(i,j) = 1.0
         end do
      end do
```

3.4 グローバルビュー通信・同期指示文

3.4.1 Reflect 指示文

[Summary]
シャドウを持つ変数について、シャドウ実体に反映元の値を設定する。

[Syntax]
[F]
!$xmp reflect *array-name [, array-name]…*

[C]
#pragma xmp reflect *array-name [, array-name]…*

[Description]
array-name で指定される実体が持つ全てのシャドウ実体に、その反映元のデータ実体の値を複写する。

3.4.2 Barrier 指示文

[Summary]
バリア同期を実行する。

[Syntax]

[F]
!$xmp barrier *[***on** *nodes-ref|template-ref]*

[C]
#pragma xmp barrier *[***on** *nodes-ref|template-ref]*

- 制約: **on** 指示節から決定されるノードの集合は、実行ノード集合の部分集合でなければならない。

[Description]
on 指示節（省略されている場合には現在のコンテキスト）で決定される実行ノード集合でバリア同期を実行する。

3.4.3 reduction 指示文

[Summary]
集計演算を実行する。

[Syntax]
[F]
!$xmp reduction(*reduction-kind* **:** *variable [***,** *variable]…***)**
*[***on** *nodes-ref|template-ref]*

ここで、*reduction-kind* は、「**+**」「*****」「**.AND.**」「**.OR.**」「**.EQV.**」「**.NEQV.**」「**MAX**」「**MIN**」「**IAND**」「**IOR**」「**IEOR**」のいずれかである。

[C]
#pragma xmp reduction(*reduction-kind* **:** *variable [***,** *variable]…***)**
*[***on** *nodes-ref|template-ref]*

ここで、*reduction-kind* は、「**+**」「*****」「**-**」「**&**」「**|**」「**^**」「**&&**」「**||**」「**max**」「**min**」のいずれかである。

variable で指定される変数を「集計変数」と呼ぶ。

- 制約: 集計変数が分散配列または分散配列の部分実体であり、かつ **on** 指示節が指定されている場合、その分散配列の整列先テンプレート(以下、**T** と呼ぶ)は、*nodes-name* で指定

されるノード配列(以下、**N** と呼ぶ)へ分散されていなければならない。

- 制約: **on** 指示節が指定されている場合、「*subscript-triplet*」である *node-subscript* が指定されている **N** の次元に関し、次の条件が満たされなくてはならない。
  - **T** に対する **distribute** 指示文において、**N** の当該次元に対応する次元に指定されている分散フォーマットは「*****」ではない。
  - 集計変数に対する **align** 指示文において、整列先として現れる **T** の当該次元の *align-subscript* は「*****」である。
- 制約: **reduction** 指示文は集団的である。したがって、ある **reduction** 指示文は全実行ノードによって実行され、かつそこに現れる全ての変数の値は全実行ノードにおいて同一でなければならない。

[Description]

on 指示節（省略されている場合には文脈）で決定される実行ノードが、*reduction-kind* で指定された演算によって、実行ノード集合に複製されている集計変数を互いに集計する。

■ 例1

```
!$xmp reduction(+:s)

!$xmp reduction(max:aa) on t(*,:)

!$xmp reduction(min:bb) on p(10:30)
```

最初の例は, 実行ノード集合において各ノードが保持しているスカラー変数 s の値をそれぞれの部分和だと見なして総和を計算し, 結果を各ノードに通知する.

2 番目の例は, template t の 2 次元目の全範囲をカバーするノード群を集計演算実行用のノード集合とし, 各ノードが保持しているスカラー変数 aa の値の最大値を計算し, 結果を各ノードに通知する.

3 番目の例は, ノード p(10:30)において各ノードが保持しているスカラー変数 bb の値の最小値を計算し, 結果を各ノードに通知する. なお, この例は, Task 構文を用いた以下の例と等価である.

```
!$xmp task on p(10:30)
!$xmp reduction(min:bb)
!$xmp end task
```

■ 例2

```
      dimension a(n,n), p(n), w(n)
!$xmp align a(i,j) with t(i,j)
!$xmp align p(i) with t(i,*)
!$xmp align w(j) with t(*,j)
      ...
!$xmp loop (j) on t(:,j)
      do j = 1, n
         sum = 0
!$xmp loop (i) on t(i,j) reduction(+:sum)
         do i = 1, n
            sum = sum + a(i,j) * p(i)
         end do
         w(j) = sum
      end do
```

行列とベクトルの積を計算するこのプログラムでは, 内側のループに Loop 構文の reduction 節を指定している. この例は, 以下の Reduction 構文と等価である.

```
      dimension a(n,n), p(n), w(n)
!$xmp align a(i,j) with t(i,j)
!$xmp align p(i) with t(i,*)
!$xmp align w(j) with t(*,j)
      ...
!$xmp loop (j) on t(:,j)
      do j = 1, n
        sum = 0
!$xmp loop (i) on t(i,j)
        do i = 1, n
          sum = sum + a(i,j) * p(i)
        end do
!$xmp reduction(+:sum) on t(1:n,j)
        w(j) = sum
      end do
```

この場合, スカラー変数 sum に対して, 外側ループの分割数と同じ回数だけ集計演算のための通信が発生する. この通信をループの外に追い出すため, 外側の Loop 構文に reduction 節を記述することはできない. なぜなら, 外側のループはループインデックス j についての負荷分散であるが, 集計演算はループインデックスiについて行っているためである. しかし, Reduction 構文では, 以下のように記述できる.

```
      dimension a(n,n), p(n), w(n)
!$xmp align a(i,j) with t(i,j)
!$xmp align p(i) with t(i,*)
!$xmp align w(j) with t(*,j)
      ...
!$xmp loop (j) on t(:,j)
      do j = 1, n
        sum = 0
!$xmp loop (i) on t(i,j)
        do i = 1, n
          sum = sum + a(i,j) * p(i)
        end do
        w(j) = sum
      end do
!$xmp reduction(+:w) on t(1:n,*)
```

この場合, 1 次元配列 w について, 一回の集計演算を実行すればいいので, 効率の良い集合演算が実現できる.

3.4.4 bcast 指示文

[Summary]
ブロードキャスト通信を実行する。

[Syntax]
[F]

```
!$xmp bcast variable [, variable]… [from nodes-ref]
                                        [on nodes-ref|template-ref]
```

[C]

```
#pragma xmp bcast variable [, variable]… [from nodes-ref]
                                        [on nodes-ref|template-ref]
```

variable で指定される変数を「ブロードキャスト変数」と呼ぶ。

- 制約: ブロードキャスト変数が分散配列または分散配列の部分実体であり、かつ **on** 指示節

が指定されている場合、その分散配列の整列先テンプレート(以下、**T** と呼ぶ)は、*nodes-name* で指定されるノード配列(以下、**N** と呼ぶ)へ分散されていなければならない。

- 制約: **on** 指示節が指定されている場合、*subscript-triplet* である *node-subscript* が指定されている **N** の次元に関し、次の条件を満たされなくてはならない。
  - **T** に対する **distribute** 指示文において、**N** の当該次元に対応する次元に指定されている分散フォーマットは「*****」ではない。
  - 集計変数に対する **align** 指示文において、整列先として現れる **T** の当該次元の *align-subscript* は「*****」である。
- 制約: **bcast** 指示文は集団的である。したがって、ある **bcast** 指示文は全実行ノードによって実行され、かつそこに現れる全ての変数の値は全実行ノードにおいて同一でなければならない。
- 制約: **from** 指示節で指定されるノードは、**on** 指示節から決定されるノードの集合に含まれていなければならない。

[Description]

from 指示節で指定されるノード(「ソースノード」と呼ぶ)が、*variable* の並びで指定される各変数を、**on** 指示節(なければ文脈)から決定される実行ノードに属する各ノードへブロードキャストする。本指示文の実行後、各実行ノードにおいて、*variable* の並びで指定される各変数の値は、本指示文の実行前のソースノード上の値に一致する。

また、**from** 節が省略されている場合、実行ノード配列のうち、最初のノードがソースノードとして指定されていると見なす。

3.4.5 Gmove 構文

[Summary]

グローバルビューにて定義された分散データ変数について、データ移動を行う。

[Syntax]

[F]

```
!$xmp gmove [in | out]
     代入文
```

[C]

```
#pragma xmp gmove [in | out]
     代入文
```

[Description]
グローバルビューにて定義された分散配列などのデータのコピー操作を各ノードが分担して集団的に行う。代入文は、スカラー変数の代入、部分配列(array section)間の代入がある。分散配列を参照する部分配列(array section)による配列間代入の場合には、ノード間の通信により代入操作を行うことになる。なお、代入文は、算術演算を伴わない単純な代入に限ることにする。XcalableMP においては、C 言語においても、部分配列が記述できるように拡張仕様を定めている。

構文の本体に出現できる実行文は、以下のいずれか:

- スカラ代入文。ただし、右辺は演算を含まない。例えば:

```
s1 = s2                 ! s1,s2 はスカラ変数
a(3) = b(i, j)          ! a,b は配列変数
```

- 配列代入文。ただし右辺は配列変数名、配列区分、スカラ変数名または配列要素に限る。例えば:

```
a = b                   ! a,b は配列変数
a(1:10) = b(n:n+9, k)
a(1:10) = s2            ! 左辺は配列、右辺はスカラ
a(1:10) = b(i, j)       ! 左辺は配列、右辺はスカラ
```

ただし、右辺がノードプライベートなデータの場合には、そのデータはすべてのノードで同じ値でなくてはならない。また、左辺にノードプライベートなデータが指定されたときには、同じ値が代入されることになる。すなわち、ブロードキャスト操作になる。

gmove 構文は、現在の node-set の全ノードで同時に実行されなくてはならない。部分配列の代入において、添え字や配列の範囲を指定する値については、この構文を実行するすべてのノードで等しくなくてはならない。

in 節、out 節の指定については以下のとおり:
in, out のいずれの指定もない場合には、左辺、右辺で参照されるデータが現在のノードセットに配置されているデータでなくてはならない。右辺にあるデータは、そのデータのあるノードから送信され、左辺のあるノードで受信され代入が行われる。
in 節が指定された場合には、左辺のデータを保持しているノードが remote copy 操作により、対応する右辺のデータを取得する(get 操作)ことにより、代入操作を行う。したがって、左辺で参照するデータは現在のノードセットに配置されているデータでなくてはならない。
out 節が指定された場合には、右辺のデータを保持しているノードが remote copy 操作により、対

応する左辺のデータを更新する(put 操作)ことにより、代入操作を行う。したがって、右辺で参照するデータは現在のノードセットに配置されているデータでなくてはならない。

同期に関しては、in 節、out 節、いずれの指定のないときには、代入する側と代入される側が同期して動作するために、implicit な同期が取られることになる。in 節、out 節が指定された時には、remote copy 操作によって実行されるために、読み込まれるデータのある側、書き決まれるデータのある側との同期は取られない。よって、双方に explicit な同期が必要な場合には適切な barrier 同期を行う必要がある。同じ実行ノードにない場合には、post-wait 指示文で、明示的に同期を取る必要がある。

■ 例1:配列代入。

右辺左辺とも分散配列の場合、all-to-all の通信が発生する。左辺が重複配列の場合、multicast となる。右辺が重複配列の場合、通信は不要なので Array 指示文(配列要素毎の代入を並列に実行)を使う方が効率がよい(かもしれない)。

`!$xmp gmove` `  a(:,1:N) = b(:,3,0:N-1)`	`#pragma xmp gmove` `  a[1:N][:] = b[0:N-1][3][:];`

例2

右辺が分散配列の場合、特定ノードからのブロードキャスト通信となる。右辺が重複配列の場合、通信は不要なので Array 指示文を使う方が効率がよい(かもしれない)。

`!$xmp gmove` `  a(:,1:N) = c(k)`	`#pragma xmp gmove` `  a[1:N][:] = c[k];`

例

```
!$xmp nodes p(4)
      real a(4),b(4)
!$xmp distribute (block) onto p :: a,b

!$xmp task on p(1:2)              ! p(1),p(2)のみこの区間を実行

!$xmp   gmove in                  ! チーム内の通信a(1)←b(2)と
        a(1:2)=b(2:3)             ! チーム外からの通信a(2)←b(3)
                                  ! が発生する。
!$xmp end task
```

3.5 ローカルビュー機能

XcalableMP では, ローカルビュープログラミングのため, 拡張仕様として coarray 記法を採用する。全ノードを実行ノードとする場合, Coarray Fortran と互換性があるが, coarray 記法で書かれたプログラム群をタスク並列で実行する場合等は, 一部互換性がないので注意すること。

3.5.1 Coarray

通常の coarray と同様に宣言可能であり, 参照できる. ただし, XcalableMP の拡張仕様として宣言の最後に on 節 (**on** *nodes-ref*) を追加することが可能である. この場合, nodes-ref で指定されたノード集合に対して coarray をマッピングする. on 節を省略した場合, 実行ノード集合が指定されたと見なす.

- 例 1

```
!$xmp nodes w(50)
      real wa(100)[*]
      ...
!$xmp tasks
!$xmp task on w(1:30)
      call task1 ( wa )
!$xmp end task
!$xmp task on w(31:50)
      call task2 ( wa )
!$xmp end task
!$xmp end tasks
      ...
      subroutine task1 ( aa )
!$xmp nodes w(50)
!$xmp nodes p(30) = *
      real aa(100)[*] on w
      real b(100)[*]
      ...
      subroutine task2 ( aa )
!$xmp nodes w(50)
!$xmp nodes p(20) = *
      real aa(100)[*] on w
      real c(100)[*]
```

coarray wa は全体ノード集合であるノード w にマッピングされ, 実引数としてサブルーチンに渡され, 仮引数 aa として受け取られる. サブルーチン内では, aa の coarray 宣言に on 節が必要である. 実行ノード集合が, メインとサブルーチンでは, 異なっていることに注意されたい. この coarray 宣言により, サブルーチン内ではメインと同様に wa をアクセス可能である. coarry b (または, c) は, このサブルーチン task1 (または, task2) 内にのみ存在するローカル変数であり, サブルーチン呼びだし時の実行ノード集合である p つまり, w(1:30) (または, w(31:50)) にマッピングされる. サブルーチン内で参照できるノードは, 全体ノード集合およびその静的な部分集合か実行ノード集合のみなので, サブルーチン task1 (または, task2) が coarray c (または, b) を参照する方法は, 存在しない.

- 例2

```
!$xmp nodes w1(200)
      real one(100)[*]
      real two(50)[20,*]
```

全体ノード集合は 200 ノードから構成され, coarray one および two は, そのノード集合にマッピングされる. この場合, one(…)[i+(j-1)*20]と two(…)[i,j]は, 同じノードにマッピングされることが保証され, Coarray Fortran と互換性がある. この例は以下のように記述しても等価である. 即ち, ノードの宣言形状は, coarray には影響を及ぼさない.

```
!$xmp nodes w2(20,10)
      real one(100)[*]
      real two(50)[20,*]

または,

!$xmp nodes w3(10,5,4)
      real one(100)[*]
      real two(50)[20,*]
```

- 例3

```
!$xmp nodes w1(200)
!$xmp nodes w2(20,10)
      real one(100)[*]
      real two(50)[20,*]
      real ichi(100)[*] on w2
      real ni(50)[20,*] on w2
```

多くのアプリケーションでは, coarray の隣接間通信が重要になるが, この通信に最適な論理ノードと物理ノードのマッピングは, 一般には次元数により異なる. 即ち, one(…)[i]･one(…)[i+1]間の通信に最適なマッピングと two(…)[i,j]･two(…)[i+1,j]間の通信に最適なマッピングは異なる. Coarray Fortran では, どちらか一方を選ばなければならないが, XcalableMP では, 複数のマッピングを可能にしている. この例の場合, 通常は Coarray Fortran と互換であり, 実行時オプション等により, どちらか一方を選ぶ. しかし, 実行時オプション等により, ノード w1 と w2 に異なるマッピングを指定することが可能である. この場合, coarray one および two は, ノード w1（つまり, 省略時の全体ノード集合）にマッピングされ, one(…)[i+(j-1)*20]と two(…)[i,j]は同じ物理ノードにマッピングされる. また, coarray ichi および ni は, 異なったノード w2 にマッピングされ, ichi(…)[i+(j-1)*20]と ni(…)[i,j]は同じ物理ノードにマッピングされる. しかし, one(…)[i]と ichi(…)[i]および two(…)[i,j]と ni(…)[i,j]は, 同じ物理ノードにマッピングされることが, 保証されない. このため, Coarray Fortran とは互換がないので注意されたい.

3.5.2 グローバル-ローカルビュー連携指示文

[Summary]
分散されたグローバル配列のローカルエイリアスを宣言する。

[Syntax]
[F]
!$xmp local_alias *local-array-name* **to** *global-array-name*

[C]
#pragma xmp local_alias *local-array-name* **to** *global-array-name*

- 制約: *local-array-name* で指定される配列は、**align** 指示文によって整列されていてはならない。
- 制約: *global-array-name* で指定される配列は、**align** 指示文によって整列されていな

ければならない。

- 制約: *local-array-name* で指定される配列と、*global-array-name* で指定される配列のデータ型および次元数は一致していなければならない。
- [F]制約: *local-array-name* で指定される配列は、形状引継ぎ配列でなければならない。
- [C]制約: *local-array-name* で指定される配列は、？？？？
  - コメント: C で「割付け配列」のようなものをどのように表現するか？ ポインタ？

[Description]

local-array-name で指定されるローカル配列が、*global-array-name* で指定されるグローバル配列のローカルエイリアスであることを宣言する。

ローカルエイリアスの形状は、元のグローバル配列に対応して自ノードに割付けられたローカル配列のそれに一致する。ここで、ローカル配列はシャドウ領域を含むことに注意されたい。

ローカルエイリアスが確定されるのは、対応するグローバル配列が確定される時点である。すなわち、グローバル配列が静的に割付けられている場合、有効域内でローカルエイリアスは常に確定されている。グローバル配列が動的に割付けられる場合、それが割付けられた時点でローカルエイリアスは確定される。

local-array-name で指定される配列は coarray であってもよいことに注意されたい。

- 例 1

```
!$xmp nodes n(4)
!$xmp template :: t (100)
!$xmp distribute (cyclic) onto n :: t

      real :: a (100)
!$xmp align (i) with t(i) :: a
!$xmp shadow (1) :: a

      real :: b(0:)
!$xmp local_alias to a :: b
```

ノード **n(2)** には、25 個の要素から成るローカル配列(**a(2:100:4)**)と両端に幅 1 のシャドウ領域が割付けられている。また、ローカルエイリアス **b** の下限は 0 と宣言されている。したがって、**n(2)** において、**b** は、27 個の要素から成る配列(**b(0:26)**)となる。このとき、**a** の要素 と **b** の要素は次のように対応する。

a	b
下シャドウ	0
2	1
6	2
10	3
…	…
98	25
上シャドウ	26

- 例 2

```
!$xmp nodes n(4)
!$xmp template :: t(:)
!$xmp distribute (block) onto n :: t

      real , allocatable :: a(:)
!$xmp align (i) with t(i) :: a

      real :: b(:)[*]
!$xmp local_alias to a :: b

      …

!$xmp template_fix :: t(128)

      allocate (a(128))

      if (me < 4) b(4) = b(4)[me +1]
```

グローバル配列 **a** は割付け配列であるため、そのローカルエイリアス **b** は、当該手続きの実行開始時点では確定されていない。**allocate** 文において **a** が割付けられた時点で **b** は確定される。また、**b** は coarray として宣言されているので、coarray として定義･引用することができる。

3.5.3 Post 指示文と Wait 指示文

3.5.4 Critical 指示構文

4 関数呼び出し

5 ベース言語からの拡張

5.1 Coarray 記法

5.2 C 言語の添字三つ組

6 組み込み関数

xmp_get_node_num()

xmp_get_num_nodes()

チーム内の番号、数

6.1 ローカルビュー実行指示文

6.2 グローバル-ローカルビュー連携指示文

[Summary]
分散されたグローバル配列のローカルエイリアスを宣言する。

[Syntax]
[F]
!$xmp local_alias *local-array-name* **to** *global-array-name*

[C]
#pragma xmp local_alias *local-array-name* **to** *global-array-name*

- 制約: *local-array-name* で指定される配列は、**align** 指示文によって整列されていてはならない。
- 制約: *global-array-name* で指定される配列は、**align** 指示文によって整列されていなければならない。
- 制約: *local-array-name* で指定される配列と、*global-array-name* で指定される配列のデータ型および次元数は一致していなければならない。
- [F]制約: *local-array-name* で指定される配列は、形状引継ぎ配列でなければならない。
- [C]制約: /*ocal-array-name* で指定される配列は、？？？？
  - コメント: C で「割付け配列」のようなものをどのように表現するか？ ポインタ？

[Description]

local-array-name で指定されるローカル配列が、*global-array-name* で指定されるグローバル配列のローカルエイリアスであることを宣言する。

ローカルエイリアスの形状は、元のグローバル配列に対応して自ノードに割付けられたローカル配列のそれに一致する。ここで、ローカル配列はシャドウ領域を含むことに注意されたい。

ローカルエイリアスが確定されるのは、対応するグローバル配列が確定される時点である。すなわち、グローバル配列が静的に割付けられている場合、有効域内でローカルエイリアスは常に確定

されている。グローバル配列が動的に割付けられる場合、それが割付けられた時点でローカルエイリアスは確定される。

local-array-name で指定される配列は coarray であってもよいことに注意されたい。

- 例 1

```
!$xmp nodes n(4)
!$xmp template :: t (100)
!$xmp distribute (cyclic) onto n :: t

      real :: a (100)
!$xmp align (i) with t(i) :: a
!$xmp shadow (1) :: a

      real :: b(0:)
!$xmp local_alias to a :: b
```

ノード **n(2)** には、25 個の要素から成るローカル配列(**a(2:100:4)**)と両端に幅 1 のシャドウ領域が割付けられている。また、ローカルエイリアス **b** の下限は 0 と宣言されている。したがって、**n(2)** において、**b** は、27 個の要素から成る配列(**b(0:26)**)となる。このとき、**a** の要素と **b** の要素は次のように対応する。

a	**b**
下シャドウ	0
2	1
6	2
10	3
…	…
98	25
上シャドウ	26

- 例 2

```
!$xmp nodes n(4)
!$xmp template :: t(:)
!$xmp distribute (block) onto n :: t

      real , allocatable :: a(:)
!$xmp align (i) with t(i) :: a

      real :: b(:)[*]
!$xmp local_alias to a :: b

      …

!$xmp template_fix :: t(128)

      allocate (a(128))

      if (me < 4) b(4) = b(4)[me +1]
```

グローバル配列 **a** は割付け配列であるため、そのローカルエイリアス **b** は、当該手続きの実行開始時点では確定されていない。**allocate** 文において **a** が割付けられた時点で **b** は確定される。また、**b** は coarray として宣言されているので、coarray として定義･引用することができる。

7 関数呼び出し

8 ベース言語からの拡張

8.1 Coarray 記法

8.2 C 言語の添字三つ組

9 組み込み関数

xmp_get_node_num()

xmp_get_num_nodes()

チーム内の番号、数

10 ハイブリッドプログラミング

10.1 MPI

10.2 OpenMP

11 数値計算ライブラリとの結合

11.1 ScaLAPACK