データ並列言語における集団通信の実行時認識手法

小笠原 武史† 小松 秀昭†

分散メモリ並列マシン上においてデータ並列プログラムの通信オーバヘッド削減には，集団通信の活用が非常に重要である．そのためにデータ並列言語の処理系は集団通信を認識すべきである．しかしデータ並列言語のユーザプログラムでは，配列やプロセッサ配列の形状等，コンパイラによる通信計算に必要な値がコンパイル時に不定である場合が多い．こうした値を不定のままコンパイラが通信を計算する場合，最適でない通信ライブラリを選択する可能性がある．本稿は，均質問題の並列化ループ内で参照される配列に対する通信計算のうち，コンパイル時に不定の値を使う部分を実行時に行うとともに実行時に集団通信を認識する手法を提案する．

A Method for Runtime Recognition of Collective Communication in Data-parallel Languages

TAKESHI OGASAWARA† and HIDEAKI KOMATSU†

Using collective communication libraries is crucial to reducing the communication overhead of data-parallel programs on distributed memory multiprocessors. Compilers of data-parallel languages and their runtime libraries should recognize collective communication patterns. However, in many user programs written in data-parallel languages, values of some parameters that are required for compilers to calculate communications are not determined at compile time. If such parameters are handled as variables and communications are calculated at compile time including those variables, compilers do not always select appropriate communication primitives. In this paper, we present a method to calculate communications for parallelized loops in regular problems and recognize patterns of collective communication at runtime.

1. はじめに

分散メモリ並列マシンにおけるデータ並列言語では，配列はあるプロセッサ形式に分配される．たとえばHigh Performance Fortran (HPF)[1]では，配列の各次元が `block(n)` や `cyclic(n)` で指定される方式で分割され，プロセッサ配列に分配される．データ並列言語のコンパイラは，プログラムを並列化し，通信を計算し，通信コードを挿入したコードを生成する．各プロセッサは，所有していない配列領域を参照する場合には，通信によって他プロセッサから当該配列領域を読み込む．

一般に，通信によるデータの読み出しはローカルメモリ上のデータへのアクセスよりも非常に遅いため，性能向上のためには通信コストを最小化することが重要である．データ並列言語のコンパイラにとって重要なことは，並列システムで提供される集団通信(collective communication)ライブラリ[2]を活用することにある[3]．集団通信ライブラリは，並列システムのハードウェア特性を反映して最適な通信を行う．

一方，HPFのようなデータ並列言語で書かれるユーザプログラムには，異なる計算機構成に対応するために，配列を分割するプロセッサ配列の形状をコンパイル時には可変にしておく場合がよくある．たとえば，ユーザプログラムは実行時に取得する計算機のプロセッサ数 n を，d 次元プロセッサ配列の各次元に $\sqrt[d]{n}$ 個ずつ分配してプロセッサ配列を構成する．

このような場合，コンパイラはコンパイル時に配列を分割するプロセッサの形状を知ることができない．そのためコンパイラは，プロセッサ配列各次元に複数のプロセッサがあるという一般的な仮定をするしかない．この仮定の下に，コンパイル時にプロセッサの形状を使って集団通信を認識する既存方式[3),4)]を行っても，たとえば実際には分割されていない配列次元を分割されたと仮定する等の影響等で，最適ではない通信ライブラリを選ぶ可能性がある．

さらにデータ並列言語のユーザプログラムでは，コ

† 日本アイ・ビー・エム東京基礎研究所
Tokyo Research Laboratory, IBM Japan

ンパイル時に配列の形状，ループの繰返し空間，配列の分配方式，配列の添字式の係数等，値がコンパイル時に決定しないものが多い．コンパイル時に不定な値を含む計算について，値を変数のままにしておき，数式処理システムを活用して通信を計算する手法[5]がある．しかし，こうした手法の場合，通信はプロセッサIDを変数とした式で求まる．そのため集団通信を認識するためには，式の値が求まる実行時に全プロセッサに関する通信を求め，さらに全通信を調べてパターンを見つけなければならず，実行時のオーバヘッドを考慮すると現実的でない．

また，サブルーチンや組込関数の境界で暗黙に行われる配列再分配においても，配列やプロセッサの形状はコンパイル時に決定しない場合が多いため，コンパイル時に通信の計算と通信ライブラリの選択を行うと，最適ではない通信ライブラリを選ぶ可能性がある．

このようにデータ並列言語のプログラムでは，通信計算に必要な値がコンパイル時に不定であることが多く，コンパイル時に最適でない通信ライブラリを選ぶ可能性がある．そのため我々は，そうした不定の値が決定される実行時まで通信ライブラリの選択を遅らせ，実行時に通信ライブラリを選べる機能を持つことがデータ並列言語処理系に必須と考える．そこで本稿では，通信の計算と通信ライブラリの選択において，コンパイル時に行える計算はコンパイル時に行い，コンパイル時に不定の値を使う計算だけを実行時に行う枠組みを提案する．本方式では，既存方式でコンパイル時に認識できた集団通信はコンパイル時に認識できる一方，前述のようにプロセッサ配列の形状がコンパイル時に不定である場合等コンパイル時に通信の計算が完結しない場合には，実行時に効率良く通信を計算し集団通信を認識する．

本稿ではこれより，データ並列プログラムで頻繁に登場する，ループ本体が配列をオペランドとする代入文であり，配列の添字式がループインデックス変数の線形式で表される，いわゆる均質問題（regular problems）の並列化ループを想定して，右辺配列の通信について議論を進める．配列再分配のための通信に関しては，並列化ループでの議論に帰着できる．すなわち，ループ本体の代入文の右辺と左辺を，それぞれ再分配の前と後の配列分配を持つ配列とすればよい．

本稿の構成は以下のとおりである．まず次節で関連研究について議論する．3章で本方式の動作概要と特徴を説明する．4章で本方式で使用するデータ構造を定義し，5章でアルゴリズムを説明する．6章でHPFコンパイラ[6]における実現例を使った評価を示す．最後に結論を述べる．

2. 関連研究

データ並列プログラムで発生する通信において集団通信を活用しようとする従来研究は，集団通信をコンパイル時に認識するか実行時に認識するかの2つに分類される．前者は，ユーザプログラム中の代入文の左辺と右辺の配列において，添字式やプロセッサへの分配の仕方の関係から，コンパイル時に集団通信を認識する方式[3),4)]である．しかしながら，この方式は，コンパイル時に配列分配方式が決定されることを前提にしており，本稿で注目する，プロセッサ配列の形状が実行時まで決まらない場合，最適な通信ライブラリを選べない場合がある．

一方後者は，配列の動的再分配に必要な通信を実行時に計算し集団通信を認識する方式である．Kalnsらは，配列の再分配において集団通信のうち，分散（scatter），集結（gather），全対全（all to all）のみを認識する方式を提示している[7]．言い換えれば，配列要素がプロセッサ間一対一で通信されるパターンのみを認識し，一対多通信である放送（broadcast）と連結（all gather）を認識できない．Kalnsらの方式は，配列各次元の各要素ごとに通信前の所有プロセッサと通信後の所有プロセッサを計算することによって通信を計算する．したがって，通信の計算は，配列各次元の要素数の和のオーダーの計算量を持ち，配列要素数の大きい数値計算では実用にならない，という問題も持つ．

3. 動作概要と特徴

本方式は図1に示した手順で通信の計算と集団通信の認識を行う．それぞれの手順の計算対象について説明する．計算方法については5章で説明する．手順(1)のarray ownership set（AOS）は，プログラムが指定した，配列各次元の分割とプロセッサ配列の次元への対応付けを表す．手順(2)のlocal iteration set（LIS）は，左辺の所有者が右辺を計算する，いわゆる所有者計算方針によってループの繰返し空間をプロセッサに分配したものである[8]．手順(3)のlocal

(1) array ownership set（AOS）作成
(2) local iteration set（LIS）作成
(3) local read set（LRS）作成
(4) in/out set（IOS）作成
(5) IOSからcommunication descriptor（CD）の抽出と集団通信の認識

図1 本方式の計算の流れ
Fig. 1 The procedures of our algorithm.

read set（LRS）は，プロセッサ各々がLISに基づいた繰返し空間でループを実行する際に右辺で参照する配列領域を表す．手順(4)のin/out set（IOS）は，in setの場合はプロセッサ各々がLRSで参照する右辺の配列領域をどのプロセッサから受信するかを表す，全プロセッサに関する集合である．out setの場合はその逆で，プロセッサ各々が所有する配列領域をどのプロセッサに送信するかを表す，全プロセッサに関する集合である．手順(5)のcommunication descriptor（CD）は，通信される配列領域とその送信・受信プロセッサの組で，通信を指定する．プロセッサは全プロセッサの通信の集合であるIOSからプロセッサ固有の通信であるCDを抽出する．

ループ本体の左辺と右辺の添字式，配列の分配方式，ループの繰返し空間を使って，ループ実行で発生する通信の計算を行うという点では，手順(1)から(4)までの手順と同等な計算はデータ並列言語の処理系には基本的なものであり，たとえば文献9)の既存コンパイラでも行われている．

集団通信の認識における本方式の新規な点は，従来は通信をプロセッサ単位で計算していたのに対し，通信を表すCDは同じ通信を行うプロセッサをグループ単位で扱っている点である．手順(5)においてあるプロセッサがIOSからCDを抽出する際，そのプロセッサと同じ通信を行うプロセッサグループのCDを抽出する．プロセッサグループが1プロセッサのみを含み，抽出したCDが1プロセッサの通信を表す場合もあるが，もし複数プロセッサの通信を表す場合，集団通信のうち複数プロセッサが同じデータを1プロセッサから受信する放送通信あるいは連結通信をCDから認識できる．このような同じデータを一対多通信する集団通信は，文献7)で認識できなかった．

我々はこのような“同じ通信を行うプロセッサグループ”をCDで表現するために，手順(1)から(4)までの手順において**3つ組プロセッサ形式**を導入した．3つ組プロセッサ形式はプロセッサ群を，同じグループに属するプロセッサは同じ通信を行うような，複数のグループに分ける．手順(1)から(4)まで，配列各次元の分割された配列区間や各ループインデックスの繰返し区間等と対応づけられるプロセッサの表現には，すべてこの3つ組プロセッサ形式が使われる．たとえば3つ組プロセッサ形式上の同じプロセッサグループに属するプロセッサは，AOSに関して同じ配列区間を所有し，LISに関して同じ繰返し区間を実行する．

また，本方式では手順(1)から(4)まで，分割された配列区間とプロセッサの対応，あるいはループ繰返し空間とプロセッサの対応を表現するために，プロセッサ間の規則性を活かした表現〔**拡大表現**（augmented expression）と呼ぶ〕を使っている[9]．たとえばプロセッサ1が配列区間1:25を，プロセッサ2が配列区間26:50を，プロセッサ3が配列区間51:75を，プロセッサ4が配列区間76:100を参照するとき，これらの参照を拡大表現$[1{:}25]_{proc=1...4}$として簡潔に表現する．拡大表現を使うことには，表現の簡潔さばかりでなく，計算量を削減するメリットがある．手順(1)から(4)までの計算を拡大表現だけで行える場合，計算量はプロセッサ数や配列要素数に関わらない．この場合，本方式は大規模数値計算においても効率的に動作する．

集団通信の認識における本方式の新規なもう1つの点は，拡大表現を集団通信認識に利用したことである．文献9)でもコンパイルに`block`分割の規則性を活かした拡大表現を用いているが，集団通信の認識には活用していない．集団通信のうち分散，集合，全対全は，連続する配列区間を規則的に分割し，各々を一対一通信するパターンである．CDが拡大表現で求まる場合，これら3種類の集団通信を配列次元数のオーダーで認識できる．文献7)では認識に配列要素数のオーダー必要であった．

ここで，本方式で集団通信と判定されず一対一通信ライブラリが使われる通信についても述べておく．一般に一対一通信を最適化するためには，一度全通信を求めて通信のスケジュールを行う必要がある[10]．本方式では，IOSからの全通信の抽出コスト，全通信を格納するメモリ量，通信スケジュールコストを費やすよりも，非ブロック（nonblocking）通信ライブラリを使ってネットワーク資源の有効活用を行う方針をとった．

また本手法の実行時オーバヘッドを削減するために，本手法の同じ計算が何度も行われる場合に計算結果を再利用している[12]．再利用によって同じ通信が繰り返し行われるときにオーバヘッドが削減される．また，コンパイル時に決定している情報だけで計算できる通信セットの一部をコンパイル時に作成するハイブリッドなアプローチ[14]をとっている．

4. データ構造の定義

本章では3章で本方式の特徴として述べた，同じ通信を行うプロセッサどうしでグループ化された，プロセッサ群の表現形式である3つ組プロセッサ形式と，3つ組プロセッサ形式上のプロセッサグループを指定する**4つ組プロセッサ指定**を説明する．次に3つ組プ

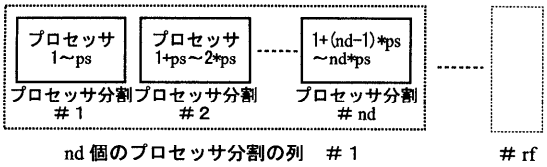


図2 3つ組プロセッサ形式
Fig. 2 Processor specification with triplet.

ロセッサ形式上のプロセッサグループと配列区間あるいはループ繰返し区間との対応付けを表現するITRリストを定義する．ITRリストは可能な限り拡大表現で表される．

4.1 3つ組プロセッサ形式と4つ組プロセッサ指定

複数プロセッサのグループへの分割を指定する3つ組プロセッサ形式（以下，3つ組形式）を定義する．3つ組形式はプロセッサ跳幅（ps），プロセッサ分割数（nd），プロセッサ合計数（np）の3要素から構成され，$ps:nd:np$で表す．これらの要素の意味は次のとおりである．まずプロセッサ合計数np個のプロセッサ列$1,2,\ldots,np$を考える．次にこのプロセッサ列をps個ずつグループ化する．グループそれぞれをプロセッサ分割と呼ぶ．さらにプロセッサ分割をnd個ずつグループ化する．その結果3つ組形式$ps:nd:np$は，ps個のプロセッサを表すプロセッサ分割のnd個の列が$rf=np/ps/nd$回繰り返すことを表す．**図2**は，3つ組形式によるプロセッサのグループ化を表す．

また，3つ組形式上でプロセッサ分割の位置を指定した，4つ組プロセッサ指定（以下，4つ組指定）を定義する．3つ組形式$ps:nd:np$で指定されるプロセッサ分割の列上において，dx番目のプロセッサ分割を指定する4つ組指定を$dx:ps:nd:np$で表す．dxをプロセッサ分割位置と呼ぶ．先に述べたようにプロセッサ分割の列はrf回繰り返すが，4つ組指定はそうした繰返しを含んだすべての列上のdx番目のプロセッサ分割を指定する．

4.2 ITRリスト

3つ組形式の各プロセッサ分割位置と，配列区間およびその配列区間の通信相手のプロセッサを対応付けるのが**ITRリスト**である．ITRリストは図1における，AOSからIOSの計算に至るまですべてに使用される基本データ構造である．ITRリストは3つ組形式である**ITRマスタ**（$\langle mps:mnd:mnp\rangle$で表す）と配列区間やループ繰返し区間を表す**ITRブロック**の列から構成される．ITRマスタの各プロセッサ分割はITRブロックと対応付いている．

ITRマスタはプロセッサp $(p=1,2,\ldots)$をITRリスト上のITRブロックに対応付けるために使われる．対応付けるにはまず，次の式によってITRマスタ上のpを含むプロセッサ分割の位置を求める．

$$mdx = 1 + mod((p-1)/mps, mnd)) \quad (1)$$

$mdx:mps:mnd:mnp$を**通信元4つ組指定**と呼ぶ．ITRリストのmdx番目のITRブロックがプロセッサpに対応したITRブロックである．

ITRブロックは**ITR**の集合である．ITRは，開始(bg):終了(ed):跳び幅(st)で指定される区間R_{ITR}と，その区間の通信相手を指定する**通信先4つ組指定**$Q_{ITR}=dx:ps:nd:np$（$ps:nd:np$を**通信先3つ組形式**と呼ぶ）とから構成され，

$$ITR=[R_{ITR},Q_{ITR}] = [bg:ed:st,dx:ps:nd:np] \quad (2)$$

と表される．IOS以外のITRでは通信相手を指定する必要がないので，Q_{ITR}はϕで表す．

ITRリスト$ITRL$をITRマスタとITRブロックの列を使って次のように表記する．

$$ITRL=\langle mps:mnd:mnp\rangle \{[bg:ed:st,dx:ps:nd:np]\} \quad (3)$$

ITRリスト上，ITRマスタのi番目のプロセッサ分割と，ITRブロック列上のi番目のITRブロックとが対応する．言い換えれば，i番目のプロセッサ分割に含まれる一般には複数のプロセッサは，AOSの場合は同じ配列区間を所有し，LISの場合は同じ繰返し空間を実行し，LRSの場合は同じ配列区間を読み出し，IOSの場合は同じ配列区間を通信する．

5. アルゴリズム

本章では，図1に示した手順における計算方法を説明する．手順(1)から(4)のうち，コンパイル時に計算できる部分についてはコンパイル時に行い，実行時オーバヘッドを削減する．手順(5)について，本稿ではsingle program multiple data（SPMD）コード[11]を生成するため，生成コード量を抑える意味で，一般にはCDを全プロセッサに対してコンパイル時に抽出しない．しかしながらコンパイル時に集団通信を認識できた場合，全プロセッサの通信コードを生成しても問題ない場合，コンパイル時にCDを抽出し通信ライブラリを選択する．また実行時に計算する手順でも，計算結果の再利用によって実行時オーバヘッドを削減する．

本章では計算方法を示すと同時に，具体的な例を使って動作を模式的に示す．使用する例は，2つの連結通信が同時に起きるプログラムである（**図3**参照）．図3は，取り扱うループと配列の分配方式を表している．左辺b(i,j)と右辺a(k,j)の配列はともに，100×100

```
配列宣言： real a(100,100), b(100,100)
配列分配方式： $hpf distribute (block,block) onto p(4,2) :: a,b
プログラム：
      do i=1,100
        do j=1,100
          do k=1,100
            b(i,j)=...a(k,j)...
          enddo
        enddo
      enddo
```

図3 2つの連結通信が起きるプログラム例
Fig. 3 An example in which two all gather communications occur.

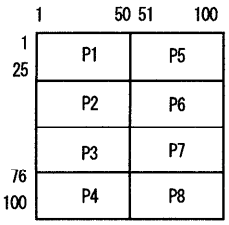


図4 配列a，bのプロセッサへの分配
Fig. 4 The distribution of arrays to processors.

で宣言され，プロセッサ配列 p(4,2) に (block,block) 方式で分配されている．ここではこの分配が，ループのコンパイル時には分からず，実行時に指定される場合を想定する．**図4** は本例で，プロセッサ配列 p(4,2) 中のプロセッサ P1（p(1,1) で指定される）から P8（p(4,2) で指定される）に，配列各領域が分配された様子を示す．本ループに関して，右辺 a(k,j) のループ繰返し間データ依存はなく，a(k,j) の通信はループ前で一括して行える．

以下図1の手順に沿って詳しく述べる．

5.1 AOS 作成

AOS は，プログラムが指定した，配列各次元の分割とプロセッサ配列の次元への対応付けを表す．

pd 次元のプロセッサ形式 $P(m_1,\ldots,m_{pd})$ と，ad 次元の配列 $A(n_1,\ldots,n_{ad})$ とのマッピングを行う．HPF の場合，P はコンパイラ指示子 `processors` で指定される配列である．A の i 次元目 A_i は P の j 次元目 P_j で分割されるか，まったく分割されない（collapsed）．分割を行わない P の次元について複製化（replicated）が起きる．

A_i が P_j で分割されている場合，AOS の i 次元目は ITR リスト $ITRL_{AOS_i^A}$ で表現され，プロセッサ数を $np=\prod_{l=1}^{pd} m_l$，mdx 番目の ITR の配列区間を $R_{mdx}^{A,i}$ とすると次のように書ける．ITR の4つ組指定は使われない．

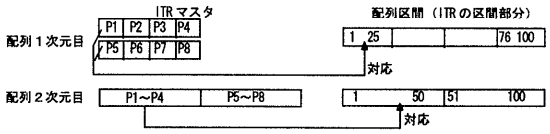


図5 配列a，bのAOS
Fig. 5 The AOS of arrays.

$$M^{P,j} \equiv \prod_{l=1}^{j-1} m_l : m_j : np$$

$$ITRL_{AOS_i^A} = \langle M^{P,j} \rangle \{[R_{mdx}^{A,i}, \emptyset]\} \qquad (4)$$

$R_{mdx}^{A,i}$ は分配方式に応じて求まる．HPF の場合，$R_{mdx}^{A,i}$ は，`block` 分配方式ならば，

$$b = \lceil n_i/m_j \rceil$$
$$R_{mdx}^{A,i} = b*mdx - b + 1 : b*mdx : 1$$

また `cyclic` 分配方式ならば，

$$R_{mdx}^{A,i} = mdx : n_i : m_j$$

で与えられる．本稿では1より大きいブロックサイズによる `cyclic`，いわゆる `block-cyclic` 分配は，ブロックサイズ倍の仮想プロセッサを割り当ててブロックサイズが1の `cyclic` として扱う，virtual processor 方式[13]で対応している．

5.1.1 例：配列a，bのAOS作成

図3における配列aとbに対して，AOSの配列各次元ごとのITRリストを**図5**に模式的に示す．図5の左側はITRマスタが表すプロセッサ群で，各プロセッサ分割はセルで示されている．セル中のプロセッサIDは，各プロセッサ分割に属するプロセッサグループを表す．また右側は配列区間を表すITRの列で，各配列区間はセルで示されている．左側のセルと右側のセルは，プロセッサ分割位置ごとに対応している．図5では一部の対応を矢印で表した．

図において，たとえば配列1次元目の1番目のプロセッサ分割位置において，プロセッサP1とP5は配列区間1:25（1から25まで）を所有し，配列2次元目の1番目のプロセッサ分割位置において，プロセッサP1からP4は配列区間1:50を所有することを表している．本例では，実行時に与えられる，配列サイズ 100×100，(block,block) 分割，プロセッサ配列 p(4,2) とからこのAOSが実行時に作成される．

本章ではこれ以降の模式図でも，プロセッサ分割や配列区間等をセルで表す．同じプロセッサ分割位置でのITRマスタのプロセッサ分割とITRとの対応の矢印は基本的に省略するが，説明を要する部分には適宜使用する．また，本例においてITR列はすべて拡大表現が可能であるが，図では展開して表示した．

5.2 LIS 作成

LISは，左辺の所有者が右辺を計算する，いわゆる所有者計算方針によってループの繰返し空間をプロセッサに分配したものである．

プロセッサに分配される前の多重ループの繰返し空間をglobal iteration set（GIS）と呼ぶ．GISのd重目GIS_dは左辺Aの配列次元と対応の有無で分類する．対応があるとは，対応する配列次元の添字式がGIS_dのインデックス変数iv_dを用いて表せることである．対応がない場合はそのままLISを構成する．

GIS_dと配列のil次元目が対応している場合，GIS_dは所有者計算方針に基づき分割されてLIS_dとなる．ここで，ループの下限，上限，跳び幅をそれぞれlb_d, ub_d, by_d，配列の添字式を$S_{il}(iv_d)$とすると，GIS_dにおいて各プロセッサが更新する配列領域（これをlocal write setまたはLWSと呼ぶ）は，$S_{il}\cdot GIS_d$と$ITRL_{AOS^A_{il}}$上の各ITRの配列区間の交わりによって求めることができる．P^lは左辺が分配されるプロセッサ形式，jlは配列次元ilが対応するP^lの次元である．

$$RLWS^{il,d}_{mdx} \equiv \{S_{il}(iv_d)\}_{iv_d=lb_d:ub_d:by_d} \cap R^{A,il}_{mdx}$$
$$ITRL_{LWS^A_{il}} = \langle M^{P^l,jl}\rangle\{[RLWS^{il,d}_{mdx},\emptyset]\} \quad (5)$$

LIS_dは，$ITRL_{LWS_{il}}$上の各ITRの$RLWS^{il,d}_{mdx}$にS^{-1}_{il}を作用させることによって，LWSの各配列添字のGIS上の対応を計算して求められる．

$$RLIS^{il,d}_{mdx} \equiv S^{-1}_{il}(RLWS^{il,d}_{mdx})$$
$$ITRL_{LIS_d} = \langle M^{P^l,jl}\rangle\{[RLIS^{il,d}_{mdx},\emptyset]\} \quad (6)$$

5.2.1 例：ループ i, j, k の LIS 作成

本例のLISを**図6**に示す．本例の左辺添字式は，iあるいはjといった逆関数も同じ自明な式なため，LISのうちインデックス変数iとjについてはAOSと同じである．インデックス変数kについては左辺添字式に含まれず繰返し区間は分配されない．

5.3 LRS 作成

LRSは，プロセッサ各々がLISに基づいた繰返し空間でループを実行する際に右辺で参照する配列領域を表す．

右辺Bのir次元目B_{ir}はGISと対応があるかないかに分けられる．対応がない場合はそのままLRSを構成する．

Bのir次元B_{ir}とループd重目GIS_d/LIS_dとが対応する場合を考える．B_{ir}の添字式を$T_{ir}(iv_d)$とする．LRSは$T_{ir}\cdot LIS_d$によって求められる．LRSは次の式で求まる．

$$RLRS^{ir,il,d}_{mdx} \equiv T_{ir}(RLIS^{il,d}_{mdx})$$
$$ITRL_{LRS^B_{ir}} = \langle M^{P^l,jl}\rangle\{[RLRS^{ir,il,d}_{mdx},\emptyset]\} \quad (7)$$

5.3.1 例：右辺 a(k,j) の LRS 作成

本例のLRSを**図7**に示す．本例では右辺a(k,j)の添字式は両次元共にインデックス変数だけからなる式である．そのため右辺a(k,j)のLRSは，1次元目はLISのインデックスk，2次元目はLISのインデックスjの部分と同じになる．

5.4 IOS 作成

LRSとAOSを用いて通信を表すIOSを計算する．IOSは全プロセッサの通信をITRリストで表現する．

まず2つのITRリストの**ITR分割**（$\tilde{/}$で表す）を定義する．ITR分割は被除数であるITRリストのITRを，除数であるITRリストのITRで分割する．分割結果のITRには，配列区間は被除数ITRの配列区間R_s，除数ITRの配列区間R_dとすると$R_s\cap R_d$，4つ組指定は（除数ITRの分割位置：除数ITRマスタ）が入る．$R_s\cap R_d\neq\emptyset$であるすべてのR_dについて行われる．

今，P^rを右辺が分配されるプロセッサ形式，jrは配列次元irが対応するP^rの次元とすると，$ITRL_{AOS^B_{ir}}$は

$$ITRL_{AOS^B_{ir}} = \langle M^{P^r,jr}\rangle\{[R^{B,ir}_{mdx},\emptyset]\} \quad (8)$$

である．IOSは式(7)と式(8)で示される2つのITRリストのITR分割をとることによって求まる．

$$ITRL_{IS^B_{ir}} = ITRL_{LRS^B_{ir}}\tilde{/}ITRL_{AOS^B_{ir}} \quad (9)$$
$$ITRL_{OS^B_{ir}} = ITRL_{AOS^B_{ir}}\tilde{/}ITRL_{LRS^B_{ir}} \quad (10)$$

これらをまとめて**in/out set（IOS）**と呼ぶ．

これら演算の意味は次のとおりである．LRSのITRリストをAOSのITRリストでITR分割すれば，結果のITRリストは各プロセッサが右辺で読み出す配列領域のそれぞれをどのプロセッサが所有するかを記述

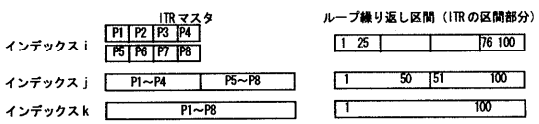


図6 ループ i, j, k の LIS
Fig. 6 The LIS of loop index variables.

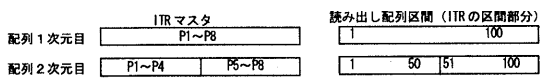


図7 右辺 a(k,j) の LRS
Fig. 7 The LRS of the array reference a(k,j).

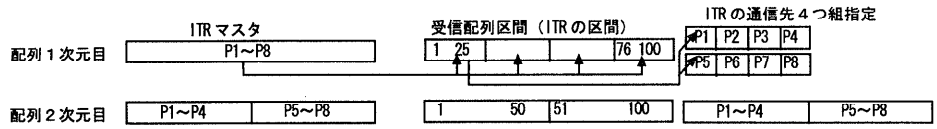


図8 右辺 a(k,j) の in set
Fig. 8 The in set of the array reference a(k,j).

する in set（$ITRL_{IS}$）が求まる．また AOS の ITR リストを LRS の ITR リストで ITR 分割すれば，結果の ITR リストは各プロセッサが所有する配列領域のうち読まれる領域と読むプロセッサを記述する out set（$ITRL_{OS}$）が求まる．分割されていない配列次元の場合は，ITR の 4 つ組指定にはいる分割位置には分割がないことを示す 0 を入れる．

5.4.1 例：右辺 a(k,j) の in set 作成

本例では in set のみ説明する．いま図 7 と図 5 で示す LRS と AOS の 1 次元目に注目する．LRS の読み出し配列区間を AOS の配列区間と同じように分割し，AOS の分割された配列区間と対応付ける．結果を図 **8** に示す．図 8 の配列 1 次元目において，プロセッサ P1 から P8 は，4 つの配列区間 1:25，26:50，51:75，76:100 を，それぞれプロセッサ P1 と P5，P2 と P6，P3 と P7，P4 と P8 より受信する．配列 2 次元目については，LRS の読み出し区間と AOS の配列区間は同じ分割であるため分割なしに対応し，プロセッサ P1 から P4 は配列区間 1:50 をプロセッサ P1 から P4 より，プロセッサ P5 から P8 は配列区間 51:100 をプロセッサ P5 から P8 より受信する．

5.5 CD 抽出と集団通信認識

本節では，全プロセッサの通信を表現する IOS からプロセッサ固有の通信である CD を抽出し，CD から集団通信を認識する方法を説明する．

5.5.1 CD 抽出

IOS が全プロセッサの通信を表すのに対し，CD はプロセッサ固有の通信を表し，IOS から抽出される．プロセッサ p に対する CD を抽出することを考える．まず IOS の各次元において，プロセッサ p に対する通信として，式 (1) で求まるプロセッサ分割位置にある，ITR マスタのプロセッサ分割と ITR ブロックを選択する．さらに，選択されたプロセッサ分割と ITR ブロックは配列 1 次元分の情報であるため，以下の手続きによってすべての次元で組み合わせ，1 つの配列領域とその通信元・通信先プロセッサで構成される CD を作成する．

d 次元の配列の場合，i 次元目で式 (1) で求まる ITR マスタのプロセッサ分割位置を 4 つ組指定 $mdx_i : M_{ITRL_i}$ とし，また mdx_i 番目の ITR ブロックが含む n_i $(1 \le i \le d)$ 個の ITR のそれぞれを $ITR^i_{k_i}$ $(1 \le k_i \le n_i)$ とし，ITR ブロックを $\{ITR^i_{k_i}\}_{k_i=1..n_i}$ とする．さらに，$ITR^i_{k_i}$ の 2 つの構成要素である配列区間と通信先 4 つ組指定をそれぞれ，$R^i_{k_i}$，$Q^i_{k_i}$ とする．これらを用いて全次元通じて組み合わせた結果を定義すると，通信元プロセッサは $\bigcap_{i=1}^{d} mdx_i : M_{ITRL_i}$，通信する配列領域は $(R^1_{k_1}, \ldots, R^d_{k_d})$，通信先プロセッサは $\bigcap_{i=1}^{d} Q^i_{k_i}$ となる．$\bigcap_{i=1}^{d}$ は，4 つ組指定をプロセッサの集合と見なしたときの積集合である．一般には同じ手続きを $k_1 = 1 \ldots n_1, \ldots, k_d = 1 \ldots n_d$ について行う．

5.5.2 例：CD 抽出

本例ではプロセッサ P1 を例にとって説明する．まずプロセッサ P1 は P1 を含む in set のプロセッサ分割位置を求める．式 (1) により in set の両次元ともに 1 である．

図 9 の横線の上側は図 8 の in set である．図 9 中，ITR マスタ中の黒箱はプロセッサ P1 を表す．図 9 中，図 8 の in set においてプロセッサ分割位置 1 に対応した ITR マスタ中のプロセッサ分割，ITR の受信配列区間と通信先 4 つ組指定が灰色で表示されている．

次にこれらプロセッサ分割位置 1 上の情報を配列全次元を通して組み合わせる．図 9 の横線の下側に，組み合わせた結果である，プロセッサ P1 の CD を示す．

5.6 集団通信認識

本節では CD からいかに集団通信（放送，連結，集結・分散，全対全，シフト）を検出するかを説明する．

CD が持つ，集団通信を認識するうえでの特徴は以下の 2 つである．

(1) CD の通信元プロセッサが複数のプロセッサを表す場合，複数のプロセッサが同じ通信を行うことが分かる．

(2) CD の通信配列領域が拡大表現されている場合，互いに規則的に異なる配列領域を通信することが分かる．

特徴 (1) は放送通信と連結通信を，特徴 (2) は集結通信，分散通信，全対全通信を検出する．

放送 in set から求めた CD の通信元プロセッサが複数プロセッサを表し，通信する配列領域が 1 つ，つまり同一配列領域を受信するプロセッサが複数

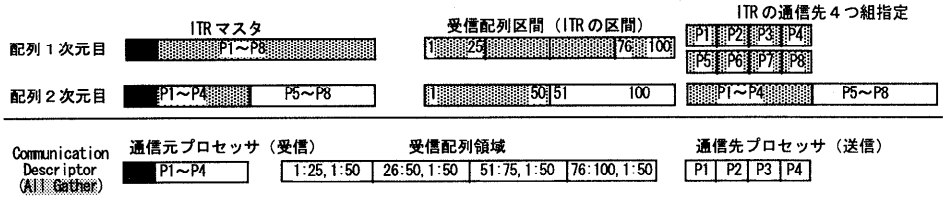


図9 右辺 a(k,j) のプロセッサ P1 の CD
Fig. 9 The CD of the array reference a(k,j) for processor P1.

ある場合，放送通信と認識する．

連結 連結通信は，あるプロセッサグループ内において，複数の異なる配列領域をそれぞれ異なるプロセッサから同時に放送通信していると見なすことができる．in set から求めた CD の通信元プロセッサが複数プロセッサを表し，しかも CD の通信先プロセッサと等しい場合，連結通信と認識する．

集結・分散 in set から求めた CD の通信元プロセッサが 1 プロセッサを表し，通信する配列領域とその通信先プロセッサとが拡大表現される場合，集結通信と認識する．逆に，in set から求めた CD の通信する配列領域が拡大表現され，通信先プロセッサが 1 プロセッサの場合，分散通信と認識する．

全対全 全対全通信は，あるプロセッサグループ内において，異なるプロセッサが同時に分散通信していると見なすことができる．in set から求めた CD の配列領域が拡大表現され，通信先プロセッサが通信元プロセッサと等しい場合，全対全通信と認識する．

シフト シフト通信は in set の形状から認識される．in set がすべて拡大表現され，ITR マスタと通信先 3 つ組形式が等しい場合，同じ 3 つ組形式上の異なるプロセッサ分割位置間の通信であり，シフト通信と認識する．

5.6.1 例：連結通信認識

図 9 の最後に示したプロセッサ P1 の CD において注目したいのは，受信プロセッサ集合にプロセッサ P1 から P4 が含まれることである．これは，プロセッサ P2 から P4 上でもプロセッサ分割位置が P1 と同じ 1 となり，図 8 の in set から P1 と同じ図 9 の CD を求めていることを意味する．つまり，プロセッサ P1 から P4 が同じ通信を行うことを意味する．プロセッサ P1（そして P2 から P4）はこの CD から次のことを知ることができる．すなわち，P1 から P4 の全プロセッサが等しく，配列領域 a(1:25,1:50), a(25:50,1:50), a(51:75,1:50), a(76:100,1:50) をそれぞれプロセッサ P1, P2, P3, P4 から受信する．この通信パターンは集団通信の一種である連結通信である．このように本方式では，通信を行うプロセッサを集合的に扱う，CD という形式で通信を求めることで集団通信を認識する．

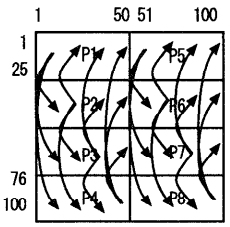


図10 右辺 a(k,j) の連結通信
Fig. 10 Communications of all gather for the array reference a(k,j).

同様にして，P5, P6, P7, P8 についても連結通信を認識し，結果として本例では同時に実行される 2 つの連結通信が認識される．**図 10** は，プロセッサ P1 から P4 までと，プロセッサ P5 から P8 までとの，同時に起きる 2 つの連結通信の様子を示す．図において矢印はデータの移動を表す．

5.7 複製化した配列における通信の最適化

CD の特性を活かして通信をさらに最適化することができる．配列領域をプロセッサ配列で複製化（replicate）し，同一配列領域を複数のプロセッサが所有する場合，in set から求めた CD の通信先プロセッサは複数プロセッサを表す．この場合 CD から送信プロセッサが複数存在することが分かるので，送信プロセッサを増やすことで放送通信におけるプロセッサ間同期を減らすことができる．

6. 効果の評価

本章では，配列とプロセッサの形状等がコンパイル時に不定であるとき 2 章で議論した従来研究では認識できない連結通信を，実際に本手法を HPF コンパイラに実装して認識する．そして，本手法の実行時オーバヘッドを考慮しても，一対一通信ライブラリの代わ

りに連結通信ライブラリを利用する価値があることを示す．

6.1 評価環境

我々は，本手法を我々のHPFコンパイラ[6]に実装した．このHPFコンパイラは中間言語上でSPMD化を行い，その後単一プロセッサを対象とした各種最適化を行う．生成されるSPMDコードは，MPIライブラリとともにIBM RS/6000 SP上で動作する．今回，RS/6000 SP（POWER2-66 MHzシン・ノード）上で評価を行った．

本実験では配列乗算を使って配列要素数に対する，5章で説明したアルゴリズムの計算時間と，一対一通信（point-to-point）ライブラリと連結通信ライブラリをそれぞれ使った場合の通信時間を測定する．本例の配列乗算では，サイズ $n \times n$ の `real*8` 配列 X，Y，Z が宣言され，それらの配列の分配方式 `(*,BLOCK)` が実行時に与えられる．本例では，配列乗算 $Z = XY$ を行うループ前で右辺 Y に関して行われる通信のパターンは，連結通信である．

6.2 アルゴリズムの実行時間と連結通信ライブラリによる通信時間の改善

図11 と**図12** はそれぞれ4個と8個のプロセッサを使った，5章で説明したアルゴリズムの実行時間と，一対一通信ライブラリと連結通信ライブラリをそれぞれ使った場合の通信時間を測った結果である．図中の縦軸は時間，横軸は配列の1次元あたりの要素数である．通信時間については縦軸の単位は秒だが，アルゴリズムの実行時間についてはミリ秒であり，後者については1,000倍して通信時間と同じグラフに載せていることに注意してほしい．

図11と図12が示すように，アルゴリズムの実行時間（4プロセッサで0.57ミリ秒，8プロセッサで0.73ミリ秒）に比べ，連結通信ライブラリが改善した通信時間の方がはるかに大きい．図において特に配列サイズ2,000を超えた時点で，4プロセッサでは0.52秒（約12%），8プロセッサで0.94秒（約21%），連結通信ライブラリで通信時間が大きく改善しており，プロセッサ数や配列サイズが大きいほど，アルゴリズムの実行オーバヘッドを費やしても，連結通信を認識し一対一通信ライブラリでなく連結通信ライブラリを選択する価値がある．

ここで本例におけるアルゴリズムの計算時間について議論する．アルゴリズムで処理されるデータは拡大表現され，配列サイズに依存しない計算時間になっている．プロセッサが4台と8台の場合の実行時オーバヘッドを比べて見られる計算時間の若干の増加は，通信準備のためのMPIライブラリ呼び出しに起因する．

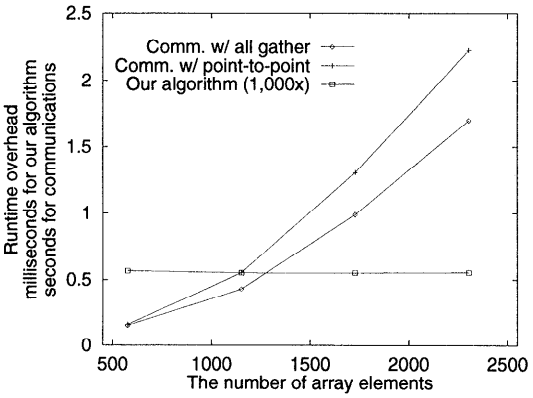


図11 アルゴリズム実行時間と通信時間（4プロセッサ）
Fig. 11 Algorithm and communication overhead (4-node).

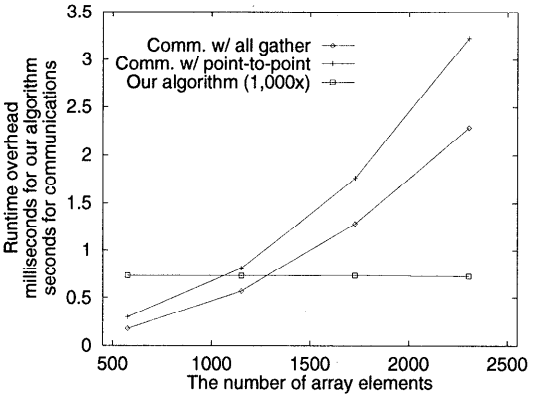


図12 アルゴリズム実行時間と通信時間（8プロセッサ）
Fig. 12 Algorithm and communication overhead (8-node).

7. まとめ

本稿では，データ並列言語のユーザプログラムで多く見られる，配列やプロセッサの形状，ループの繰返し空間，配列の添字式の係数等，通信の計算に必要な値がコンパイル時に不定である場合の通信計算と通信ライブラリ選択という問題に対して，コンパイル時に計算できない部分のみを実行時に行い，実行時に集団通信を認識する方式を提案した．本方式において通信計算に必要な値がコンパイル時にすべて分かる場合，コンパイル時にすべて計算して通信ライブラリを選択するため，実行時オーバヘッドはかからない．

本方式では実行時オーバヘッドを考慮し，プロセッサ群を同じ振舞いをするプロセッサごとにグループ化する3つ組プロセッサで表現し，配列区間やループ繰

返し区間の規則的な分割を拡大表現することで，実行時においても効率的に動作することが特徴である．

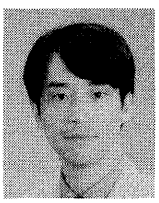
また本方式を HPF に実装し連結通信を例にとり，本方式による実行時オーバヘッドを費やして連結通信を認識しても，連結通信ライブラリが一対一通信ライブラリに対して改善する通信時間の方がはるかに大きいことを示した．

謝辞 本研究にあたり，日本アイ・ビー・エム東京基礎研究所の先進コンパイラグループの協力に感謝いたします．

参考文献

1) Koclbcl, C.H., Loveman, D.B., Schreiber, R.S., Steele Jr., G.L. and Zosel, M.E.: *The High Performance Fortran Handbook*, The MIT Press (1994).
2) McKinley, P.K., Tsai, Y.-J. and Robinson, D.F.: A Survey of Collective Communication in Wormhole-Routed Massively Parallel Computers, Technical Report, MSU-CPS-94-35, Department of Computer Science, Michigan State University (1994).
3) Li, J. and Chen, M.: Compiling Communication-Efficient Programs for Massively Parallel Machines, *IEEE Trans. Parallel and Distributed Systems*, Vol.2, No.3, pp.361–376 (1991).
4) Gupta, M. and Banerjee, P.: A Methodology for High-Level Synthesis of Communication on Multicomputers, *Proc. 6th ACM International Conference on Supercomputing*, Washington, D.C., pp.357–367 (1992).
5) Su, E., Lain, A., Ramaswamy, S., Palermo, D.J., Hodges IV, E.W. and Banerjee, P.: Advanced Compilation Techniques in the PARADIGM Compiler for Distributed-Memory Multicomputers, *Proc. 9th ACM International Conference on Supercomputing*, Barcelona, Spain, pp.424–433 (1995).
6) 郷田 修，大澤 暁，小松秀昭，菅沼俊夫，小笠原武史，石崎一明，中谷登志男：HPF コンパイラの実装と評価，情報処理学会研究報告，Vol.95, No.81, pp.115–120 (1995).
7) Kalns, E.T. and Ni, L.M.: DaReL: A Portable Data Rcdistribution Library for Distributed-Memory Machines, *Proc. Scalable Parallel Libraries Conference*, Mississippi State University, Mississippi, pp.78–87 (1994).
8) Midkiff, S.P.: Local Iteration Set Computation for Block-Cyclic Distributions, *Proc. the 1995 International Conference on Parallel Processing*, Boca Raton, FL, pp.II/77–84 (1995).
9) Hiranandani, S., Kennedy, K. and Tseng, C.-W.: Compiler Optimizations for Fortran D on MIMD Distributed-Memory Machines, *Proc. Supercomputing '91*, Albuquerque, NM, pp.86–100 (1991).
10) Ranka, S., Wang, J.-C. and Kumar, M.: Irregular Personalized Communication on Distributed Memory Machines, *Journal of Parallel and Distributed Computing*, Vol.25, No.1, pp.58–71 (1995).
11) Quinn, M. and Hatcher, P.: Data-Parallel Programming on Multicomputers, *IEEE Software*, Vol.7, No.5, pp.69–76 (1990).
12) 小笠原武史，石崎一明，小松秀昭：HPF における実行時の通信解析オーバヘッドの削減手法，情報処理学会研究報告，Vol.95, No.81, pp.109–114 (1995).
13) Chatterjee, S., Gilbert, J.R., Long, F.J.E., Schreiber, R. and Teng, S.-H.: Generating Local Address and Communication Sets for Data-Parallel Programs, *Proc. 4th ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming* (1993).
14) Gupta, M., Midkiff, S., Schonberg, E., Sweeney, P. and Wang, K.Y.: PTRAN II: A Compiler for High Performance Fortran, *Proc. 4th Workshop on Compilers for Parallel Computers*, Delft, Netherlands, pp.479–493 (1993).

(平成 10 年 6 月 17 日受付)
(平成 11 年 9 月 2 日採録)



小笠原武史（正会員）

1991 年東京大学大学院理学系研究科修士課程修了．同年，日本 IBM（株）入社．以来，東京基礎研究所において，HPF コンパイラ，Java JIT コンパイラ等の研究に従事．IEEE-CS，ACM 各会員．

小松 秀昭（正会員）

1985 年早稲田大学大学院理工学研究科修士課程修了．同年，日本 IBM（株）入社．以来，東京基礎研究所において，Prolog コンパイラ，HPF コンパイラ，Java JIT コンパイラ等の研究に従事．また，早稲田大学の特別研究員として，命令レベル並列アーキテクチャ，コンパイラの研究に従事．博士（情報科学），ACM 会員．