データ並列言語における通信最適化のためのコード移動手法

渡邊誠也[†,☆] 湯淺太一[††]

本論文では，MIMD型分散メモリ並列計算機をターゲットとするデータ並列言語のコンパイラにおいて，プロセッサ間通信の最適化を強力に行うためのコード移動手法を提案する．提案する手法では，通信コードのみならず計算コードをも対象とする広域的なコード移動を行う．通信と計算のオーバラップを効率良く行うために，通信コードを可能な限りまとめて配置する．さらに，通信コードをまとめる際は，通信集合化（message aggregation）の適用機会が最大となるように配置する．本手法により，通信遅延の隠蔽と通信集合化を適用しやすいコード配置が得られ，通信オーバヘッドを軽減することが可能となる．性能評価の結果から，プロセッサ間通信のコストが計算コストと比較して大きい実行環境において，本手法が特に有効であることを確認した．

A Code Motion Technique for Communication Optimization of Data-parallel Languages

NOBUYA WATANABE[†,☆] and TAIICHI YUASA[††]

This paper presents a code motion technique which enables optimization on inter-processor communication, for compilers of data-parallel languages for MIMD distributed-memory parallel computers. The proposed technique performs code motion over a wide area of code sequence, including not only communication code but also computation code. It collects as many communication codes as possible and moves them into a same place for efficient overlapping of computation with communication. Furthermore, communication codes are located so that the resulting program can maximize the opportunity of applying *message aggregation*. This technique makes it easy to apply communication latency hiding and message aggregation, and thus to reduce communication overhead. From the results of performance evaluation, we confirmed that the proposed technique is particularly useful for those execution environments where the communication cost is higher than the computation cost.

1. はじめに

データ並列は，大規模なデータ集合の各要素に対して同一の演算を行う際の並列性に着目した計算モデルである．実用的な並列アプリケーションの多くは，データ並列モデルで記述されていることが知られている．これまでにデータ並列に基づく様々なプログラミング言語が提案されてきた[3),7),11),13)]．これらのデータ並列言語の多くは，意味論としてSIMD実行モデルを採用しているが，SIMD方式以外の並列計算機の普及にともない，広範囲の並列計算機上で利用されることがある．近年，多数の汎用プロセッサで構成されるMIMD型分散メモリ並列計算機が登場しており，本稿では，これらのマシンをターゲットとしたデータ並列言語のコンパイル方式を考察する．

ターゲットマシンを特定しないデータ並列言語では，プロセッサ数を意識しないプログラミングが可能である．そのため，通常，物理プロセッサの数を大きく上回る数のデータが用いられる．このようなプログラムを分散メモリ並列マシンで実行する場合，プログラムで使われている膨大な数のデータは，各プロセッサのメモリに分散して割り当てられる．各プロセッサでは，割り当てられたデータのサイズに一致する反復回数の繰返しによりプログラムの各ステップが実行される．そのため，ソースプログラムのテキスト上は単純に表現された計算でも，実際には粒度の大きい計算となる特性がある．

分散メモリ並列マシンにおいては，プロセッサ間通信のコストがプロセッサにおける計算のコストに比較

† 豊橋技術科学大学情報工学系
Department of Information and Computer Sciences, Toyohashi University of Technology
☆ 現在，岡山大学工学部情報工学科
Presently with Department of Information Technology, Faculty of Engineering, Okayama University
†† 京都大学大学院情報学研究科通信情報システム専攻
Department of Communications and Computer Engineering, Graduate School of Informatics, Kyoto University

して大きい．そのため，通信によるオーバヘッドを極力少なくすることが重要となる．これまでに通信のオーバヘッドをおさえるために提案された手法として，次のものがあげられる．

ループ文（繰返し文）の本体で実行される粒度の細かい通信を，ループ文の繰返しの前で一括化する通信ベクトル化と呼ばれる手法がある．この手法は，1つのループ文に着目して行われる処理である．さらに，複数のベクトル化された通信で同一配列に対する通信を1つにまとめる合体化（coalescing），同一プロセッサ間ごとの複数の通信を1つの通信で行う集合化（aggregation）といった手法[4),12)]が提案されている．これらの手法も，基本的に，ループ文の繰返しの前に置かれた複数の通信に対して局所的に行われる．また，データフロー解析の情報に基づき，冗長な通信を削減する手法，通信を送信と受信に分割して移動させることで，通信と計算のオーバラップを行う手法[2)]も提案されている．

これらの手法におけるコード移動は，通信コードのみの移動にとどまっている．通信コードの移動のみならず，通信以外のコード（計算コードと呼ぶ）をも移動の対象とすることにより，通信と計算のオーバラップをより効率的に行える可能性がある．

SIMD意味論のデータ並列言語において，ソースプログラムレベルでコード移動を行うことは，ターゲットコードレベルではループ単位の移動に相当する．つまり，移動したコードに対する実際の計算量はかなり大きいことになる．通信と計算のオーバラップを行う場合，双方の処理時間が等しいときに最大の速度向上がはかれることが知られている．計算量の多い計算コードに対しては，通信コードをまとめて配置することにより，効率の良い通信と計算のオーバラップが期待される．

本論文では，MIMD型分散メモリ並列計算機をターゲットとするデータ並列言語のコンパイラにおいて，通信最適化を強力に行うための広域的なコード移動手法を提案する．提案する手法では，コード列をいくつかの実行段階（ステージ）に分割することを基本とするコード移動を行う．提案する手法により，通信と計算のオーバラップをより効率的に行うことが可能になる．さらに，通信集合化を行う機会が増える．

以下，まず2章にて，通信と計算の効率的なオーバラップを行うためのコード移動の必要性と，通信コードの一括配置の有効性を示し，コード移動の基本方針を示す．3章にて，本稿で用いる用語の定義と表記法を示す．4章にて本手法の基本となるステージ分割について述べ，5章で，ステージ分割のアルゴリズムを示す．6章では具体的な適用例を示し，7章で性能評価を行った結果を示す．8章で関連研究を示し，最後に9章でまとめる．

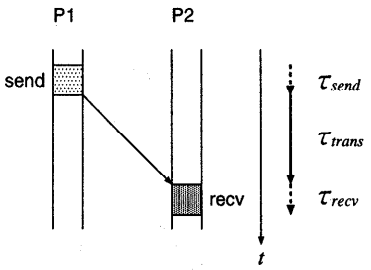


図1 プロセッサ間通信に要する時間
Fig. 1 Time intervals of an inter-processor communication.

2. 通信と計算のオーバラップ

本章では，まず分散メモリ並列マシンにおけるプロセッサ間通信のモデルを示す．そのモデルに基づき通信と計算の効率的なオーバラップを行うためには，広域的な計算コードの移動が必要であることを示す．また，コード移動の方針としては，通信コードをできる限り同一箇所にまとめて配置することが有効であることを示す．

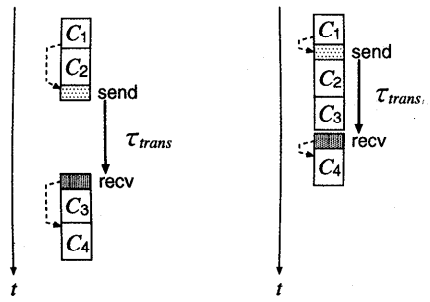
2.1 プロセッサ間通信のモデル

図1に，プロセッサP1からプロセッサP2へのプロセッサ間通信における処理時間を図示する．1回のプロセッサ間通信の処理は，次のように分割して考えることができる．

(1) 送信データをユーザ領域から通信システムの送信バッファ領域にコピーし，データの送信を開始する．
(2) プロセッサ間のネットワークを介して，データを転送する．
(3) システムの受信バッファ領域からユーザ領域へデータをコピーする．

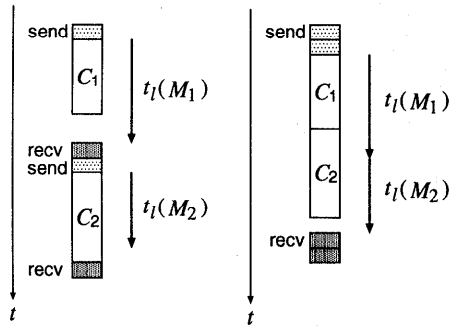
それぞれの処理時間を τ_{send}, τ_{trans}, τ_{recv} で表す．$\tau_{send}+\tau_{recv}$ は**通信オーバヘッド**と呼ばれ，τ_{trans} は**通信遅延**と呼ばれる．

データ並列プログラムを実行する場合，一般に，各プロセッサはデータの送信と受信の両方を行う．図1のP1は，(1)の処理を行った後，他のプロセッサから送信されたデータを待ち，データが到着したら(3)の処理を行う．データの到着を待っている間（つまり，(2)の処理の間），プロセッサは他の処理を行うことが可能である．受信データ待ちの間，他の計算処理を行うことは，通信と計算のオーバラップと呼ばれる．



(a) before code motion (b) after communication code motion

図2 通信と計算のオーバラップ
Fig. 2 Overlapping communication and computation.



(a) individual overlapping (b) combined overlapping

図3 コード一括化の効果
Fig. 3 Effects of combining codes.

通信と計算のオーバラップにより，通信遅延 τ_{trans} を隠蔽することができる．

2.2 コード移動による通信と計算のオーバラップ

図2 に，オーバラップのための通信コード移動の例を示す．図において，C_1 は送信する値の計算，C_4 は受信した値を使用する計算，C_2 と C_3 は通信と依存関係のない計算を表す．ここでの計算は，一般にループ文により実行される計算となる．点線の矢印はコード間の依存関係を表し，太い実線は通信遅延を表している．

図2の(a)は，通信コードの移動を行わない場合の処理を示している．図2の(b)に示すように，送信する値が確定した直後に送信コードを，受信したデータを参照する直前に受信コードを移動すれば，通信と計算のオーバラップが行える．

C_2 と C_3 が空の場合，通信コードの移動だけでは通信と計算のオーバラップを行うことはできない．C_1，C_4，および，通信の処理をいくつかに分割し，局所的なコード移動によりパイプライン的な実行を行うことでオーバラップを実現する手法が提案されている[5]．しかし，この手法では，最適な分割サイズを求めることが難しい．また，この手法でのオーバラップによる速度向上はきわめて低いという報告[10]もあり，効率的なオーバラップとはいえない．

現在着目しているコード領域よりさらに広い範囲のコードに着目し，送信コードと受信コードの間に，通信とオーバラップ可能な計算コードを移動して配置することで，効率の良いオーバラップを行うことができる．すなわち，より広域的なコード領域に対して，通信コードのみならず計算コードの移動をも考慮したコード移動が要求される．

2.3 通信コードの配置

通信と計算のオーバラップは，通信遅延と等しい時間を要する計算をオーバラップさせたときに，最も効率が良いことが知られている[10]．しかし，コンパイル時に，通信や計算に要する時間を正確に見積もることは困難である．つまり，プログラム中に現れる個々の通信に対して，最も効率の良いオーバラップを行える計算コードを求めるのは困難である．複数の通信コードを1カ所にまとめて配置し，それらとオーバラップ可能な計算コードもまとめて配置することが効率的である．その有効性を次に示す．

計算 C_1，C_2 と通信 M_1，M_2 があり，各コード間に依存関係が存在しない場合を考える．計算 C の処理に要する時間を $t(C)$，通信 M における通信遅延を $t_l(M)$，通信オーバヘッドを $t_o(M)$ とする．各コード間には依存関係が存在しないため，計算 C_1 と通信 M_1 をオーバラップでき，また，計算 C_2 と通信 M_2 もオーバラップできる．この2組のコードについて，(a) 個別にオーバラップする場合，(b) 一括して行う場合の実行時間を考える．**図3** に，それぞれの場合における処理の流れを示す．(a)，(b) それぞれの場合における実行時間 t_a，t_b は，次の式で与えられる．

$$t_a = \max\{t(C_1), t_l(M_1)\} + \max\{t(C_2), t_l(M_2)\} + t_o(M_1) + t_o(M_2)$$

$$t_b = \max\{t(C_1) + t(C_2), t_l(M_1) + t_l(M_2)\} + t_o(M_1) + t_o(M_2)$$

ここで，$\max\{x_1, y_1\} + \max\{x_2, y_2\} \geq \max\{x_1 + x_2, y_1 + y_2\}$ より，$t_a \geq t_b$ の関係が成立する．これは，2つの通信を一括してオーバラップを行う場合の方が，個別に行う場合より実行時間を短縮できることを意味する．

以上の議論から，通信コードは，可能な限り同一箇所に配置することが有効であり，通信と計算のオーバラップを効率良く行うことが可能である．さらに，同一箇所に配置した通信が，同一プロセッサに対する通信の場合，通信集合化を適用することが可能となる．集合化によ

り，通信オーバヘッドをより小さくできる可能性もある．本論文で提案する手法は，この方針を基本とするコード移動を行う．その詳細は，4章と5章で述べる．

3. 準　　備

3.1 入力プログラムの表記

本稿で想定する入力プログラムは，SIMD意味論のデータ並列言語で記述されているものとする．本稿では，入力プログラムの表記にデータ並列C言語NCX[13]の構文を用いる．

NCXはSIMD的に動作する仮想プロセッサ（virtual processor，以下VPと略す）群を仮定し，VPに対するプログラムを与えることでデータ並列計算を記述する．各VPに変数xとyが割り当てられているとき，

```
x = y + 1;
```

は，その時点で計算に参加している各VPにより並列に計算される．

VP間通信は，あるVPが他のVP上の変数を参照する遠隔参照の形で表現する．次に示す例は，各VPが2近傍のVP上の変数yの値の平均を計算し，その値を個々のVP上の変数xに格納するプログラムである．ここで変数iは，個々のVPを特定するための変数である．

```
x = (y@(i-1) + y@(i+1)) / 2.0;
```

3.2 分散メモリマシンにおけるデータ並列プログラムの処理

本稿で対象とするターゲットマシンはMIMD型分散メモリ並列計算機である．MIMD型分散メモリマシンにおけるプログラムにおいて，プロセッサ間通信は，通常，メッセージパッシングで記述される．

データ並列言語を分散メモリマシンで実行する場合，ソースプログラムは，コンパイラによって，メッセージパッシングのコードが追加されたプログラムに変換される．また，ソースプログラムでは，通常，プロセッサの個数をはるかに越える数のVPが使用されるため，1台のプロセッサで複数のVPの実行を担当するプログラムに変換される．VP上に割り当てられた変数は，各プロセッサが担当するVP数に等しいサイズの配列で表現される．SIMD意味論では，基本的に演算の各ステップを担当VP数と等しい反復回数のループ文で逐次実行することにより，VPの実行がエミュレートされる．

ソースプログラムにおける遠隔参照は，プロセッサ間通信を要する処理と，そうでない処理に分けられる．遠隔参照されるVPと参照しているVPが，同一のプロセッサに割り当てられている場合，その遠隔参照は，プロセッサにローカルなメモリ参照で実現される．一方，2つのVPが異なるプロセッサに割り当てられている場合，プロセッサ間通信を必要とする．したがって，遠隔参照を含む計算は，次の3つの処理に分けられる．

(1) 通信処理
(2) 計算に必要なすべてのデータがローカルメモリ上に揃っているVPに対する計算処理
(3) (2)以外のVPに対する計算処理

(1)では，他のプロセッサにおける計算で使用されるデータをローカルメモリに保持している各プロセッサが，そのデータの送信を行う．また，他のプロセッサのメモリ上のデータを必要とするプロセッサは，データの受信を行う．受信したデータは，(3)の処理で使用される．

本稿では，通信と計算のオーバラップを考慮する観点から，メッセージパッシングの通信命令としては，ノンブロッキング送信を仮定する．つまり，送信の完了を待たずに，送信命令に続くコードを実行できるものとする．また，各VPのプロセッサへの割当て方法は，近傍通信に対して性能が良いとされるブロック分割を仮定する．

3.3 諸　定　義

定義1（処理間の依存関係）　2つの処理c_1とc_2があるとき，c_1の実行後でなければc_2が実行できない場合，「c_2はc_1に依存する」という．　□

たとえば，処理c_1で変数に値を代入して，c_2でその変数を参照している場合，c_2はc_1に依存する．

定義2（実行依存グラフ）　処理の有限集合Cと，処理間の依存関係を表す有向辺の集合$D(\subseteq C \times C)$で表される有向グラフを**実行依存グラフ**と呼び，$G=(C,D)$と表記する．処理c_2がc_1に依存するとき，c_1からc_2に伸びる有向辺により依存関係を表し，$\langle c_1, c_2\rangle$と表記する．実行依存グラフの有向辺$\langle c_1, c_2\rangle$には，非負の重み$w(c_1, c_2)$が付いているものとする．　□

有向辺に対してどのような重み付けを行うかは，4.3節で述べる．

定義3（開始点と終了点）　実行依存グラフGにおいて，他のどのノードにも依存しないノードのことを**開始点**と呼び，他のどのノードにも依存されないノードのことを**終了点**と呼ぶ．　□

本稿では，ソースレベルにおけるif文やループ文といった制御構造は考慮せず，代入文が連続する基本ブロックのコードを対象とする．したがって，実行依存グラフにおいては，必ず開始点と終了点が存在し，また，循環は存在しない．

定義4（クリティカルパス）　実行依存グラフGにおいて，処理c_1から処理c_nに至る経路$c_1 \rightarrow c_2 \rightarrow$

$\ldots \rightarrow c_n$ に対し，$\sum_{i=1}^{n-1} w(c_i, c_{i+1})$ をその経路の長さと定義する．いずれかの開始点から，いずれかの終了点までの経路のうち，長さが最大となる経路を G のクリティカルパスと呼ぶ． □

4. ステージ分割

提案するコード移動手法の基本は，与えられたコードに対応する処理集合を，ステージと呼ばれるいくつかの実行段階に分割することである．ステージとは，1カ所にまとめることのできる通信処理の集合，および，それらと並行に実行可能な計算処理の集合である．本章では，まず，ステージ分割の定義を示し，どのようなステージ分割を行うかを述べる．

4.1 ステージ分割の定義

定義5（ステージ分割） 実行依存グラフ $G = (C, D)$ のステージ分割とは，次の条件を満たす C の部分集合列 $S_1, S_2, \cdots, S_{n_s}$（$n_s$ はステージ数）とする．

(1) S_i と S_j $(i \neq j)$ は互いに素 $(S_i \cap S_j = \emptyset)$ であり，$\bigcup_{i=1}^{n_s} S_i = C$ を満たす．

(2) 同じステージに属する2つの通信処理の間には，依存関係が存在しない．

(3) 同じステージに属する通信処理と計算処理との間には，依存関係が存在しない． □

定義5では，同一ステージに属する計算処理間に依存関係が存在することを許している．これは，2つの計算処理間に依存関係があっても，それらが同一ステージ内の通信処理との間に依存関係が存在しない限り，通信処理と並行して実行できるからである．また，計算処理間に依存関係の存在を許すことによって，より多くの計算処理を通信とオーバラップさせることができる．

4.2 ステージ分割における目標

本稿で述べるコード移動手法では，多くの並列計算機で通信オーバヘッドが大きいことから，通信集合化の機会を最大化することに主眼をおき，通信処理をまとめて配置することを基本方針とする．

通信と計算のオーバラップによる通信遅延隠蔽の最大化を優先することも考えられるが，次の理由により集合化を優先する．オーバラップによる効果が最大となるのは，通信と計算に要する時間が同じとなる場合であり，コンパイル時にそれらに要する時間を正確に見積もることは困難である．また，通信処理をまとめて配置することにより，2.3節で示したように，通信と計算のオーバラップも効率的に行える．

以上の基本方針に基づき，ステージ分割における目標を次のように定める．

- ステージ数を最小にする．
- 通信集合化の可能性をできるだけ高くする．

これらの目標は，4.4節で定義する評価関数の値を最小とすることにより達成する．

4.3 実行依存グラフの有向辺に対する重み付け

実行依存グラフ $G = (C, D)$ の有向辺 $\langle c_i, c_j \rangle \in D$ に対する重み $w(c_i, c_j)$ を次のように定める．

$$w(c_i, c_j) = \begin{cases} 0, & (c_i, c_j \text{ が計算ノードの場合}) \\ 1, & (\text{上記以外の場合}) \end{cases}$$

重み $w(c_i, c_j)$ の値は，c_i と c_j が同一ステージに配置可能である（0のとき）か，同一ステージに配置不可能である（1のとき）かを表している．

4.4 通信集合化を考慮した評価関数

ここでは，ステージ分割での通信処理の配置を評価する関数を導入する．複数の通信処理を1つのステージに配置する際，その配置の仕方により，生成されたコードの実行速度に大きく影響を与える．これは，通信処理の配置の仕方により，通信集合化が適用できるかどうかが決まり，プロセッサ間通信の回数が異なるためである．

通信集合化では，同一プロセッサに対する複数の通信を一括化する．同一ステージに複数の通信処理があっても，それらが同一プロセッサに対する通信でなければ，集合化を行うことはできない．集合化が適用された場合，適用前に比べてプロセッサ間通信の回数が減少することに着目し，プロセッサ間通信の回数をステージ分割の評価関数として用いる．しかし，コンパイル時に，プロセッサ間通信の正確な回数を求めることは困難であるため，最小限必要なプロセッサ間通信の回数を求めることにより近似する．

ステージ分割 $S = \{S_1, S_2, \cdots, S_{n_s}\}$ において，必要なプロセッサ間通信の回数の下限 $M(S)$ は，次式で与えられる．

$$M(S) = \sum_{i=1}^{n_s} m(S_i) \qquad (1)$$

ここで $m(S_i)$ は，ステージ $S_i (1 \leq i \leq n_s)$ におけるプロセッサ間通信回数の下限とする．この関数 $M(S)$ をステージ分割の評価関数とする．関数 $M(S)$ の値が最小となるようなステージ分割 S を求めることで，通信処理を同一ステージにできる限りまとめて配置すること，および，通信集合化を行いやすくまとめることを達成できる．

ここでは，VPを特定する一次元目のインデックスに基づいたプロセッサへのブロック割当てを行ってい

る場合の，ステージ S_i におけるプロセッサ間通信回数の下限 $m(S_i)$ の求め方を示す．この場合，遠隔参照における一次元目のインデックスを調べることで，その遠隔参照がプロセッサ間通信を要するかどうかを判定する．K を整数，vを変数名とするとき，次の式で表現される近傍 VP に対する遠隔参照

v `@(i+`K `, ···)`

は，$K<0$ のとき「左への遠隔参照」，$K>0$ のとき「右への遠隔参照」，$K=0$ のとき「自分への遠隔参照」と呼ぶ．一次元目のインデックスが，上記の形式でない場合は，「不規則な遠隔参照」と呼ぶ．

まず，ステージ S_i に不規則な遠隔参照を1つも含まないとき，次の3つの場合がある．S_i に遠隔参照を1つも含まない，あるいは，自分への遠隔参照のみを含む場合（ケース1），左への遠隔参照を含まない，あるいは，右への遠隔参照を含まない場合（ケース2），上記以外の場合（ケース3）の3つである．ステージ S_i におけるプロセッサ間通信の回数の近似値 $m(S_i)$ は，次のように定義できる．

$$m(S_i)=\begin{cases}0, & (\text{ケース}1)\\ 1, & (\text{ケース}2)\\ 2, & (\text{ケース}3)\end{cases}$$

同一方向の遠隔参照だけが存在する場合，最低1回のプロセッサ間通信が必要であり，方向の異なる遠隔参照が存在する場合，各方向で1回ずつ，最低2回の通信が必要である．

ステージ S_i に不規則な遠隔参照を含む場合，生成されるプロセッサ間通信の数は，処理系のコード生成方式と実行時の環境（プロセッサ数）に依存する．この論文では，不規則な遠隔参照を先に述べたような特殊な意味で用いている．不規則な遠隔参照の場合でも，特定の通信パターンを有する場合には，通信コストを予測できることがある．たとえば，すべての VP が特定の1つの VP 上の変数を参照する遠隔参照に対しては，ブロードキャストを用いることができ，これは全対全通信に比べ，一般的に通信のコストは低い．そのような場合には，実際の通信コストを反映した値を $m(S_i)$ に設定することが望ましい．

4.5 通信と計算のオーバラップの実現方法

通信処理と計算処理の双方の処理で構成されるステージにおいて，通信と計算のオーバラップを行うことができる．通信と計算のオーバラップは，ステージに含まれる通信処理と計算処理に対応するコード列を次のように並べることで実現できる．

(1) 送信コード列
(2) 計算コード列
(3) 受信コード列

ここで，(1)，(3)はそれぞれ，ステージを構成するすべての通信処理に対応する送信コードの列，受信コードの列である．(2)は，ステージを構成する計算処理に対応する計算コード列である．

5. ステージ分割アルゴリズム

本章では，実行依存グラフ G のステージ分割を行うアルゴリズムを示す．ここで示すアルゴリズムは，ターゲットの並列計算機に固有の特性，たとえば，通信性能や，計算性能の詳細は考慮しない．これらのターゲットに固有の特性を反映したコード配置を行うことにより，最大の速度向上を得られることが期待されるが，ここでは，ターゲット特性に依存する詳細には言及せずに，一般的なアルゴリズムを示すにとどめる．

アルゴリズムは以下の3つのステップで構成される．

step 1 クリティカルパスを求め，クリティカルパス上の各ノードのステージを決定する．

step 2 クリティカルパス上にない各通信ノードに対して，それらを配置するステージを決定する．

step 3 ステージの決定していない各計算ノードに対して，それらを配置するステージを決定する．

以下の節で，各ステップの処理の詳細を説明する．

5.1 クリティカルパスの決定（step 1）

まず，実行依存グラフ $G=(C,D)$ の各ノード $c\in C$ に対して，開始点から c に至る経路の長さの最大値 $a(c)$ と，c から終了点に至る経路の長さの最大値 $b(c)$ を求める．クリティカルパスの経路の長さ L_{CP} は，次の式で与えられる．

$$L_{CP}=\max_{c\in C} a(c)$$

分割するステージ数を $n_s=L_{CP}+1$ とし，ステージを順に，$S_1,S_2,\cdots,S_{n_s}$ とする．

クリティカルパス上の処理集合 C_{CP} は，次の集合で与えられる．

$$C_{CP}=\{c\in C \mid a(c)+b(c)=L_{CP}\}$$

C_{CP} に含まれる各処理 c を，ステージ $S_{a(c)+1}$ に配置する．

5.2 通信処理のステージ決定（step 2）

step 2 では，クリティカルパス上にない通信処理のステージを決定する．本ステップのアルゴリズムは，基本的に，4.1 節で述べた定義5の制約下で通信処理の配置方法のすべての組合せに対して，式(1)で示した評価関数 $M(S)$ が最小となるような配置を探索するものであり，組合せ最適化問題を解くアルゴリズム

と同じである．次に示す関数 $M(S)$ の特性により，分枝限定法を用いることが可能であり，無駄な探索を行わなくて済む．

与えられたグラフ G のステージ分割における関数 $M(S)$ の下限は，step 1 を完了した時点のステージから計算される $M(S)$ の値である．クリティカルパス上にない通信処理をステージに追加した際，$M(S)$ の増加量はつねに非負であり，$M(S)$ の値が減少することはない．

$M(S)$ が最小となる配置が複数ある場合，どの配置方法を選ぶかにより，通信と計算のオーバラップの程度が大きく異なってくる．理想的には，step 3 における計算処理の配置についても，評価関数を導入し，最大限のオーバラップを行える配置を選択することが望まれるが，ここでは，各ステージの通信処理の数が均等となるような配置を選択する方法をとる．

5.3 計算処理のステージ決定（step 3）

step 3 では，クリティカルパス上にない計算処理のステージを決定する．ここでの目的は，通信に対して多くの計算をオーバラップする配置を求めることである．これは，通信処理を含むステージに対して優先的に，ステージの決定していない計算処理を追加することにより実現する．

以下にアルゴリズムを示す．ここで C_f の初期値は，クリティカルパス上にない計算処理全体の集合とする．

(1) $C_f = \emptyset$ なら，終了．
(2) 処理 $c \in C_f$（ただし，c は $a(c)$ が最小の c）を配置可能なステージの集合 S を求める．
(3) S に通信処理を含むステージが存在すれば，通信処理を含むステージのいずれかに c を追加し，存在しなければ，S の任意のステージに c を追加する．
(4) C_f から c を取り除き，(1) へ飛ぶ．

(3) において，処理 c を配置可能なステージが複数ある場合，多くの自由度が残っている．理想的には，ターゲットに固有の特性を考慮し，c を配置するステージを決定することが望ましい．本稿では，ステージの決定していない残りの処理の配置に対する自由度を高めるために，ステージ番号の最も小さいステージを選択する方法をとる．

6. 適 用 例

入力として，**図 4** に示す 5 つの代入文で構成されるコード列が与えられたとする．図 4 に対応する処理の列は，**図 5** のようになる．図 5 において，各処理のラベルに付けられた番号は，対応する入力コードの番号であり，ラベル C は計算処理を，ラベル M は通信処理を，ラベル C' は通信結果を使用する計算処理をそれぞれ表す．また，**remote**(*dest*, *src*) は，プロセッサ間通信により *src* を *dest* に格納する通信処理を表している．

```
1:  a = ··· ;
2:  b = a@(i+1) ···;
3:  c = ···;
4:  d = c@(i+1) ···;
5:  e = b@(i-1) ···d ···;
```

図 4 入力コード列
Fig. 4 An input code sequence.

```
C1:   a = ···;
M2:   remote(temp1, a@(i+1));
C2:   b = a@(i+1) ···;
C2':  b = temp1 ···;
C3:   c = ···;
M4:   remote(temp2, c@(i+1));
C4:   d = c@(i+1) ···;
C4':  d = temp2 ···;
M5:   remote(temp3, b@(i-1));
C5:   e = b@(i-1) ···d ···;
C5':  e = temp3 ···d ···;
```

図 5 入力に対応する処理の列
Fig. 5 Process sequence corresponding to the input code.

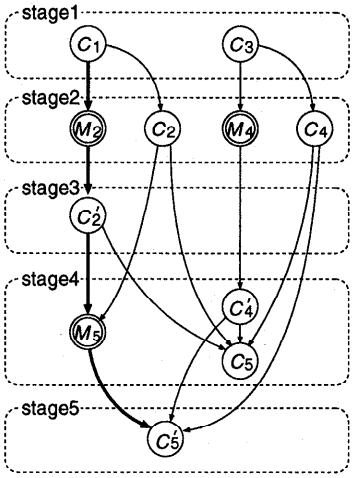


図 6 実行依存グラフとそのステージ分割
Fig. 6 Execution dependency graph and its stage partition.

図 5 の処理の列から実行依存グラフを生成し，ステージ分割を行った結果を**図 6** に示す．図 6 で，“○”は計算処理に対応するノードを，“◎”は通信処理に対応するノードを，太い線はクリティカルパスをそれぞれ表している．

最終的に図 **7** に示すコード列が得られる．図 7

```
stage 1
      C1:   a = ···;
      C3:   c = ···;
stage 2
M2(send):   send(a, P-1);
M4(send):   send(c, P-1);
      C2:   b = a@(i+1) ···;
      C4:   d = c@(i+1) ···;
M2(recv):   recv(temp1, P+1);
M4(recv):   recv(temp2, P+1);
stage 3
      C2':  b = temp1 ···;
stage 4
M5(send):   send(b, P+1);
      C4':  d = temp2 ···;
      C5:   e = b@(i-1) ···d ···;
M5(recv):   recv(temp3, P-1);
stage 5
      C5':  e = temp3 ···d ···;
```

図7 コード移動が行われたコード列
Fig. 7 Code sequence after code motion.

において，P はプロセッサ番号が格納された変数，**send**(*var*, *p*) は，変数 *var* の値をプロセッサ *p* へ送信するコード，**recv**(*dest*, *p*) は，プロセッサ *p* から受信した値を変数 *dest* に格納するコードをそれぞれ表す．2つのステージに通信コードが配置され，それぞれのステージで，通信と計算のオーバラップが行われている．また，ステージ2における2つの通信は，同一プロセッサに対する送信および受信であるため，通信集合化を行うことが可能である．

従来の手法のように移動の対象を通信コードに限った場合，図5のラベル M_4 の通信処理における送信は，ラベル C_3 の代入に依存するために，C_3 の上方へ移動することはできず，ラベル M_2 の通信処理との集合化を行うことはできない．

7. 性能評価

本章では，提案する手法による効果を確認するために，並列計算機とワークステーション・クラスタ上でプログラムを実行し，実行時間の測定を行った結果を示す．

評価に用いたプログラムは，Livermore Loop[8]の Kernel #18 である．ループの本体に，6個の代入文と，合計23個の遠隔参照がある，このうち，ループ不変の変数に対する遠隔参照が9個ある．

ソースプログラムは，データ並列C言語NCXで記述し，MPI[9]版NCXコンパイラ[6]により変換されたプログラムに対し，コード移動を行わないコード（C1と略す）とコード移動を行ったコード（C2と略す）を用意した．どちらのコードも，ループ不変となる変数に対する通信は，ループ文の前へ移動されている．

表1 NEC Cenju-3での実行時間
Table 1 Execution times on NEC Cenju-3.

プロセッサ数	C1 (sec)	C2 (sec)	速度向上率 σ
2	938.14	994.58	−6.0%
4	370.35	321.53	13.2%
8	160.06	144.77	9.6%
16	71.41	65.25	8.6%
32	35.53	33.90	4.6%
64	20.54	17.53	14.7%
128	13.54	10.61	21.6%

表2 Hitachi SR2201での実行時間
Table 2 Execution times on Hitachi SR2201.

プロセッサ数	C1 (sec)	C2 (sec)	速度向上率 σ
2	314.40	354.54	−12.8%
4	155.41	153.55	1.2%
8	70.82	57.85	18.3%
16	32.44	27.38	15.6%
32	14.72	12.45	15.4%
64	8.03	6.81	15.2%
128	4.29	3.56	17.0%

実行環境として次の3つの計算機を用い，使用するプロセッサ数を変化させて実行時間の測定を行った．

- 分散メモリ型並列計算機 NEC Cenju-3
- 分散メモリ型並列計算機 Hitachi SR2201
- ワークステーション・クラスタ

ワークステーション・クラスタは，HP9000 モデル715/80（PA-RISC 80 MHz）互換機を10 Mbpsのイーサネットで接続した環境である．メッセージパッシングのライブラリには，すべての環境でMPICH[1]を用いた．

問題のサイズは，512 × 512 とし，ループ回数は，Cenju-3 と SR2201 では1000，ワークステーション・クラスタでは250とした．ソースプログラムで使用したVP数は，問題のサイズと同一数であり，一次元目のインデックスでブロック分割しプロセッサに割り当てた．ループ不変となる変数以外の14個の遠隔参照のうち，プロセッサ間通信をともなう遠隔参照は7個である．このうち冗長となる通信が2個あり，最終的に5組の送受信命令が含まれるコードとなる．

表1〜3 に，各実行環境での実行時間と，C1 に対する C2 の速度向上率を示す．速度向上率 σ は，$\sigma = (t_1 - t_2)/t_1$ で求めた．ここで t_1，t_2 は，それぞれ C1 と C2 の実行時間を表す．

7.1 並列計算機での実験結果に対する考察

Cenju-3での実験結果より，プロセッサ数が2のときを除いて，5%から22%の速度向上が確認できる．また，SR2201 での実験結果より，プロセッサ数が8以

表3 ワークステーション・クラスタでの実行時間
Table 3 Execution times on workstation cluster.

プロセッサ数	C1 (sec)	C2 (sec)	速度向上率 σ
2	139.23	119.00	14.5%
4	79.19	60.20	24.0%
8	46.30	32.42	30.0%
16	47.13	34.63	26.5%

上のとき，15%から18%の速度向上が確認できる．

双方の結果において，プロセッサ数が2の場合，速度向上が現れていない．C2は，コード移動によりVP実行エミュレーションのためのループの数が増えたコードになっており，ループ実行のオーバヘッドが増えている．プロセッサ数が少ない場合，計算量に比べて通信量は少なく，コード移動による通信と計算のオーバラップの効果があまり現れないためである．

表1において，速度向上率がプロセッサ数32で落ち込んでいる．これは，C1とC2の台数効果の違いが原因である．つまり，C2の場合，プロセッサ数16を越えたときに台数効果の落込みがあるのに対して，C1の場合，プロセッサ数32を超えたときに落込みを生じているからである．

また，使用プロセッサ台数が増えたときにsuper-linearな速度向上が現れている．この原因は，キャッシュのヒット率の向上にともなう速度向上と考えられる．つまり，使用プロセッサ数が増えると，1台のプロセッサが扱うデータ量が減少し，キャッシュのヒット率が高まり，各プロセッサにおける実行速度が向上したためである．

7.2 ワークステーション・クラスタでの実験結果に対する考察

プロセッサ数が8のときに，最大で30.0%の速度向上を確認でき，あらゆるプロセッサ数において良好な速度向上を達成できている．実験に使用した実行環境は，並列計算機の実行環境に比べ通信性能が著しく低い．そのため，コード移動による通信と計算のオーバラップの効果が顕著に現れている．

台数効果の観点からは，プロセッサ数が8のとき，最も良い性能となっている．使用プロセッサ数が増加した際，各プロセッサにおける計算量は少なくなるが，通信に要する時間は増加する．実験に用いたワークステーション・クラスタ環境では，並列計算機に比べ，通信性能が著しく低いために，通信に要する時間の増加量が大きい．したがって，プロセッサ数を8から16に増やした際に，通信に要する時間の増加が，プロセッサ1台あたりの計算時間の減少を上回ってしまい，実行時間が増加している．

7.3 従来方法との比較

比較のため，送信コードをそれが依存するコードの直後に移動し，通信遅延隠蔽を最大化したコード（C3と略す）の実行時間を測定した．実験の結果，Cenju-3では26%から45%，SR2201では10%から12%，ワークステーション・クラスタでは11%から12%，それぞれC3に比較してC2の方が高速であった．

この理由は，次のように考えられる．C3では，VPエミュレーションのためのループの数がC2に比べ多いコードとなっており，このループ実行にともなうオーバヘッドが大きい．また，同一プロセッサに対する通信が一括配置されていないため，通信オーバヘッドをC2ほど低下させることができていない．この実験により，提案する手法において，通信をまとめて配置することを優先することが，有効であることが分かった．

8. 関連研究

文献2)では，データの生産者と消費者に注目して送信命令，受信命令の移動を行い，不必要な通信の除去，通信遅延の隠蔽を行うための枠組みが述べられている．ここでの移動の対象は通信命令であり，計算コードを積極的に移動することは行われていない．

文献5)では，コンパイラが自動的に通信と計算をオーバラップするためのアルゴリズムが述べられている．そのままではオーバラップを行えないループ文本体のコードに対して，tilingという手法によりループをいくつかに分割し，パイプライン的な実行を行うことで，通信と計算のオーバラップを行っている．本稿で提案した手法を適用した後，通信とオーバラップ可能な計算コードがない場合，この手法を適用することも可能である．しかし，この手法では，通信回数が増えるために，通信オーバヘッドが比較的小さい並列計算機においては有効であるが，通信オーバヘッドが大きい計算機では，速度向上が期待できないと思われる．また，最適な分割数を求めるのも困難である．

9. おわりに

本稿では，MIMD型分散メモリ並列計算機をターゲットマシンとするデータ並列言語のコンパイラにおいて，プロセッサ間通信の最適化を強力に行うためのコード移動手法を提案した．提案した手法では，従来の手法で行われている通信コードの移動だけではなく，計算コードをも移動の対象としたコード移動を行うことで，より自由度の高いコード移動が可能となる．コード移動においては，通信コードをまとめて配置することで，効率的な計算と通信のオーバラップが可能

となる．さらに，通信集合化を行いやすいように通信をまとめるために，集合化による効果も期待できる．

本手法を用いてコード移動を行ったプログラムを並列計算機とワークステーション・クラスタで実際に動作させ，速度向上がはかれることを確認した．プロセッサ間通信のコストが大きい環境において，特に有効であることが分かった．

本稿で提案したコード移動手法では，プログラムの実行環境に固有のプロセッサ性能や通信性能といった特性は考慮していない．実験結果から，プログラムの実行環境により速度向上に違いが現れることも確認できた．用いる実行環境における最高の性能を得るためには，実行環境の特性を考慮し，その環境に特化したコード配置を行うことが必要である．実行環境に固有の特性をいかにパラメータ化し，また，その特性をどのようにアルゴリズムに対して組み込むかが課題である．

謝辞 実行環境を提供していただいたNEC並列処理センターの方々，ならびに，有益なコメントをいただいた査読者の方々に感謝する．

参考文献

1) Gropp, W. and Lusk, E.: *User's Guide for mpich, a Portable Implementation of MPI*, Argonne National Laboratory (1996).
2) Hanxlenden, R. and Kennedy, K.: GIVE-N-TAKE - A Balanced Code Placement Framework, *Proc. ACM SIGPLAN '94 Conference on Program Language Design and Implementation*, pp.107–120 (1994).
3) Hatcher, P.J. and Quinn, M.J.: *Data-Parallel Programming on MIMD Computers*, The MIT Press (1991).
4) Hiranandani, S., Kennedy, K. and Tseng, C.-W.: Evaluation of Compiler Optimizations for Fortran D, *Journal of Parallel and Distributed Computing*, Vol.21, pp.27–45 (1994).
5) 石崎一明，小松秀昭：分散メモリ並列計算機のためのコンパイラによる通信遅延隠蔽アルゴリズム，情報処理学会論文誌，Vol.38, No.9, pp.1849–1858 (1997).
6) 河内隆仁，渡邊誠也，小宮常康，湯淺太一：MPIを用いたデータ並列C言語NCXの実装，第55回情報処理学会全国大会論文集(1)，pp.337–338 (1997).
7) MasPar Computer Corp.: MasPar Parallel Application Language (MPL) Reference Manual, Ver.2.0 (1991).
8) McMahon, F.H.: *The Livermore Fortran Kernels: A Computer Test of the Numerical Performance Range*, Lawrence National Laboratory, UCRL–53745 (1986).
9) Message Passing Interface Forum: MPI: A Message-Passing Interface Standard (1995).
10) Quinn, M.J. and Hatcher, P.J.: On the Utility of Communication-Computation Overlap in Data-Parallel Programs, *Journal of Parallel and Distributed Computing*, Vol.33, pp.197–204 (1996).
11) Thinking Machines Corporation: C* Programming Guide (1993).
12) Tseng, C.-W.: An Optimizing FORTRAN D Compiler for MIMD Distributed Memory Machines, Ph.D. Thesis, Department of Computer Science, Rice Univercity (1993).
13) 湯淺太一，貴島寿郎，小西 浩：データ並列計算のための拡張C言語NCX，電子情報通信学会論文誌，Vol.J78-D-I, No.2, pp.200–209 (1995).

(平成10年3月26日受付)
(平成10年12月7日採録)

渡邊 誠也（正会員）

1970年生．1991年鹿児島工業高等専門学校情報工学科卒業．1993年豊橋技術科学大学工学部情報工学課程卒業．1995年同大学院工学研究科情報工学専攻修士課程修了．1998年同大学院工学研究科電子・情報工学専攻博士後期課程単位取得退学．現在，岡山大学工学部情報工学科助手．並列計算とプログラミング言語に興味を持ち，超並列計算用プログラミング言語に関する研究に従事．

湯淺 太一（正会員）

1952年神戸生．1977年京都大学理学部卒業．1982年同大学院理学研究科博士課程修了．同年京都大学数理解析研究所助手．1987年豊橋技術科学大学講師．1988年同大学助教授，1995年同大学教授，1996年京都大学大学院工学研究科教授，1998年同大学院情報学研究科教授となり現在に至る．理学博士．記号処理と超並列計算に興味を持っている．著書「Common Lisp入門」(共著)，「Scheme入門」，「C言語によるプログラミング入門」他．情報処理学会，ソフトウェア科学会，電子情報通信学会，IEEE，ACM各会員．