並列化コンパイラ TINPAR における
重複プロセッサ間通信の削減手法

窪田 昌史　辰巳 尚吾　後藤 慎也
森 眞一郎　中島 浩　富田 眞治

京都大学 大学院 工学研究科 情報工学専攻

データ並列性を内在するプログラムを対象とするメッセージ交換型並列計算機用並列化コンパイラでは，プロセッサ間通信の最適化による高速化が重要である．本稿では，同一プロセッサへの同一データの複数回の重複する送信を，1 度の送信へと削減する手法を述べる．この削減のため，データフロー解析を行い，送受信されるデータが送信側プロセッサで更新される点を求める．データの送受信では，送受信されたデータの送受信履歴を管理し，送信側プロセッサでデータの更新が行われると，その送受信履歴から該当データをはずすことで，新たに送受信を行う必要があることを表す．これは，ソフトウェアによるコヒーレンス制御を行う共有キャッシュにおける選択的自己無効化に相当し，プロセッサ間通信を行わずにデータの一貫性保証を実現している．

Eliminating Redundant Inter-Processor Communications for Parallelizing Compiler TINPAR

Atsushi KUBOTA,　Shogo TATSUMI,　Shin-ya GOTO,
Shin-ichiro MORI,　Hiroshi NAKASHIMA　and Shinji TOMITA

Department of Information Science,
Faculty of Engineering, Kyoto University

Optimizing inter-processor(PE) communications is crucial for parallelizing compilers for message-passing parallel machines to achieve high performance. In this paper, we propose a scheme to eliminate redundant inter-PE communications. This scheme utilizes data-flow analysis to find the definition point for each reference point where the definition and the reference are occurred in different PEs. If several references occurred in the same PE refer the data defined at the same definition point in another PE, redundant inter-PE communications are eliminated as follows: only one inter-PE communication is performed for the earliest reference and the local references to the previously received data are performed for the rest of the references. In order to guarantee the consistency of the data, a valid flag and a sent flag are provided for each chunk of received data. The Control of these flags is equivalent to the coherence control by the self invalidation in the compiler aided cache coherence scheme.

1　はじめに

現在，我々は並列化コンパイラ TINPAR(TINy PARallelizer) を開発している [4]．TINPAR は，多くの HPF[1] コンパイラと同様にデータ並列性を内在する科学技術計算のプログラムを対象としている．入力はデータ分割配置を指定した Fortran タイプの逐次言語で記述されたプログラムであり，出力はメッセージ交換型の分散メモリ型並列計算機用プログラムである．また，owner computes rule によって実行文の静的スケジュールを行う．

この前提下で，プログラムの実行を高速化するには，PE 間通信の最適化が最も重要である．この最適化には大きく分けて，以下の 2 つの手法が存在する．

- 通信時間の短縮
- 通信と計算のオーバーラップによる通信時間の隠蔽

このうち，通信時間の短縮については，

- 通信の起動回数の削減
- 通信量の削減

という手法がとられる．

前者の実現方には，同一配列の異なる要素の同一 PE への通信を一括化するメッセージベクトル化 (message vectorization) と，異なる配列の要素の同一 PE への通信を一括化するメッセージ集合化 (message aggregation) があり，TINPAR ではメッセージベクトル化機構が既に実現されている [4]．

本報告では，通信量を削減する方法であるメッセージ合体化 (message coalescing) と，TINPAR への実装について述べる．

以下，2章でメッセージ合体化を適用するための要件について述べる．3章でメッセージ合体化に必要なデータの定義，参照の連鎖を求めるためのデータフロー解析について述べる．4章では，データフロー解析の結果を用いたメッセージ合体化の実現法について述べる．最後に 5章でまとめとする．

2　メッセージ合体化

2.1　メッセージ合体化

メッセージ合体化 [2] とは，ある配列要素への参照が並列化によって PE 間にまたがるため通信が必要となり，かつ，その値への複数回の参照を行う受信 PE が存在する場合に，参照毎に PE 間通信を行わず，通信を 1 回にまとめることをいう．受信したデータを受信 PE 側で保管しておくことで，通信を伴う参照は，その保管されているデータの読み出しとして実行されることになる．

Hiranandani ら [2] がメッセージ合体化の例に挙げているプログラムを図 1に示す．配列 A,B が，各 PE に分散配置されていると，代入文の右辺の B の要素への参照が通信によって必要になることがある．この場合，B(i+2) で参照された要素は次のイタレーションでは B(i+1) として参照される．このように同一データを複数回参照する場合に，参照の度に必要な要素を通信する送受信コードを生成せずに，1 度の通信にまとめることができる．このような最適化がメッセージ合体化と呼ばれる．

```
do i=1,n
  A(i) = B(i+1) + B(i+2)
enddo
```

⇓配列 A，B をブロック分割して並列化

```
/* P は PE 数，p は PE 番号 */
send B(1..2) to p-1
recv tmp_B(1..2) from p+1
for l_i=1,n/P-2
  A(l_i) = B(l_i+1) + B(l_i+2)
endfor
A(n/P-1) = B(n/P) + tmp_B(1)
A(n/P) = tmp_B(1) + tmp_B(2)
```

図 1: メッセージ合体化

2.2　メッセージ合体化の適用条件

Hiranandani らの採用しているメッセージ合体化は，その適用条件について明確にされていない．そこで，我々は，データフロー解析を用いることにより，一度送信されたデータの再利用の可能性を検査し，不要な通信を削減する手法を提案する．

メッセージ合体化を行うためには，通信を伴う参照の起こる配列の各要素について，読み出される（参

照される）値が，書き込まれる（定義される）プログラム上の文，さらにループであればイタレーションまでを求める必要がある．以下，定義，参照の行われるプログラム上の文，ループであればさらにイタレーションをあわせて，それぞれ定義点，参照点と呼ぶ．プログラム中で同一の配列の要素について，定義，参照が繰り返し行われる場合は，時系列上の定義点と参照点を求めることが必要である．この時系列上の定義点を d, 次の定義点を d' とすると，d と d' の間に存在する参照点 $r_1, r_2, \ldots r_n$ では，定義点 d で更新された値が参照される．本手法では，参照点 r_1 では，データの受信，受信したデータのバッファへの保管，バッファからのデータの読み出しが行われるが，それ以降の参照点 $r_2 \ldots r_n$ では，バッファからのデータの読み出しのみが行われるように最適化を行う．

3　データフロー解析

配列要素のデータフロー解析として，データ依存解析を拡張し，LWT(Last Write Tree) というデータ構造でデータフローを表現する手法 [3] を採用する．この手法では，ループネスト内の定義，参照関係が求められる．本章では以下，ループネスト内の解析について議論する．

3.1　LWT の計算

本節では，LWT の求め方 [3] について述べる．

```
for i = 1, n do
  for j = 1, n do
    ... = a(i, j-1)
    a(i, j) = ...
  endfor
endfor
```

図 2: 1 対の定義，参照をもつループ

図 2の例では，参照点のイタレーションを (i^r, j^r)，定義点のイタレーションを (i^w, j^w) とすると，配列 a の参照に対する定義点は以下のようになる．

- $j^r = 1$ のとき，参照される値はループに入る前に定義される $(\perp)$．
- $j^r = 2 \ldots n$ のときは，イタレーション $(i^r, j^r - 1)$ で定義された値が参照される．この依存は内側ループによって運搬される．

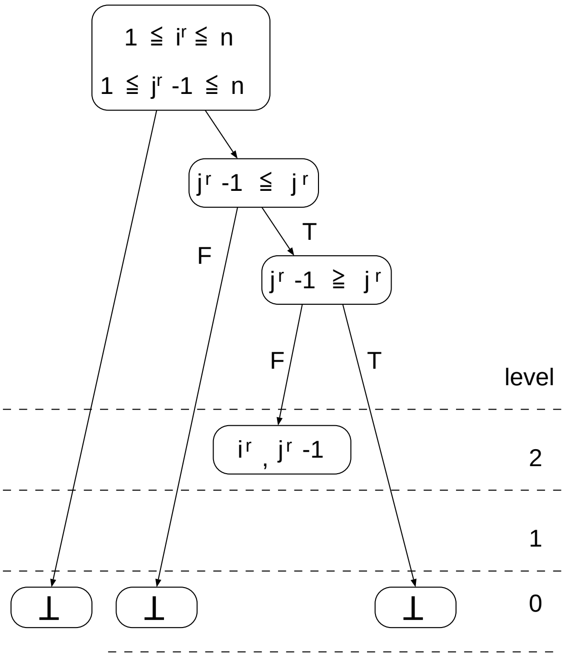

図 3: LWT

このコードの LWT を図 3に示す．LWT では，ある参照点と，それに対応する定義点ついて，定義点のイタレーションが，参照点のイタレーションに現れる変数 i^r, j^r を用いて表される．同時に，ループネスト内での依存レベルも表される．

依存レベル (dependence level) とは，イタレーション間で依存を運搬する場合の，依存を運搬する最も外側のループのネストレベルのことをいう．図 2では，依存レベルは内側ループのネストレベルである 2 となる．

外側のループで運搬されるとすると，依存レベルは 1 であり，定義点がループの外にある場合 ($\perp$の場合) は依存レベル 0，ループ運搬の依存がない場合は ∞ でとなる．

一般に LWT は次のように求められる．

n 重ループについて，定義点のイタレーション $\vec{i^w} = (i^w_1, i^w_2 \ldots, i^w_n)$ と，参照点のイタレーション $\vec{i^r} = (i^r_1, i^r_2 \ldots, i^r_n)$ を考える．

また，参照，定義イタレーション $\vec{i^r}, \vec{i^w}$ から，参照，定義される配列の要素を求める関数をそれぞれ F_r,

F_w とすると，依存がある場合には $F_w\vec{i^w} = F_r\vec{i^r}$ を満足する $\vec{i^w}, \vec{i^r}$ が存在する．

1. $\vec{i^w}$ を $\vec{i^r}$ を用いて表す．
2. イタレーション $\vec{i^w}$ のループの実行範囲から，$\vec{i^r}$ の制約を求める．この制約を満足しない参照点 $\vec{i^r}$ に対応する定義点は $\perp$ となる．
3. ループのネストレベル $l{=}1...n$ について，以下を繰り返す．
    - $i^w_l > i^r_l$ ならば，定義が参照より後になるので，定義点は $\perp$ である．
    - $i^w_l < i^r_l$ ならば，依存があり，その定義点は $\vec{i^w} = F_w^{-1}F_r\vec{i^r}$ となる．
    - $i^w_l = i^r_l$ ならば，もう 1 段深いネスト $l+1$ について同じことを繰り返す．
4. すべてのネストレベル l で $i^w_l = i^r_l$ が成り立つときは，最内ループのボディにある定義と参照がコード上でどちらが先に現れるかで依存が決まる．定義が先にであれは依存があり定義点は $\vec{i^w} = \vec{i^r}$ でとなり，参照が先であれば依存がなく，定義点は $\perp$ となる．

図 2の例では，定義点と参照点の間に，$(i^w, j^w) = (i^r, j^r - 1)$ の関係がある．

まず，定義点のループの実行範囲 $1{\le}i^w = i^r{\le}n$, $1{\le}j^w = j^r{-}1{\le}n$ を考えると，後者より，$2{\le}j^r{\le}n{+}1$ となり，$j^r = 1$ の場合は対応する定義点がループ内に存在せず未定義 $(\perp)$ となる．これは，ループ実行前に代入された値が参照されることを意味する．

次に，外側ループに対応する分岐ノード，$i^w{\le}i^r$, $i^w{\ge}i^r$ が入るが，これらは常に真であるため，図 3の LWT にはノードを加えていない．

内側ループについては，$j^w = j^r{-}1{\le}j^r$ は常に真となり $j^w = j^r - 1{\ge}j^r$ は常に偽である．ゆえに，ノード $(i^r, j^r - 1)$ が定義点となる．定義点から参照点への依存は内側ループ（レベル 2）で運搬されるため，図 3のレベル 2 の行に定義点のノード $(i^r, j^r - 1)$ が加えられる．

3.2　複数の LWT のマージ

我々の提案する手法では，まず，ある通信を伴う参照点を選び，対応する定義点を求める．この定義点に対応する通信を伴う参照点をすべて選び，実行順にならべることで，定義，参照の連鎖が構成できる．

この連鎖で各参照点に対応する LWT を並べ，各ループネストレベルでの定義，参照の連鎖が構成される．

4　メッセージ合体化の実現

本章では，前章で述べたデータフロー解析の結果を利用して，メッセージ合体化を行うための，コンパイル時の最適化と実行時に必要な処理について述べる．

メッセージ合体化を定義点と参照点のネストレベルが一致する場合と，一致しない場合に分けて実装を行う．ネストレベルが一致する場合は実行時ランタイムルーチンによる動的なメッセージ合体化（これをメッセージ・キャッシングと呼ぶ）を行う．一致しない場合，すばわち定義点が参照点を含むループネストの外にある場合は，通信コード移動と一時変数を利用した静的メッセージ合体化を行う．

4.1　メッセージ・キャッシング

4.1.1　基本原理

参照側 PE で過去に受信したデータを，実行時送受信ランタイム・ルーチンが管理する一時記憶領域（以後，単に受信バッファと呼ぶ）に保管しておく．同一データに対する受信要求に対し，受信バッファに有効なデータが存在すれば，受信バッファからデータを供給することで PE 間通信を削減することができる．どの受信データをいつまで保管しておくか，また，保管したデータの有効性をどのように保証するかを実装時の方針として決定しなければならない．

4.1.2　受信バッファ管理

3章で述べた LWT を解析することで，どのデータが複数回参照されるか，また，そのデータに対する定義点と最後の参照点がどこかを判定することができる．これらの情報をもとに受信バッファの割当て，解放，さらにはデータの一貫性保証（再定義が行われる場合）を行う．

そのために，受信 PE のバッファには，保管されたデータに対してその有効性を示す有効フラグ，送

信PEにはデータ定義後にデータが送信されたかどうかを示す送信フラグがそれぞれ用意される．

なお，以下の説明は，同期型のsend/recvを用いて実装することを前提としているため，送信側にも受信バッファのフラグに対応する送信フラグが必要となる．

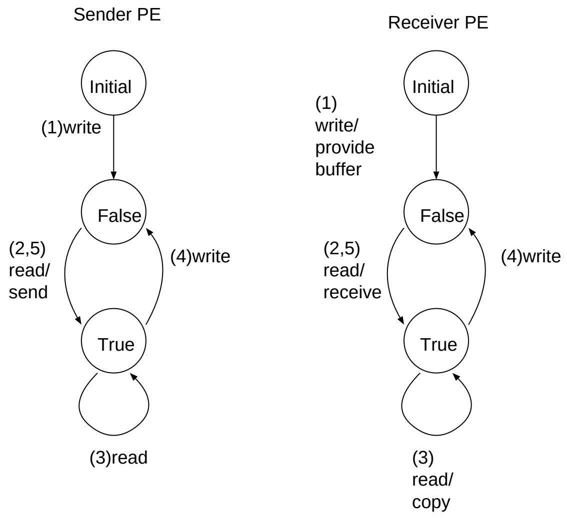

図4: バッファ制御アルゴリズム

受信バッファは図4のように制御される．以下にその手順を示す．

1. 送信PEで初めてデータの定義が起こると，送信フラグをFalseに，受信PEでは受信バッファを用意し，有効フラグをFalseにする．
2. 受信PEでデータの参照が起こると，送信PEはデータを送信し，送信フラグをTrueとする．受信PEはデータを受信し，有効フラグをTrueとし，データを受信バッファに保存し，さらに受信バッファからデータが読み出し出される．
3. 引き続き受信PEでデータの参照が行われる間，受信PEに用意された受信バッファからデータが読み出される．
4. 再び送信PEでデータの定義が起こると，送信PEでは送信フラグをFalseにする．受信PEはそのデータに対応する受信バッファの有効フラグをFalseにし，受信バッファのデータを無効とする．
5. 受信PEで再度データの参照が起こると，送信PEは新しく更新されたデータを送信し，送信フラグをTrueにする．受信PEは通信によって新しいデータを受信して受信バッファに保存し，有効フラグをTrueにし，さらに受信バッファからデータが読み出される．

受信バッファとそれに付加される有効フラグ，送信フラグは，不要となったときに削除することができる．これは，データフローをたどることにより，一番最後の参照点（それ以降，参照も定義も行われない）を求め，その参照終了後に受信バッファと有効フラグ，送信フラグを削除可能なことが保証される．これにより受信バッファによるメモリ領域の圧迫が避けられる．

ループネスト中に`if`文などの制御文が入った場合でも，データフローが正確に求められなくとも，参照点と定義点の対応関係のみを求め，定義があれば送信/有効フラグをFalseに，フラグがFalseのときに参照があれば送受信行って送信/有効フラグをTrueとすればよい．

受信バッファの有効フラグ，送信PE送信フラグは，配列の各要素毎に1つのビットを与えると，各配列要素のPE間での送受信の状態を正確に表現することができる．しかし，このビット配列は大きなテーブルになり，実行時のテーブルへの参照，定義のコストが大きくなる．そこで，配列の各次元について，下限l，上限u，ストライド幅sの三つ組$[l:u:s]$で矩形領域を表現し，その矩形領域について受信バッファ，フラグを用意する．

実行時に一括して管理する領域の大きさが変化した場合は領域を分割し，それぞれに対応して新たなビットを用意する．

受信バッファを，ソフトウェアによって制御されるキャッシュエリアと考えると，コンパイラの解析により，値の書き込みが行われた時点で制御信号を発生させずにキャッシュエリアの無効化を行っていることになる．

4.2 静的メッセージ合体化

ループネスト内の通信を伴うデータ参照の定義点がループ外に存在する場合であり，一時変数を用意して，データの受信をループネスト外に移動し，ループネスト内ではデータへの参照を一時変数への参照

に置き換える．ただし，通信されるデータに対するメッセージベクトル化可能な場合，一時変数として配列変数を用意する必要がある．

このコード変換は，通信を伴う参照が，通信とデータの読み出しとに明確に分離できる場合に，一時変数を用意することで，実行時ライブラリの領域の確保，解放のオーバヘッドを減少させることを可能とするものである．

```
for i = 1, n do
  for j = 1, n do
    for k = 1, n do
      c(i,j) = c(i,j) + a(i,k) * b(k,j)
    endfor
  endfor
endfor
```

⇓並列化による通信コードの挿入

```
for l_i = 1, n/P do /* P: PE 数 */
  for j = 1, n do
    for k = 1, n do
      send b(k,j) to owner(a(i, j))
      recv tmp_b = b(k,j)
      c(l_i,j) = c(l_i,j) + a(l_i,k) * tmp_b
    endfor
  endfor
endfor
```

⇓通信コードの移動

```
for pe=1, P (pe ≠ p) do
  send b(n(p-1)/P+1..n(p-1)P,1..n) to pe
  recv tmp_b(n(pe-1)/P+1..n(pe-1),1..n) from pe
endfor
for l_i = 1, n/P do
  for j = 1, n do
    for k = 1, n do
      c(l_i,j) = c(l_i,j) + a(l_i,k) * tmp_b(k,j)
    endfor
  endfor
endfor
```

図 5: 行列積における通信コードの移動

図 5の行列積の例では，配列 `a`, `b`, `c` をすべて第 1 次元で分割して各 PE に分散配置する場合を考える．すると，配列 `b` のうち，PE にローカルに配置されている行以外のすべての要素を通信によって求める必要がある．配列 `b` は，この行列積のループでは値が更新されないため，通信コードはすべてループの外へ移動させることができる．

5 おわりに

本報告では，重複するデータの送受信を削減するするための，重複するデータ送受信を検出するためのデータフロー解析手法と，受信したデータを再利用する動的／静的メッセージ合体化について述べた．

現時点では，ループ内に定義点が存在しない場合に，静的メッセージ合体化ルーチンが実装済みである．これを用いて問題サイズ 128×128 の図 5のコードを並列化し，並列計算機 AP1000 上で実行したところ，メッセージ合体化を行わなかった場合に比べ 4 倍程度の高速化が得られた．

今後は，さらに詳細なデータフロー解析として，ループネスト内に if 文などの制御文が入った場合，ループネスト間の解析について検討し，並列化コンパイラ TINPAR に対して実装を行いその有効性を検証する予定である．

謝辞

並列計算機 AP1000 の実行環境を提供して頂いた (株) 富士通研究所に感謝致します．また，日頃より有益な御意見をいただく富田研究室の諸氏に感謝致します．

参考文献

[1] High Performance Fortran Forum. High Performance Fortran language specification (ver. 1.0). Technical Report CRPC–TR92225, Rice University, May 1993.

[2] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Evaluating compiler optimizations for fortran D. *J. Parallel and Distributed Computing*, Vol. 21, pp. 27–45, 1994.

[3] Dror E. Maydan, Saman P. Amarasinghe, and Monica S. Lam. Array data-flow analysis and its use in array privatization. In *20'th Annual ACM Symposium on Principles of Programming Languages*, pp. 2–15, January 1993.

[4] 三吉郁夫, 前山浩二, 後藤慎也, 森眞一郎, 中島浩, 富田眞治. メッセージ交換型並列計算機のための並列化コンパイラ TINPAR. 情報処理学会論文誌, Vol. 37, No. 7, pp. 1265–1275, July 1996.