スレッドベース実行における積極的データ転送のための Plan-Do 型コンパイル技法とその評価

八杉 昌宏[†] 松岡 聡[††] 米澤 明憲[†††]

分散メモリ型並列計算機において積極的データ転送を行うための新しいコンパイルの枠組みとして，*Plan-do* コンパイル技法を開発した．この技法は近年の細粒度アーキテクチャにて，高スループット低遅延の通信手法 —パイプライン送信— を実現するときに特に有効である．変換関数を先頭から適用することで，高レベルの Plan-Do 型コードから，より低レベルの積極的データ転送を行うコードへ変換できる．また，開発した ABCL/ST コンパイラを用いて並列計算機 EM-4 における実験を行い，よい性能が得られることを確認した．

The Plan-Do Style Compilation Technique for Eager Data Transfer in Thread-Based Execution and its Evaluation

Masahiro Yasugi,[†] Satoshi Matsuoka[††] and Akinori Yonezawa[†††]

Plan-do compilation technique is a new, advanced compilation framework for *eager data transfer* on distributed-memory parallel architectures. The technique is especially effective for a recent breed of fine-grain architectures by realizing a high-throughput low-latency communication scheme, *pipelined sends.* The compilation of high-level, plan-do style code into low-level, eager data transfer code is achieved via straightforward application of the translation function. Benchmark results on a real parallel architecture, EM-4, with the developed ABCL/ST compiler exhibit good performance.

1. はじめに

分散メモリ型超並列計算機は，スケーラビリティの面で魅力的である．様々なタイプの分散メモリ型超並列計算機が考えられるが，その基本的な抽象実行モデルは，非同期に直接交信するスレッドというモデルといえる．そのような抽象マシンは，スレッドベースの抽象マシンにメッセージパッシングのためのルーチンを加えてやることでとりあえずは実現できる．そのようなルーチン集はメッセージパッシングライブラリと呼ばれている．

しかしながら，そのようなライブラリに基づくアプローチでは，通信のための，より進んだ実装技術を取り入れることができない．コンパイルに基づくアプローチによってのみ可能となる技術がそれである．例えば，メッセージ解釈をコンパイルする技術があり，アクティブメッセージ[7]として知られている．そこでは，メッセージにはフォーマットに関する実行時タグを付けず，コンパイルされたメッセージハンドラのアドレスのみを付けてやる．さらに進んだ実装技術には，通信をもコンパイルすることではじめて可能となるものがある．

本稿では，積極的(eager)データ転送という実装技術に着目し，そのコンパイルの枠組について述べる．積極的データ転送という考え方は，データフロー実行モデルに由来しているが，我々はそれをスレッドベース実行モデルにおいて用いる．積極的データ転送とは，「データが使われる場所にデータを積極的に送る」といい直すことができ，送り手，受け手双方におけるメッセージのローカルなバッファリングを減らすことで，スループット，遅延とも改良できる．もちろん，積極的

† 神戸大学工学部情報知能工学科
Department of Computer and Systems Engineering, Faculty of Engineering, Kobe University
†† 東京工業大学大学院情報理工学研究科数理・計算科学専攻
Department of Mathematical and Computing Sciences, Graduate School of Information Science and Engineering, Tokyo Institute of Technology
††† 東京大学大学院理学系研究科情報科学専攻
Department of Information Science, Faculty of Science, The University of Tokyo

データ転送の過度の使用にはトレードオフがあり，対象とした分散メモリ型並列計算機が細粒度通信を得意としない場合には，逆にオーバーヘッドが大きくなる．通信と待ち合わせの回数の増加を意味するからである．しかし，細粒度通信を得意とする並列アーキテクチャでは，論理的に一回のメッセージ送信に対しても積極的データ転送を有効とすることができる．本稿では，細粒度ハイブリッド並列アーキテクチャEM-4[3),5)]上での性能測定を通じて，コンパイルしたパイプライン送信におけるその有効性を示す．

一口に「直接交信するスレッド」といっても，様々なレベルの抽象化が可能である．例えば，ABCL[11)]のような並列オブジェクト指向言語は，高いレベルの抽象化を提供しており，一方，分散メモリマシンそれ自身は，最も低いレベルの抽象化を提供している．本稿では，コンパイル過程を，直接交信スレッドモデルにおける抽象度の段階的引き下げと捉え，コンパイラとして何が必要かを議論する．

積極的データ転送を実現するには，データフロー情報が不可欠となる．このため原理的には，すべての抽象化レベルでデータフロー情報が必要となる．つまり，コンパイラの各レイヤのモジュラリティを保つには，各段階の中間コードがデータフロー情報を持つ必要があり，データフロー情報に基づく最適化を行わないレベルですらデータフロー情報が必要になってしまう．このため，いくつかのコンパイルのフェーズをまとめて，1 パスにすることが難しくなる．この問題は，制御フローとデータフローをストリームの形に表した Plan-Do 型中間コード[10),14)]を使うことで解決できる．さらにいえば，コンパイルのフェーズ間のパイプライン並列化が望まれるときも便利な中間コードとなる．

2. 積極的データ転送

我々の動機付けをはっきりさせるために，ここでは，積極的データ転送のいくつかの例を用いて，いろいろな抽象化レベルでそれを活かせることをみる．

2.1 細粒度の例 — パイプライン送信

通信と計算をオーバーラップさせる（遠隔）メッセージパッシング実装方式として，パイプライン送信は，EM-4 上の並列オブジェクト指向言語処理系 ABCL/EM-4[8),9),12),13)] において提案されてきた．この方式はメッセージ引数の評価とその送信をパイプライン的に行う．これにより，送り手受け手双方におけるローカルなバッファリングを減らし，スループット，遅延とも改良できる．この方式は，EM-4 で効率良くサポートされている (1) 遠隔コードの起動，(2) 遠隔書き込み，(3) ネットワークの FIFO 性，により実現される．

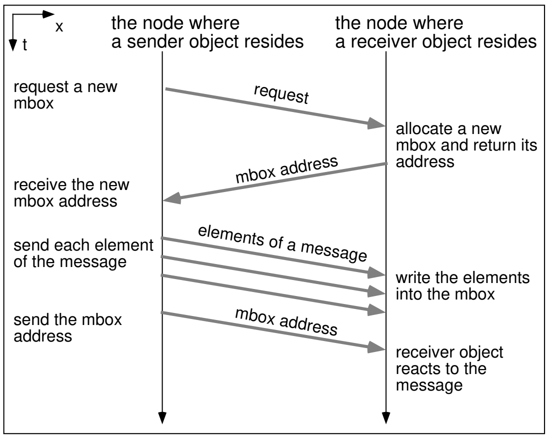

図 1 要求メッセージ送信のためのパケット交換
Fig. 1 Packet Exchange for a Pipelined Request Message Send

パイプライン送信のためのパケット交換手順は次のように進む（図 1）: (1) 送り手は，受け手ノードで割り当てられたメッセージボックスアドレスを返すコードを起動することで，受け手ノードにメッセージボックスを予約する．(2) 送り手は，メッセージ引数を評価し，それぞれの引数毎に，そのパケットを受け手ノード上のメッセージボックスの適切な位置に向けて送信する．評価と送信は細粒度にパイプライン化される．(3) 送り手は，受け手にそのメッセージボックスアドレスを送信する．受け手は，そのメッセージを処理できるまではキューに入れるかもしれない．

メッセージボックスを予約するためのパケットの伝達には，往復の遅延を要するが，送り手ノードにおけるマルチスレッディングにより効率的に隠蔽できる．さらには，予約により次の利点が得られる: (1) 受け手ノードにおけるメッセージキューの管理が，メッセージをまるごとコピーするのではなくポインタ操作のみで効率的にできる．(2) 送信先のメッセージボックスアドレスがわかっているので，パイプライン送信が可能になり，(a) メッセージの評価完了前の積極的送信による遅延の削減，(b) 送り手ノードにおけるローカルなバッファリングに要するメモリアクセスの削減，が達成される．さらに，パイプライン送信は中断/再開可能であり，他の送信とインターリーブできる．

2.2 より粗粒度の例

積極的データ転送はより粗粒度のレベルにおいても利用できる:

```
[Obj1 <= [:Msg1 [:cons 1 [:cons 2 [:nil]]]]]
```

並列オブジェクト指向言語 ABCL によるこのプログラム断片は，Obj1 へのメッセージ送信を表す．メッセージは，メッセージタグ:Msg1 と整数のリスト 1，2 である．これが遠隔メッセージ送信であり，コンスセルを用いてリストを実装すると仮定すると，ローカルにコンスしてから遠隔コピーするのではなく，直接 Obj1 がいるノードに遠隔コンスするコードを生成したい．

この種の積極的データ転送は何も並列オブジェクト指向言語に限った話ではなく，例えば（スレッドベースの）関数型言語では次のようになるであろう:

```
(Fun1 [:cons 1 [:cons 2 [:nil]]])
```

ここで，直接 Fun1 の駆動フレームがあるノードに遠隔コンスするコードを生成したい．

3. 積極的データ転送のためのコンパイルの枠組

より低レベルでの積極的データ転送を実現するには，高レベルコードはそれなりのデータフロー情報を持っていなくてはならない．データフロー情報は（同一レベル内での）コード最適化に必要なのではなく，コード変換（高レベルコードからの，より低レベルコードの生成）に必要となる．その情報には（ある値 v を考えると），(1) v がそのうち使われるかどうか，(2) どのスレッド（ノード）が v を使うことになるか，(3) どのメモリ位置に v は最終的に書き込まれるか，などがある．(2) と (3) が特に，2.1節で述べたパイプライン送信を実現するのには不可欠である．

3.1 Plan-Do スタイルにデータフロー情報を埋め込む

Plan-Do 型中間コード[10),14)]は上記の要件を満たすとともに，コンパイルの過程そのものに対する利点も有する．この節では簡単に Plan-Do 型コードについて述べるが，その重要な効果（つまり，積極的データ転送コードの生成）は，Plan-Do 型コードから，より低レベルのコードを生成するときに現れる．これは後の 3.2節で示す．

Plan-Do スタイルの基本的なアイデアは，(1) Plan 部（プラン宣言部）がデータフロー情報を提供し，(2) Do 部（プラン実行部）が制御フロー情報を提供することである．Plan-Do スタイルを用いることの利点は: (1) 任意のグラフ構造でなくストリームの形をとること，(2) プラン宣言とプラン実行をインターリーブすることができること，である．こうして，Plan-Do スタイルにより，Plan-Do 型コードを出力とするフェーズと，その Plan-Do 型コードを入力とするフェーズを，同じパスにする（または，パイプライン的並列化する）ことができる．

Plan-Do 型コードに適したコンパイルの対象としては，自然なデータフローを表している式（expression）がある．Plan-Do 型コードを用いることで，時間のかかるフロー解析を行わずに，その情報を表現することが可能になる．Plan-Do 型コードは，ソースプログラムの意味を制御スレッド（の逐次実行）の点から表すとともに，高レベルコードのマシン独立性を失うことなく，データフロー情報をも提供する．

簡単なメッセージ送信の例を用いる:

```
[obj1 <= [(+ i 1) (+ j 2)]]
```

ABCL 言語でユーザが記述したこのプログラム断片は，obj1 への要求メッセージ送信を表しており，そのメッセージは整数 i+1 と j+2 のタプルである．このソースプログラムは高レベル構文木を通して，次の Plan-Do 型コードに変換される:

```
(newplan (p1 d1 d2) (request-send))
(throw obj1 d1)
(newplan (p2 d3 d4) (make-tuple d2))
(newplan (p3 d5 d6) (arith3-+ d3))
(throw i d5)
(throw 1 d6)
(do p3)
(newplan (p4 d7 d8) (arith3-+ d4))
(throw j d7)
(throw 2 d8)
(do p4 p2 p1)
```

(newplan (p1 d1 d2) (request-send)) により，プラン p1 をディスティネーション d1，d2 とともに宣言する．ここでディスティネーションとは，プランの引数であり，データフローモデルにおける（プラン）ノードへのデータフロー枝に相当する．ここでのプランは，要求メッセージをターゲットオブジェクトへ送信することである．ターゲットオブジェクトの名前は，その名前を d1 に ‘throw’ することで与えられる．‘throw’ とは，ある値をプランの引数として渡すこととする．(throw obj1 d1) により obj1 の値を d1 に ‘throw’ する．(do p1) によりプラン p1 を実行する．同様に，要求メッセージの実引数は，他のプランの実行により得られた値（つまり，i+1 と j+2 からなるタプル）を d2 に ‘throw’ することにより与えられる．

(newplan (p2 d3 d4) (make-tuple d2)) により，「(1) d3 と d4 に ‘throw’ された値からタプルを構成し，(2) d2 にそのタプルを ‘throw’ する」というプラン p2 を宣言する．(newplan (p3 d5 d6) (arith3-+

d3)) により「(1) d5 と d6 に 'throw' された値を足し，(2) その和を d3 に 'throw' する」というプラン p3 を宣言する．

Plan-Do 型コードの標準的な（当たり前の，積極的データ転送をしない）解釈は次のようになる: (1) プランの宣言は単に宣言として解釈され，何の実行もしない．ディスティネーションは一時変数として解釈される．(2) 'throw' は対応する一時変数への代入と解釈される．(3) プランの実行はプランの実際の実行として解釈される．3.2節では，別の解釈として積極的データ転送のための解釈を示す．

構文木から，Plan-Do 型コードへの変換は非常に簡単である: （再帰的）変換関数が，ある（構文木の）ノードに到達したとき，その部分木の変換をする前に，そのノードに関するプランを宣言し，その部分木の変換後にプランを実行すればよい．このように，構文木からの Plan-Do 型コードの生成と，その Plan-Do 型コードからの積極的 (または，積極的ではない) データ転送コードの生成の，二つのフェーズに分けることで，Plan-Do 型コードを生成するコンパイルフェーズはアーキテクチャに非依存（積極的データ転送の有無に依らない）とすることができる．

3.2 Plan-Do 型コードの別解釈による積極的データ転送コードの生成

この節では，高レベル Plan-Do 型コードを解釈して，積極的データ転送を実現するためのコンパイルの過程を述べる．2.1節で述べたパイプライン送信実現の過程を，前節の簡単なメッセージ送信の例を用いて例示する．

図 2 に，高レベルコードからより低レベルのコードへの変換関数 Tr を示す．$Tr[\![hcode]\!](penv, denv)$ は，$hcode$ を高レベル Plan-Do 型中間コードとすると，より低レベルのコードを，プラン環境 $penv$ とディスティネーション環境 $denv$ のコンテキストのもとで返す．$penv$ は，プラン識別子（p で表す）から，プランへの写像であり（p は，データフローモデルにおけるデータフローノードに相当），$denv$ は，ディスティネーション識別子（d で表す）から，p と $i(\in \mathbf{N})$ のペアへの写像である（d は，データフローモデルにおける p への i 番目のデータフロー枝に相当）．補助関数 $Trnsfr[\![v_l, fl, d]\!](penv, denv)$ は，低レベルのコードの断片を返す．その断片は 値 v_l をディスティネーション d の fl フィールドに転送する．$Trnsfr$ の結果は，ある条件下では積極的データ転送命令を選択するため，データフローコンテキスト $(penv, denv)$ に影響される．また図中，関数 f に対して，"$f\{x \mapsto y\}$" により，$Dom(f') = Dom(f) \cup \{x\}$ かつ $f'(x) = y$ かつ すべての $x' \in Dom(f) - \{x\}$ について $f'(x') = f(x')$ となるような関数 f' を表す．"::" は，"$h :: r$" としたときヘッド h とリスト r のコンスを表す．"$[a, b, c]$" は，a, b, c からなるリストを表す． "@" は，"l_1 @ l_2" としたとき二つのリスト l_1, l_2 の連結を表す．

この変換関数は，基本的には高レベルコードを次のようにして変換する:

P1,P2,P3 プラン宣言については，プランとディスティネーションを環境 $penv$，$denv$ の中に記憶する．必要に応じて，新しい (fresh な) 低レベル変数識別子を導入する．

D1,D2,D3 プラン実行については，プランを実行完了するためのコードを生成し，まだ結果の値が積極的に送られていない場合は，関数 $Trnsfr$ を呼び出して，データ転送命令を生成する．

TH 値のディスティネーションへの 'throw' については，$Decomp$ を呼び出して高レベルの値をより低レベルの変数とフィールド数リストのペアの集合に分解した後，$Trnsfr$ を呼び出してデータ転送命令を生成する．

Tr 及び補助関数 $Trnsfr$ により，積極的データ転送コードへの変換，つまりパイプライン送信が実現される．このため Tr は高レベルコードを次のように変換する:

P1,T1,T1',D1 要求メッセージ送信については，パイプライン送信を実現するメッセージボックスのためのポインタを準備し，ポインタ操作に関する命令だけを生成する．

P2,T2,D2 タプル構成については，タプルの要素がすべて計算されるまでバッファリングすることなく，タプルの個々の要素を積極的に転送する．

P3,T3,D3 算術演算については，単に三つの一時変数を用意して，演算結果を $Trnsfr$ を用いて転送する．

$Trnsfr$ はディスティネーション d へのデータ転送を次のようにして実現する:

T1 d が要求メッセージ送信のターゲットを指定するためのものならば，単に値をそのための一時変数に代入する．ただし，その直後に，パイプライン送信のためのメッセージボックスを予約する．

T1' d が要求メッセージ送信のメッセージを指定するためのものならば，パイプライン送信のための「遠隔書き込み」命令を生成する．

T2 d がタプル構成のためのものならば，$Trnsfr$ を再帰的に呼ぶことで，d のタプル内の特定のフィー

(P1) $Tr[\![\texttt{(newplan (}p_1\ d_1\ d_2\texttt{) (request-send))} :: r]\!]\,(penv, denv) =$
$Tr[\![r]\!]\,(penv\{p_1 \mapsto \mathbf{request\text{-}send}(t_1, t_2, t_3)\}, denv\{d_1 \mapsto (p_1, 1), d_2 \mapsto (p_1, 2)\})$
$(t_1,\ t_2,\ t_3$ fresh$)$

(P2) $Tr[\![\texttt{(newplan (}p_1\ d_1\ d_2\texttt{) (make-tuple }d_3\texttt{))} :: r]\!]\,(penv, denv) =$
$Tr[\![r]\!]\,(penv\{p_1 \mapsto \mathbf{make\text{-}tuple}(d_3)\}, denv\{d_1 \mapsto (p_1, 1), d_2 \mapsto (p_1, 2)\})$

(P3) $Tr[\![\texttt{(newplan (}p_1\ d_1\ d_2\texttt{) (arith3-+ }d_3\texttt{))} :: r]\!]\,(penv, denv) =$
$Tr[\![r]\!]\,(penv\{p_1 \mapsto \mathbf{arith3\text{-}+}(d_3, (t_1, t_2, t_3))\}, denv\{d_1 \mapsto (p_1, 1), d_2 \mapsto (p_1, 2)\})$
$(t_1,\ t_2,\ t_3$ fresh$)$

(D1) $Tr[\![\texttt{(do }p_1\texttt{)} :: r]\!]\,(penv, denv) =$
$\texttt{(get-read-pointer }t_2\ t_3\texttt{)} :: \texttt{(request-send }t_3\ t_1\texttt{)} :: Tr[\![r]\!]\,(penv, denv)$
if $penv(p_1) = \mathbf{request\text{-}send}(t_1, t_2, t_3)$

(D2) $Tr[\![\texttt{(do }p_1\texttt{)} :: r]\!]\,(penv, denv) = Tr[\![r]\!]\,(penv, denv)$ if $penv(p_1) = \mathbf{make\text{-}tuple}(d_1)$

(D3) $Tr[\![\texttt{(do }p_1\texttt{)} :: r]\!]\,(penv, denv) =$
$\texttt{(arith3-+ }t_1\ t_2\ t_3\texttt{)} :: (Trnsfr[\![t_3, \mathbf{nil}, d_1]\!]\,(penv, denv) @ Tr[\![r]\!]\,(penv, denv))$
if $penv(p_1) = \mathbf{arith3\text{-}+}(d_1, (t_1, t_2, t_3))$

(TH) $Tr[\![\texttt{(throw }v_h\ d_1\texttt{)} :: r]\!]\,(penv, denv) =$
$(Trnsfr[\![v_{l1}, \mathit{fl}_1, d_1]\!]\,(penv, denv) @ \cdots @ Trnsfr[\![v_{ln}, \mathit{fl}_n, d_1]\!]\,(penv, denv)) @$
$Tr[\![r]\!]\,(penv, denv)$
where $Decomp(v_h) = \{(v_{l1}, \mathit{fl}_1), \ldots, (v_{ln}, \mathit{fl}_n)\}$

(T1) $Trnsfr[\![v_l, \mathbf{nil}, d_1]\!]\,(penv, denv) = [\texttt{(assign }v_l\ t_1\texttt{)}, \texttt{(get-rqst-mbox-on }t_1\ t_2\texttt{)}]$
if $denv(d_1) = (p_1, 1)$ and $penv(p_1) = \mathbf{request\text{-}send}(t_1, t_2, t_3)$

(T1') $Trnsfr[\![v_l, \mathit{fl}, d_1]\!]\,(penv, denv) = [\texttt{(setarg-remote }v_l\ t_2\ \mathit{fl}\texttt{)}]$
if $denv(d_1) = (p_1, 2)$ and $penv(p_1) = \mathbf{request\text{-}send}(t_1, t_2, t_3)$

(T2) $Trnsfr[\![v_l, \mathit{fl}, d_1]\!]\,(penv, denv) = Trnsfr[\![v_l, (i-1) :: \mathit{fl}, d_2]\!]\,(penv, denv)$
if $denv(d_1) = (p_1, i)$ and $penv(p_1) = \mathbf{make\text{-}tuple}(d_2)$

(T3) $Trnsfr[\![v_l, \mathbf{nil}, d_1]\!]\,(penv, denv) = [\texttt{(assign }v_l\ t_i\texttt{)}]$
if $denv(d_1) = (p_1, i)$ and $penv(p_1) = \mathbf{arith3\text{-}+}(d_3, (t_1, t_2, t_3))$ (for $1 \le i \le 2$)

図 2 Plan-Do 型コードから，より低レベルのコードへの変換関数
Fig. 2 Translation Function from Plan-Do Style Code into Lower-Level Code for Pipelined Sends.

ルドに，積極的に値を転送する．

T3 d が算術演算のためのものならば，単に，対応する一時変数に値を代入する．

図 **3** は，`[obj1 <= [(+ i 1) (+ j 2)]]` に関して，高レベルコードから，より低レベルのコードへの変換結果を示す．左側のコードが高レベルコードであり，同じパス内で，構文木から，高レベルコードも，より低レベルのコードも生成される．積極的データ転送は，`(do p3)` と `(do p4)` のところに現れており，タプルの要素が積極的に遠隔ノード上の予約されたメッセージボックスに送信される．`(do p2)` が何もしないのは，タプルの要素はすでに積極的に送られているためである．

4. 性能測定

この節では，積極的データ転送の効果を並列オブジェクト指向言語処理系 ABCL/EM-4[9),12),13)] における性能測定によって示す．

ABCL/EM-4 のソース言語は静的に型付けされた ABCL，ABCL/ST である．我々は ABCL/ST の EM-4 用コンパイラを開発してきた[4),8)]．コンパイラでは「直接交信スレッド」の各抽象化レベルとして次の言語階層を採用した: (1) ソース言語，(2) 高レベル言語（ソース言語の意味を制御スレッド（の逐次実行）の点から表す），(3) 中レベル言語（実装のためのポインタの導入），(4) 低レベル言語（データサイズとメモリアドレスの概念の導入），(5) I 言語（フロー

```
 Plan-Do Style Code                           Lower-Level Code               Applied Translation
(newplan (p1 d1 d2) (request-send))                                              P1
(throw obj1 d1)                             (assign obj1-0 t1)                   TH - T1
                                            (get-rqst-mbox-on t1 t2)  [*1]
(newplan (p2 d3 d4) (make-tuple d2))                                             P2
(newplan (p3 d5 d6) (arith3-+ d3))                                               P3
(throw i d5)                                (assign i-0 t4)                      TH - T3
(throw 1 d6)                                (assign 1 t5)                        TH - T3
(do p3)                                     (arith3-+ t4 t5 t6)                  D3
                                            (setarg-remote t6 t2 (0)) [*2]          - T2 - T1'
(newplan (p4 d7 d8) (arith3-+ d4))                                               P3
(throw j d7)                                (assign j-0 t7)                      TH - T3
(throw 2 d8)                                (assign 2 t8)                        TH - T3
(do p4)                                     (arith3-+ t7 t8 t9)                  D3
                                            (setarg-remote t9 t2 (1)) [*3]          - T2 - T1'
(do p2)                                                                          D2
(do p1)                                     (get-read-pointer t2 t3)  [*4]       D1
                                            (request-send t3 t1)      [*5]
```

[*1] allocate a message box t2 on the node of t1, [*2] pipelined remote write of i+1,

[*3] pipelined remote write of j+2, [*4] complete the initialization of the message box,

[*5] send the message box address.

図 3 [obj1 <= [(+ i 1) (+ j 2)]] の生成された低レベルコード
Fig. 3 Generated Lower Level Code for [obj1 <= [(+ i 1) (+ j 2)]]

解析に基づく最適化用），(6) アセンブリ言語，である．Plan-Do スタイルは高レベルコードにマシン独立性を失うことなくデータフロー情報を付加するのに用いている．

コンパイラは，電総研で開発され，稼働中の細粒度ハイブリッド並列アーキテクチャEM-4[3),5)] のアセンブリコードを生成する．EM-4（プロトタイプ）は 80 個の要素プロセッサからなり 12.5 MHz のクロック速度で動作し，細粒度の通信メカニズムを提供する．例えば，レジスタ上から高速なオメガ網にデータを直接送出する 2 ワードパケット出力命令などがある．EM-4 のハードウェアはまたそのデータ駆動（パケット駆動）の特徴と組み合わせることにより，複数個のスレッドの基本的スケジューリングメカニズムも提供する．ハイブリッドというのは，制御フローアーキテクチャとデータ駆動アーキテクチャの融合を意味する．

一つ目の測定では，遠隔メッセージパッシングの二つの実装方式を比較する: (1) パイプライン送信（2.1 節参照）と (2) 非パイプライン送信（すべてのメッセージ引数を評価しおわるまで通信を開始しない），である．

遠隔メッセージ送信の遅延を測定するために，測定用プログラムはオブジェクトのチェインを生成し，そのチェインに沿ってメッセージを送る: オブジェクト O_{i-1}からのメッセージ M_iによって活性化した O_iは，EM-4 の隣ノード上の O_{i+1}に M_{i+1}を送る．

隣り合うオブジェクト O_i，O_{i+1}間の平均の活性化間隔を，メッセージとして送信するタプルのサイズを変えて測定した結果を図 **4** に示す．図から分かるように，パイプライン送信（図中 Pipelined Send と表示）は非パイプライン送信（図中 Bufferd Send と表示）より常に良い結果が得られた．主な原因は非パイプライン送信では，メッセージ引数の送り手ノードでの余分なメモリアクセスが必要となるからである．このバッファリングはパイプライン送信では必要なく，レジスタ上の評価された値はそのままネットワークへと送られる．

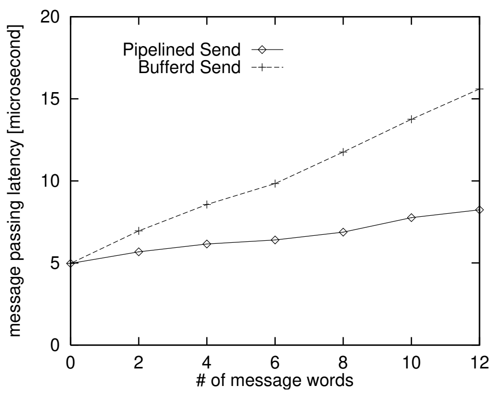

図 4 メッセージ送信における積極的データ転送の効果
Fig. 4 The effect of eager data transfer in message passing

図 4 にみられる遅延には，通信遅延のみならずオブジェクトが計算に要する実行時間も含まれている．にもかかわず，最小遅延は 0 ワードメッセージのときの約 5.0 μs（62 clock cycles）であり，12 ワードメッセージのときの非パイプライン送信とパイプライン送信との遅延の差は約 7.4 μs（92 clock cycles）であった．これらの値からパイプライン送信の重要度は，

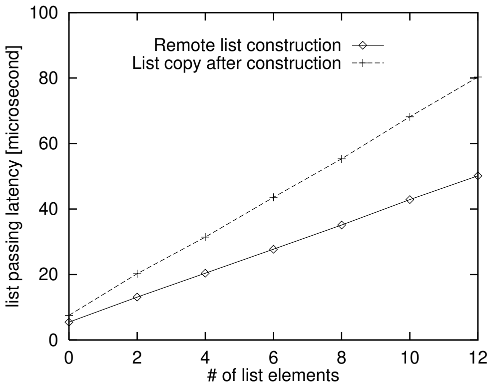

図 5 リスト送信における積極的データ転送の効果
Fig. 5 The effect of eager data transfer in list passing

EM-4 のように通信が高速なアーキテクチャではかなり高まることが分かる．

一方，二番目の測定として，リストの形のメッセージを送信した場合の結果を図 **5** に示す．積極的データ転送を行わない場合（図中 List copy after construction と表示），2.2節で述べたように，一旦，送り手ノードにリストを構成してからリストを遠隔コピーすることで送信を行うため，積極的データ転送を行って直接 EM-4 の隣のノードにリストを構成する場合（図中 Remote list construction と表示）より，12 要素のリストで 30μs（60%）も遅延が増加している．

5. 議　論

5.1 コンパイルに基づくメッセージパッシング

メッセージパッシングに，そのためのルーチン呼び出しを使った場合，呼び出しのオーバーヘッドを無視しても，呼び出されるルーチンが対象とするメッセージ全体を評価し終えている必要があるという問題が残る（多数のメッセージパッシングライブラリと同様に）メッセージをメモリに置いてから呼び出されるルーチンを用いた場合，メモリに書き出すためのオーバーヘッドが生ずる．このオーバーヘッドは，4節の測定結果からわかるように，EM-4 のような細粒度アーキテクチャでは無視できない．また，メッセージをレジスタに乗せてルーチンを呼び出すとした場合でも，メッセージを評価中に（特に他のメッセージパッシングが必要になるなどして），評価済みのメッセージデータを，一旦，レジスタから退避しなければならないことが多い．以上の点から，我々は，メッセージパッシングのためのルーチンではなく，コンパイルに基づいて，パイプライン送信のようなバッファリングを省いたメッセージ送信を実現している．

本稿では積極的データ転送を実現するために，Plan-Do スタイルの高レベルコードを用いることを提案しているが，これは，コンパイルに基づいてメッセージパッシングを行うのであれば，他の方法によっても実現はできる．例えば，一旦，高レベルコードから 3.1 節の最後で述べたような標準的な解釈に基づいて低レベルのコードを生成した後，そのコードを最適化することで実現することができる．しかし，Plan-Do スタイルを用いることで，この最適化の手間を省くことができる．

5.2 節度ある積極性

積極的データ転送の考え方は，データフロー実行モデルに由来する．データフロー実行モデル（つまり積極的評価モデル）では，非常にたくさんの細粒度の並列性が引き出せ，理想的には，実行の完璧な高速化が得られる．しかし，現実問題としては，並列度の爆発がしばしば，資源の枯渇やネットワークでの衝突などの問題を引き起こす．

一方、我々が対象としているスレッドベース実行モデルでは，並列度はスレッド数を制御するだけで簡単に抑えることができる．さらにいえば，それぞれのスレッドは，パイプラインやレジスタを使って効率的に実行できる．細粒度のゆるやかな（lenient）セマンティクスを持つ言語であっても，最近の TAM[2] のようなアプローチでは，コンパイラ補助の高効率のスレッドベース実行モデルに頼ることでこの利点を利用しようとしている．

我々のアプローチでは，最近のアーキテクチャの細粒度の能力を活かすために，直接交信スレッドモデルで通信をコンパイルするのに積極性を導入した．しかしながら，これは必ずしも積極的評価と同じ問題（並列度の爆発）を引き起こさない．我々のアプローチでは，積極性は積極的データ転送の形でのみ現れていて，データフロー実行モデルのように “発火” を含んでいない（データフロー実行モデルは，積極的データ転送+待ち合わせ+発火からなる）．我々のアプローチにおける積極性はやはりスレッド実行モデルによって節度あるものとなっている．

5.3 最適化技法との結合

もし最初からデータフロー情報に基づく最適化が高レベルにおいて必要ならば，3.1節で述べた Plan-Do 型コードの生成は必要なく，積極的データ転送を実現するには，3.2節で述べたより低レベルのコードへの変換のみ適用してやればよい．

例えば配列計算では，文献 1) は，データフロー情報に基づく最適化により，最適化された「直接交信ス

レッド」コードを生成する．ノンストリクト計算では，文献6)は，データフローグラフの分割により最適化された「直接交信スレッド」コードを生成する．いずれの場合も，EM-4のようなアーキテクチャでの細粒度の通信へのコンパイルが望まれるときは，スレッドにデータフロー情報を残しておくことにより，3.2節で述べた我々のコンパイル手法は，通信の部分に積極性を導入するのに利用することができる．

6. まとめ

我々は「直接交信スレッド」に基づく実行モデルで，高スループット低遅延の通信を実現するための，コンパイルに基づく新しいアプローチを示した．通信に関するより進んだ実装技術を活かすために，ライブラリベースのアプローチではなく，コンパイルに基づくアプローチを用いた．特に，積極的データ転送に着目し，そのコンパイルの枠組みを述べた．また，細粒度ハイブリッド並列アーキテクチャEM-4における性能測定を通じて，コンパイルされたパイプライン送信を使ったときの積極的データ転送の効果を示した．

積極的データ転送を実現するには，データフロー情報がコード変換に不可欠となる．コンパイラはデータフロー情報に基づく最適化を行わないレベルでもデータフロー情報を提供しなくてはならない．そこで，いくつかのコンパイルのフェーズを1パスにすることが難しくなる．我々の *Plan-Do* 型中間コードは制御フローとデータフローをストリームの形にまとめて表現することによりこの問題を解決した．

謝辞　電総研の山口喜教，坂井修一（現筑波大学），佐藤三久（現RWCP），児玉祐悦の各氏には，EM-4の使用に際して技術的な点を含む貴重な御助言をいただいた．また，東大の平木敬教授には有用なコメントをいただいた．ここに深謝する．

参考文献

1) Amarasinghe, S. P. and Lam, M. S.: Communication Optimization and Code Generation for Distributed Memory Machines, *Proc. of the SIGPLAN'93 Conference on Programming Language Design and Implementation*, pp.126–138 (1993).

2) Culler, D. E., Sah, A., Schauser, K. E., von Eicken, T. and Wawrzynek, J.: Fine-Grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine, *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 166–175 (1991).

3) Kodama, Y., Sakai, S. and Yamaguchi, Y.: A Prototype of a Highly Parallel Dataflow Machine EM-4 and its Preliminary Evaluation, *Proc. of InfoJapan '90*, pp. 291–298 (1990).

4) Matsuoka, S., Yasugi, M., Taura, K., Kamada, T. and Yonezawa, A.: Compiling and Managing Concurrent Objects for Efficient Execution on High-Performance MPPs, *Parallel Language and Compiler Research in Japan* (Bic, L. F., Nicolau, A. and Sato, M.(eds.)), Kluwer Academic Publishers, pp. 91–125 (1995).

5) Sakai, S., Yamaguchi, Y., Hiraki, K., Kodama, Y. and Yuba, T.: An Architecture of a Dataflow Single Chip Processor, *Proc. of the 16th Annual International Symposium on Computer Architecture*, pp. 46–53 (1989).

6) Traub, K. R., Culler, D. E. and Schauser, K. E.: Global Analysis for Partitioning Non-Strict Programs into Sequential Threads, *Proceedings of the ACM Conference on Lisp and Functional Programming*, pp. 324–334 (1992).

7) von Eicken, T., Culler, D. E., Goldstein, S. C. and Schauser, K.E.: Active Messages: a Mechanism for Integrated Communication and Computation, *The 19th Annual International Symposium on Computer Architecture*, pp. 256–266 (1992).

8) Yasugi, M.: *A Concurrent Object-Oriented Programming Language System for Highly Parallel Data-Driven Computers and its Applications*, PhD Thesis, Department of Information Science, The University of Tokyo (1994).

9) Yasugi, M., Matsuoka, S. and Yonezawa, A.: ABCL/onEM-4: A New Software/Hardware Architecture for Object-Oriented Concurrent Computing on an Extended Dataflow Supercomputer, *Proc. of 6th ACM International Conference on Supercomputing*, pp. 93–103 (1992).

10) Yasugi, M., Matsuoka, S. and Yonezawa, A.: The Plan-Do Style Compilation Technique for Eager Data Transfer in Thread-Based Execution, *Proc. of the IFIP WG10.3 International Conference on Parallel Architectures and Compilation Techniques*, pp. 57–66 (1994).

11) Yonezawa, A.(ed.): *ABCL: An Object-Oriented Concurrent System*, The MIT Press (1990).

12) Yonezawa, A., Matsuoka, S., Yasugi, M. and Taura, K.: Implementing Concurrent Object-Oriented Languages on Multicomputers, *IEEE Parallel & Distributed Technology*, Vol.1, No.2, pp. 49–61 (1993).

13) 八杉昌宏, 松岡聡, 米澤明憲: ABCL/onEM-4:

データ駆動計算機上の並列オブジェクト指向計算システムの高性能実装, 並列処理シンポジウム (JSPP'92) 論文集, pp. 171–178 (1992).
14) 八杉昌宏, 松岡聡, 米澤明憲: スレッドベース実行における積極的データ転送のための Plan-Do 型コンパイル技法, 情報処理学会研究報告 94-PRG-18(SWoPP'94), Vol.94, No.65, pp.9–16 (1994).

(平成 8 年 9 月 16 日受付)
(平成 9 年 7 月 4 日採録)

八杉 昌宏(正会員)

1967 年生．1989 年東京大学工学部電子工学科卒．1991 年同大学大学院工学系研究科電気工学専攻修士課程修了．1994 年同大学大学院理学系研究科情報科学専攻博士課程修了．1993 ~ 1995 年日本学術振興会特別研究員(東京大学，マンチェスター大学)．1995 年より神戸大学工学部情報知能工学科助手．博士(理学)．並列処理(並列システム，並列言語)，オブジェクト指向言語，言語処理系(コンパイラ，ランタイム) などに興味を持つ．日本ソフトウェア科学会，ACM 会員．

松岡 聡(正会員)

1963 年生．1986 年東京大学理学部情報科学科卒．1989 年同大学大学院博士課程中退．同年情報科学科助手，同大学情報工学専攻講師を経て，1996 年 10 月より東京工業大学情報理工学研究科助教授．理学博士．オブジェクト指向言語，言語処理系，並列システム，自己反映言語，ユーザ・インターフェースソフトウェアなどの研究に従事．ECOOP'97 のプログラム委員長をはじめ，各種学会のプログラム委員を歴任．ACM，IEEE-CS 各会員．

米澤 明憲(正会員)

1947 年生．1977 年 Ph.D. in Computer Science(MIT)．1989 年より東京大学理学部情報科学科教授．超並列・分散ソフトウェアアーキテクチャ，などに興味を持つ．共著書「算法表現論」「モデルと表現」(岩波書店)，編著書「ABCL: An Object-Oriented Concurrent System」(MIT Press)などがある．4 年間ドイツ国立情報処理研究所(GMD)科学顧問，ACM Transaction on Programming Languages and Systems 副編集長，IEEE Parallel & Distributed Technology 編集委員などを歴任，現在日本ソフトウエア科学会理事長．