静的解析による並列論理型言語KL1のメッセージ通信最適化

大野和彦[†] 伊川雅彦[†,☆] 森眞一郎[†]
中島浩[†,☆☆] 富田眞治[†]

並列論理型言語KL1で記述されたプログラムをメッセージ交換型並列計算機上で動作させる場合，個々のデータが参照されるごとに転送を行うため細粒度通信が頻発し，大幅な速度低下を生じる．そこで本研究では，静的解析により受信側で参照されるデータ型を抽出し，実行時に生成されるデータのうちその型を持つもののみを一括送信する最適化手法を提案する．本手法により，余分なデータ送信を生じることなくメッセージの粒度を上げ，通信オーバヘッドを削減できる．性能評価の結果，通信回数が大幅に削減され，相対的に通信負荷の高いプログラムや計算機環境では，実行時間も大きく改善されることが確認できた．

Efficient Message Communication of Concurrent Logic Programming Language KL1 Based on Static Analysis

KAZUHIKO OHNO,[†] MASAHIKO IKAWA,[†,☆] SHIN-ICHIRO MORI,[†]
HIROSHI NAKASHIMA[†,☆☆] and SHINJI TOMITA[†]

In the execution of concurrent logic language KL1 on message-passing multiprocessors, frequent fine-grained communications cause a drastic inefficiency. We propose an optimization scheme which achieves high granularity of messages by packing data transfer. Using static analysis, we derive data types which are required by the receiver process. With this information, each data of these types are packed into large messages. As a result of evaluation, the number of communications was considerably reduced. This effects to reduce the execution time of programs which have large communication overhead.

1. はじめに

並列論理型言語KL1[1),2)]は，第五世代計算機計画の核言語として開発され，動的データ構造や並列性を自然に記述できるという特徴を持っている．KL1を用いて，定理証明や遺伝子情報処理など，数々の知識処理プログラムが作成されてきた．また，最近では移植性の高いKL1処理系KLIC[3),4)]が開発されており，様々な並列計算機やワークステーション上で，これらのソフトウェア資産が利用可能になりつつある．したがって将来的には多様な並列計算機環境上で，KL1の記述能力を生かした並列プログラム開発が期待できる．

しかしながら現時点でのKLICの実装は，速度やメモリなど実行効率の面で手続き型言語をベースとした他の並列言語処理系に及ばず，実用的な処理系実現のためには実行方式の改良が欠かせない．とくにメッセージ交換型並列計算機を対象とした場合，頻発する細粒度通信のオーバヘッドが速度低下の大きな要因になる．これは，動的なデータ構造を扱うためデータ中の通信対象部分が実行時まで決定せず，必要になるたびに要求・返信という形で通信が行われるためである．この問題はプログラム記述の工夫によりある程度まで改善できるが，ユーザが物理的な通信を意識せずに記述できるというKL1の利点を生かすには，処理系側で自動的に最適化することが望ましい．

そこで本研究では，静的解析によりデータ構造中の通信対象となる要素や送信方向をコンパイル時に決定することによって，粒度の高い効率的な通信を行う一括送信手法を提案する．

本手法ではKL1プログラムを並列実行単位ごとに並列プロセスとして分割し，モード・型解析[5),6)]を適用して，生じるデータの型やデータフローを抽出する．

† 京都大学工学部
Faculty of Engineering, Kyoto University
☆ 現在，三菱電機株式会社
Presently with Mitsubishi Electric Corporation
☆☆ 現在，豊橋技術科学大学
Presently with Toyohashi University of Technology

このとき解析は各々に対して個別に行い，その並列プロセスが必要とするデータ要素だけを決定する．このため，プログラム全体に対し完全な解析を行うのに比べると，解析コストを減らすことができる．続いてこの情報を利用し，受信側が必要なデータのみを一括送信するコードを生成する．

以下，2章でKL1とKLICについて説明し，3章で本手法の概要を述べる．続いて4章および5章で，それぞれ静的解析および一括送信コードについて具体的に説明し，6章で性能評価について述べ，最後に7章でまとめを行う．

2. 背　景

2.1 並列論理型言語KL1の概要

2.1.1 プログラム構造

KL1はFlat GHCに基づく言語であり，プログラムは以下の形をした節（クローズ）の集合で表される．

$$H :- G_1,\ldots,G_m \mid B_1,\ldots,B_n.$$

H，G，Bは，それぞれクローズヘッド，ガードゴール，ボディゴールと呼ばれる．ゴールは述語の呼び出しであり，述語はヘッドの名前と引数の数が等しい節の集合で定義され，“述語名/引数の個数”の形で表記される．

KL1での実行単位はゴールである．与えられたゴールHに対し，述語Hを定義する各節のヘッドH_iとの同一化およびガード部$G_{i1},\ldots,G_{im}$の実行を試み，成功した節のボディ部$B_{j1},\ldots,B_{jn}$を次の新たな実行ゴールとする．この過程を繰り返すことによって，プログラムを実行する．このとき，ボディゴールは同時に実行することができ，これによってプログラムを並列実行するようになっている（並列実行ゴールの指定は後述のプラグマにより行う）．

2.1.2 データ構造

KL1の標準データ型には，記号アトムや整数などのアトミックデータ型と，リストやファンクタなどの構造型データ型がある．

KLICなど多くの処理系では，KL1のデータ構造はセルと呼ばれる最小単位で管理され，アトミックデータ型は1セル中に値を直接格納する．また，構造型データ型は複数のセルからなり，別のデータを指す参照ポインタを格納することによって，複雑なデータ構造を表現できるようになっている．

KL1の変数は論理変数であり，初期状態では特定の値や型を持たない（未具体化）．実行中にアトミックデータや構造型データと同一化（具体化）されると，そのデータを具体値として持つようになる．一度具体化された変数は，他の値に書き換えることはできない．以下，本論ではこの論理変数を単に変数と呼ぶ．

なお，KL1での値の参照は，ある変数が特定の名前を持つファンクタであるかどうかの検査など，構造型データの引数が未具体化でも実行可能なものが多い．そこで本論での具体値とは，基底項に限らず，一部の引数が未具体化変数のままの構造型データも含むものとする．

2.2 KL1処理系KLICの概要

KLICは，(財）新世代コンピュータ技術開発機構（ICOT）で開発されたKL1処理系である．これはKL1プログラムをいったんCプログラムに変換し，ターゲットマシンのオブジェクトコード生成は，その計算機用のCコンパイラに行わせる方式をとっている．このため，物理通信などの機種依存部を除き，移植性の高い処理系になっている．

KLIC分散処理系の計算モデルは，共有メモリを持たない複数のノードが，互いにメッセージ通信をしながら処理を行うというものである．実行開始時の初期ゴールとして，`main/0`が与えられ，並列実行は`@node`プラグマを用いて明示的に記述する．$goal(X_1,\ldots,X_n)$`@node(`N_c`)`という形のボディゴールがノードN_p上に現れると，このゴールはノードN_cに送られ，そこで実行される．このとき，未具体化変数X_iについては，N_p上の変数を指す外部参照ポインタがN_cに送られる（**図1**(a)）．

N_c上でX_iの値が参照されると，N_cからN_pに，この外部参照ポインタが指す値を要求するreadメッセージが送られる．N_pはanswerメッセージにより，要求された値を送信する（図1(b)）．また，N_c上でX_iを具体化した場合は，N_pにこの具体値との同一化を要求するunifyメッセージを送る（図1(c)）．

answerを待っている間や，unifyが来る前にN_pで値を参照しようとした場合，参照を行おうとするゴールは外部参照ポインタやX_iにフックして中断し，他のゴールの実行に切り替わる．メッセージ到着によりフック先のポインタや変数が具体化されると，中断ゴールの実行が再開される．

3. 手法の概要

3.1 KLICの問題点

KLICでは実行コード生成をCコンパイラが行うので，低レベル部分の最適化に関しては比較的効率の良いコードが得られる．しかし，動的なゴール切替えのオーバヘッドなど，KL1の実行モデルによる速度低下は解決できていない．なかでもメッセージ交換型並列

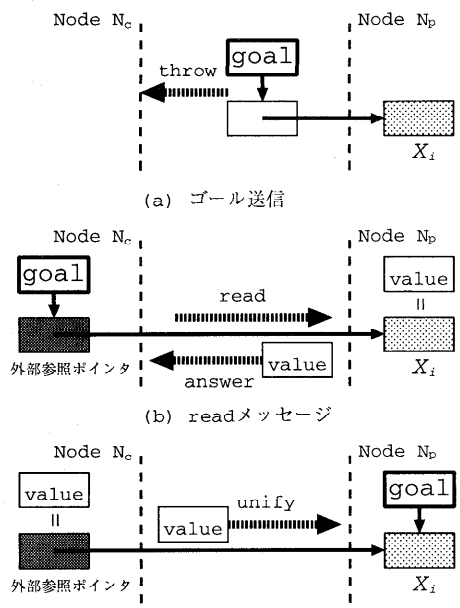


図1 ノード間通信
Fig. 1 Inter-node communications.

```
main :- drive(100,S),stack(S,none)@node(1).
drive(M,S) :- M=:=0 | S=[].
drive(M,S) :- M=\=0 | S=[push(int(M))|S0],
                      S0=[pop(D)|S1],drive1(D,S1).
drive1(D,S1) :- D=int(N) | N1:=N-1,drive(N1,S1).
stack([]          ,D      ) :- terminate(D).
stack([push(X)|S],D      ) :- stack(S,p(X,D)).
stack([pop(X)|S] ,p(Y,D1)) :- X=Y,stack(S,D1).
terminate(D).
```

図2 スタックプログラム
Fig. 2 Stack program.

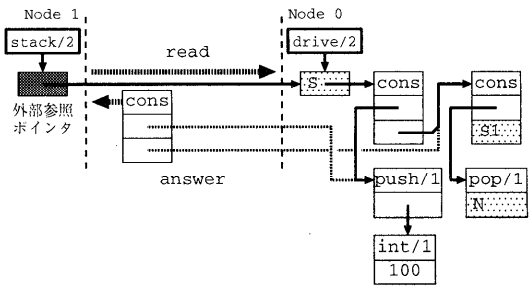


図3 構造型データの送信
Fig. 3 Transfer of structured data.

計算機を対象とした場合，細粒度通信の頻発が大きなオーバヘッドとなっている．

これは，動的なデータ構造生成やゴールスケジューリングのため，データ中の通信対象部分や定義・参照を行うノードが実行時まで判明せず，必要になった時点で送信を要求するという要求駆動的なデータのやりとりを行うためである．この結果，リストやファンクタなどの構造型データを扱う場合，どの部分を参照するかが事前に分からないため，個々の要素が参照されるごとに通信を行ってしまい，通信が細粒度化する．

例として，**図2** に示した，2つのノード間でスタック操作を繰り返すプログラムを考える．ノード1上のゴール **stack/2** はガード部で第1引数を検査するため，ノード0にその値を要求する．しかし，**図3** のように，参照された値がネストした構造型データであっても，ノード1上でどの部分が参照されるか分からないため，ネストの一番外側であるコンス・セルしか送信されない．その結果，このコンス・セルの引数を参照するために，あらためて通信が必要になる．

この問題への対策として，現在のKLICにはバッチ転送モードが実装されている[4]．これは，最初の要素が参照された時点で構造型データ全体を送信することにより，通信回数を減らすというものである．しかしこの方式では，送信した構造型データ内に受信側で参照しない要素があった場合，その部分を送信する時間と受信側で占めるメモリが無駄になる．図3の例では，ノード1で参照しない **int(100)** まで送信してしまう☆．この結果，大規模なデータを論理的に共有するプログラムでは，データ全体を物理的にも共有することになってしまい，メモリ不足によりGCの頻発を生じたり実行不能に陥ったりする恐れがある．

3.2 一括送信手法

3.1節で述べた問題点を改善するため，本研究では，参照側ノードで必要なデータのみをまとめて送る一括送信手法を提案する．

本手法では，対象となるKL1プログラムを並列実行単位ごとに並列プロセスとして分割し，これらの並列プロセス間で一括送信を行う．2.2節で述べたように，KL1ではプラグマ **@node** により並列実行するゴールを指定し，それ以外の部分は同じノード上で実行される．したがって，プラグマで指定されたゴールの各々を開始点とし，そこから同一ノード上で実行される部分を，それぞれ並列プロセスと見なすことができる．

プログラムの初期ゴールである **main/0**，およびすべての **@node** プラグマつきのゴールを並列プロセス初期ゴールとすると，並列プロセス初期ゴール G_I から他の並列プロセス初期ゴールを経由せずに呼ばれるゴールの集合が，G_I の割り当てられたノード上で逐

☆ ネスト構造の特定レベルまでをバッチ転送にすることもできるが，このレベルは実行時にユーザが与えなければならず，全データに対して同じレベルが適用されてしまう．

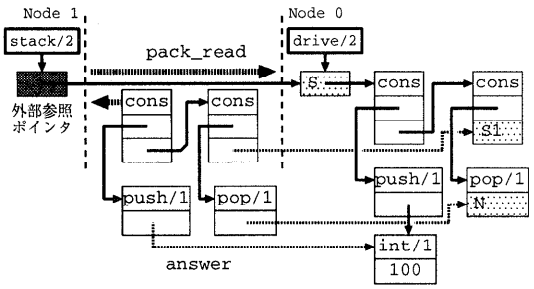


図4 構造型データの一括送信
Fig. 4 Packing transfer of data structures.

次実行される．この集合を，並列プロセス G_I と呼ぶ．以下，本論でのプロセスとは，この並列プロセスを指すものとする．たとえば図2のプログラムは，**main/0** と **stack/2** の2つの並列プロセスに分割できる．

2つの並列プロセス P_p，P_c において，P_p 上の構造型データを P_c が参照する場合を考える．P_p が P_c からのデータ要求メッセージを受信したとき，構造型データ中の具体化済み部分のうち，P_c が必要とする部分のみをすべて送るようにすれば，P_c に不要なデータを送信することなくメッセージ粒度を上げることができ，通信回数が減少する．たとえば図2のプログラムの場合，コンス・セルの連鎖や，その引数であるファンクタ **push/1**，**pop/1** は参照するが，**push/1** の引数は参照しないことが分かっていれば，図3と同じ状況で**図4**のようにして，1往復の通信で具体化済み部分についての必要十分なデータを送ることができる．

本論で提案する一括送信手法では，プログラムの静的解析によりこの参照情報をコンパイル時に調べ，上記のような送信を行うための一括送信コードを生成する．プログラムの実行時には，データ要求に対し，対応する一括送信コードを実行することにより，必要なデータの一括送信を行う．

以下，静的解析および一括送信コードの生成方法について，それぞれ詳しく述べる．なお，例として図2のプログラムを用いる．

4. 静的解析手法

本章では，3章で述べた一括送信を行うためのプログラム解析手法について述べる．

一括送信を行うには，あるプロセスが他のプロセスと論理的に共有する構造型データのうち，どのような要素について，他のプロセスが具体化し，そのプロセス内で参照するかの情報が必要となる．本手法では型解析によりプロセス内で参照・具体化される要素の種類を求め，モード解析により参照・具体化の区別を求める．なお，実際に参照されるかどうかが実行時まで決まらない可能性もあるが，本手法では参照する可能性がある部分すべてを，一括送信の対象とする．

4.1 入力プログラム

対象となるKL1プログラムは，*Moded Flat GHC* [5] と同様に，以下の条件を満たすとする．

(1) 同一変数の出現のうち1個のみが出力出現で，残りはすべて入力出現である．

(2) 組込み述語 **=/2** を除いて，述語の各引数のモードは述語名より一意に決定する．また，ファンクタなど構造データ中の引数のモードは，プログラム中におけるそのデータの出現箇所に依存するが，データ中の他の引数のモードには依存しない．

この条件を満たしたうえで，プログラムは **well-moded** [5]，すなわちすべてのモード制約を満たすモードが存在するとする．また，算術演算式などのマクロは展開され，すべてのゴールが $goal(X_1, \ldots, X_n)$ の形で表されているとする．

解析対象プログラムは，まず3.2節で述べた並列プロセスに分割し，各々のプロセスに対して解析を行う．

4.2 モード解析

本手法では，文献5)で提案されているモード解析を適用し，各プロセス内の変数の入出力方向を解析する．プロセス間のモード解析を行わないことにより，一部の変数のモードが決定できない場合も生じるが，その場合はモードを不明とする．

この手法では完全なモード解析は得られない．しかし，この解析でモード不明な変数はプロセス内で具体化も参照も行われないので，一括送信の対象外となる．したがって，一括送信に必要な情報を得るには，この不完全なモード解析結果で必要十分であり，完全なモード解析に比べ，プロセス間のモード波及を追跡しない分，解析コストを小さくすることができる．

モード解析の結果，プログラム中の変数の出現ごとに入力，出力，不明のいずれであるかが得られる．本論では，これらをそれぞれ *in*，*out*，？で表す．

並列プロセス **stack/2** のモード解析結果を，**図5**に示す．

4.3 型 解 析

2.1.2項で述べたようにKL1の変数は特定の型を持たない．しかし，一度具体化されると値を変更できないから，この具体値の型を，その変数の型と考えることができる．

ただし，同じ変数でも実行パスにより異なるデータ型の値に具体化されることがあるから，静的解析では

```
1 stack([]                ,D(i)          )
 :- terminate(D(i)).
2 stack([push(X(?))|S(i)],D(i)          )
 :- stack(S(i),p(X(?),D(i))).
3 stack([pop(X(?))|S(i)] ,p(Y(?),D1(i)))
 :- =(X(?),Y(?)),stack(S(i),D1(i)).
4 terminate(D(i)).

変数は 変数名 (モード) で表記.
モードは  i: in,  o: out,  ?: ?
```

図5 stack/2のモード解析結果
Fig. 5 stack/2 result of mode analysis.

一般に変数の型は一意に決まらない．そこで，本手法では変数の型を，その変数が実行時にとりうる具体値の型の集合と定義する．

実際にはKLICの実装などの点から，KL1本来のデータ型の分類では不都合を生じる場合がある．また，3.2節で述べた例のような参照パターンを認識するには，名前や引数の個数の異なるファンクタを区別する必要がある．さらに，一括送信ではそれぞれの型について，プロセス内で具体化・参照のいずれの操作が行われるかの情報も必要である．

そこで本手法では，次項に示すようなデータ型の分類を用いて，モードつきの型集合を定義する．そして，プログラム中の各変数についてこれを求めることを，型解析の目的とする．

4.3.1 モードつき型集合の定義

本手法では変数 v_i のモードつき型集合を，$T(v_i) = \{e_1, \ldots, e_n\}$ と定義する．各 e_j は (t_j, m_j)，$v_j(m_j)$，? のいずれかである．

4.3.1.1 (t_j, m_j)

型とそのモードの組であり，型 t_j は**表1**のいずれかの値をとる．

本手法では，KLICの実装上はアトミックと異なる浮動小数点数・文字列や，実際は記号アトムの一種である `[]` を，それぞれアトミックデータ型の一種として扱う．構造型データ型については，前記の理由から，名前や引数の個数が異なるファンクタを，すべて異なる構造型と見なす．コンス・セルやベクタについても，特殊な名前を持つファンクタと見なす．

また，構造型データ型の各引数については，実行パスにより異なる型に具体化される可能性がある．また，個々の引数について，モードが異なる場合もある．このため，それぞれの引数を，再帰的にモードつき型集合 T_i で表す．したがって，以下の型解析手法の記述では，構造型データ型の一般形を $f(T_1, \ldots, T_n)$ のように表す．

モード m_j は，変数 v_i が型 t_j の値をとるときの，解析中のプロセスによる入出力の方向を表す．もしプロセスが該当部分を参照するのであれば，v_i の出現より値がとり出されることになるので，out とし，具体化されるのであれば，v_i の出現に対して値が入力されることになるので in とする．

表1 データ型一覧
Table 1 Data types.

アトミックデータ型		構造型データ型	
記号アトム	a	ファンクタ	$name(T_1, \ldots, T_n)$
空リスト	[]	コンス・セル	$[T_{car} \mid T_{cdr}]$
整数	i	ベクタ	$\{T_1, \ldots, T_n\}$
浮動小数点数	f		
文字列	s		

実際には v_i の出現が複数存在しうるから，v_i に対して具体化と参照がともに起こる可能性がある．しかし，他プロセスからの一括送信対象となるのは，このプロセス内で参照のみ行われるデータ型だけであるから，そのようなデータ型を他と区別できればよい．そこで，m_j を以下のように定義する．

プロセス内で v_i の型 t_j の値に対し，

(1) 参照のみ行われるなら $m_j = out$.
(2) 具体化のみ，もしくは具体化および参照が行われるなら $m_j = in$.
(3) 参照・具体化とも行われないなら $m_j = ?$.

4.3.1.2 $v_j(m_j)$

構造型データ型の引数として変数が含まれる場合，再帰構造などの扱いを簡単にするため，そのモードつき型集合を展開せずに，そのままモードつきの型変数として扱う．

モード m_j については4.3.1.1項と同様に考え，v_j の値に対しプロセスが参照のみ行うなら out，具体化（および参照）を行うなら in，どちらも行わないなら ? とする．

4.3.1.3 ?

モードつき型集合中の未判明要素を表す．本手法ではプロセス単位の解析を行うため，完全な型解析は得られず，最終的に ? が残る可能性もある．しかしモード解析と同様，このような要素についてはプロセス内で参照・具体化が行われないため，問題は生じない．

以上の定義より，たとえばボディゴール `X=f(Y)`, `add(Y,1,Z)` に対し，$T(\mathtt{X}) = \{(\mathtt{f}(\{Y(out)\}), in)\}$ となる．

また，以下の記述ではこのモードつき型集合を，単に型集合と表記する．

4.3.2 型集合の演算定義

型集合に対するAND，OR演算を次のように定義

する．

4.3.2.1 モード m の融合演算

$merge(m_A, m_B)$ は以下の値をとる．

(1) m_A, m_B の少なくとも一方が $in \rightarrow in$

(2) (1) を満たさず，m_A，m_B の少なくとも一方が $out \rightarrow out$

(3) m_A，m_B がともに ？→？

この定義は，型や型変数のモードに関して，4.3.1.1, 4.3.1.2 項で述べた性質を保つためである．

4.3.2.2 型集合の OR 演算

$T_{OR} = OR(T_A, T_B)$ とすると，T_{OR} は以下を満たす最小の集合である．

ただし，$m_{OR} = merge(m_A, m_B)$ である．

(1) $(T_A \ni (t_j, m_A)) \wedge (T_B \ni (t_j, m_B))$
$\rightarrow T_{OR} \ni (t_j, m_{OR})$
ただし構造型データ型については，
$(T_A \ni (f(T_{A,1}, \ldots, T_{A,n}), m_A)) \wedge$
$(T_B \ni (f(T_{B,1}, \ldots, T_{B,n}), m_B))$
$\rightarrow T_{OR,k} = OR(T_{A,k}, T_{B,k})$ とすると
$T_{OR} \ni (f(T_{OR,1}, \ldots, T_{OR,n}), m_{OR})$

(2) $(T_A \ni v_i(m_A)) \wedge (T_B \ni v_i(m_B))$
$\rightarrow T_{OR} \ni v_i(m_{OR})$

(3) $(T_A \ni ?) \vee (T_B \ni ?) \rightarrow T_{OR} \ni ?$

(4) (1) に該当しない $(t_j, m_A) \in T_A$，(2) に該当しない $v_i(m_A) \in T_A$ については，$T_{OR} \ni (t_j, m_A)$，$T_{OR} \ni v_i(m_A)$．T_B についても同様．

この演算は，意味上は 2 つの型集合の和集合を求めるものであるが，簡潔な型集合を得るために，同じ型や型変数についてはモードや引数の型集合を融合している．

4.3.2.3 型集合の AND 演算

$T_{AND} = AND(T_A, T_B)$ とすると，T_{AND} は以下のように定義される．

ただし，$m_{AND} = merge(m_A, m_B)$ である．

(1) $T_A \ni ?$, $T_B \ni ?$
$\rightarrow T_{AND} = OR(T_A, T_B)$

(2) $T_A \not\ni ?$, $T_B \not\ni ?$
$\rightarrow T_{AND}$ は以下を満たす最小の集合

(a) $(T_A \ni (t_j, m_A)) \wedge (T_B \ni (t_j, m_B))$
$\rightarrow T_{AND} \ni (t_j, m_{AND})$
ただし構造型データ型については，
$(T_A \ni (f(T_{A,1}, \ldots, T_{A,n}), m_A)) \wedge$
$(T_B \ni (f(T_{B,1}, \ldots, T_{B,n}), m_B))$
$\rightarrow T_{AND,k} = AND(T_{A,k}, T_{B,k})$
とすると
$T_{AND} \ni (f(T_{AND,1}, \ldots, T_{AND,n}), m_{AND})$

(b) $(T_A \ni v_i(out)) \vee (T_B \ni v_i(out))$
$\rightarrow T_{AND} \ni v_i(out)$

(c) (2)(b) に該当しない $v_i(m_i)$ について，
$(T_A \ni v_i(?)) \vee (T_B \ni v_i(?))$
$\rightarrow T_{AND} \ni v_i(?)$

(3) $T_A \ni ?$, $T_B \not\ni ?$
$\rightarrow T_A$ から ？を除いたものを T'_A として，
$T_{AND} = OR(AND(T'_A, T_B), T_B)$
$T_A \not\ni ?$, $T_B \ni ?$ の場合も同様

この演算は，意味上は 2 つの型集合の共通要素を持つ集合を求めるものである．不明要素？はもう一方の任意要素に対応しうるので，T_A，T_B がともに？を含む場合は，OR 演算と同じ結果になる．また，T_A のみ？を含む場合は，T_A の要素のうち T_B のどの要素とも型や型変数が異なるものは除かれるが，T_B の要素は T_A の？に対応しうるため，すべて残される．

さらに，プログラム中の同一化による同値関係を表す集合 $U_1, \ldots, U_M$ を定義する．U_k の各要素は変数 v_i，もしくは集合 $\hat{U}_l = \{u_{l_1}, \ldots, u_{l_m}\}$ であり，u_{l_j} は変数，またはヘッドやボディゴールの引数に直接現れる具体値である．$\hat{U}_l$ は，実行時に選択される節により，その要素のいずれかが U_k 内のすべての v_i と同一化することを表す．以下，U_k を同一化集合，$\hat{U}_l$ を同一化候補集合と呼ぶ．

4.3.3 型集合の解析アルゴリズム

以下，モードつき型集合の解析アルゴリズムを述べる．この解析により，プロセス内の各変数について，とりうる値の型の集合，および個々の型について，プロセス内で参照のみが行われるか否かが得られる．

本アルゴリズムでは，まず組込み述語により局所的に型集合を決定した後，組込み述語 =/2 やボディ・ヘッド間の同一化による変数間の型波及を，同一化集合を用いて解析する．

以下，各集合の値は解析の過程での一時的なものとし，$\leftarrow$ は右辺を左辺の新たな値とする操作を示す．

また，構造型データの変数引数間で型を波及させるため，以下のアルゴリズム中の演算 $AND(T_A, T_B)$ で，$T_A \ni v_i(m_i)$ ならば，$T(v_i) \leftarrow OR(T(v_i), T_B)$ とする（逆の場合も同様）．ここで OR 演算を用いるのは，この型波及が，実行されるパスに依存した非決定的なものであるためである．

4.3.3.1 組込み述語による型決定

型検査・整数演算などの組込み述語や，変数と具体値との同一化述語は，各引数にとりうる型やモードを決定できる．これを利用し，引数に現れる変数につい

て型集合の初期値を決定する．

ここで判明しなかった $T(v_j)$ については，初期値 $T(v_j)=\{?\}$ とする．

4.3.3.2 同一化集合の作成

変数と変数の同一化 $v_i=v_j$ に対し，$(U_k \ni v_i) \wedge (U_l \ni v_j)$ を満たす U_k, U_l があれば $U_k \leftarrow U_k \cup U_l$ とし，U_l は削除する．U_k のみ存在する場合は，$U_k \leftarrow U_k \cup \{v_j\}$ とする（U_l のみ存在する場合も同様）．どちらも存在しなければ，新たな同一化集合 $U_m=\{v_i, v_j\}$ を生成する．

この操作により，各節内の変数間の同一関係が，同一化集合の形で表される．

次に，ヘッド・ボディの同一化による引数の同一化を解析する．述語 p/n について，プロセス内のクローズヘッド p/n およびボディゴール p/n の集合をそれぞれ $H_{p/n}$, $B_{p/n}$，第 i 引数の集合をそれぞれ $H_{p/n,i}$, $B_{p/n,i}$ と表す．ただし，引数に現れる具体値 b_m については，その型 t_m を用いて代わりに $(t_m, out) \in H_{p/n,i}$，あるいは $(t_m, in) \in B_{p/n,i}$ とする．ここでのモードは，ヘッドの引数は値の検査（参照），ボディゴールの引数は対応するヘッド引数に対する具体化であることを意味している．各変数 $v_h \in H_{p/n,i}$ について，同一化集合 $U_k \ni v_h$ があれば $U_k \leftarrow U_k \cup \{B_{p/n,i}\}$ とし，なければ，新たな同一化集合 $U_m=\{v_h, B_{p/n,i}\}$ を生成する．各変数 $v_b \in B_{p/n,i}$ についても同様にする．

この操作により，ヘッド・ボディ間で同一化しうる変数の集まりが，同一化候補集合の形で同一化集合に加えられる．

4.3.3.3 同一化集合の型集合作成

同一化集合 U_k について，各 $v_i \in U_k$ は必ず同じ値を持つから，すべての $T(v_i)$ は最終的に同じ型集合を持つ．これを T_k とすると，T_k は次の操作で求められる．

(1) 初期値 $T_k=\{?\}$ とする

(2) 各 $v_i \in U_k$ について，$T_k \leftarrow AND(T_k, T(v_i))$

(3) 各同一化候補集合 $\hat{U}_l \in U_k$ について，初期値 $\hat{T}_l=\emptyset$ とする．各変数 $v_m \in \hat{U}_l$ や型とモードの組 $(t_n, m_n) \in \hat{U}_l$ について，$\hat{T}_l \leftarrow OR(\hat{T}_l, T(v_m))$, $\hat{T}_l \leftarrow OR(\hat{T}_l, \{(t_n, m_n)\})$ として型集合 $\hat{T}_l$ を求め，$T_k \leftarrow AND(T_k, \hat{T}_l)$ とする．

4.3.3.4 型集合の適用と終了判定

各 $v_i \in U_k$ に対し，$T(v_i) \leftarrow AND(T(v_i), T_k)$ とする．この結果 $T(v_i)$ が更新された各 v_i について，$v_i \in U_k$ または $v_i \in \hat{U}_l \in U_k$ となる同一化集合 U_k に対して，再び 4.3.3.3 項からの処理を行う．

以上の操作を繰り返し，更新がなくなれば解析は終了する．

```
T(1,D) = T(2,D) = T(3,D1) = T(4,D)
       = {( p({X(2,?),Y(3,?)}, {D(2,i),D1(3,i)}),i)}
T(2,X) = T(3,X) = T(3,Y)
       = {?}
T(2,S) = T(3,S)
       = {([],o),
          ([{(push({X(2,?)}),o),(pop(X(3,?)),o)} |
            {S(2,o),S(3,o)} ] ,o)}
```

$T(i,v)$: 第 i 節の変数 v の型，
$v(i,m)$: 第 i 節の変数 v（モード m）（節番号 i は図 5 参照）
モードは i: *in*, o: *out*, ?: ?

図 6 stack/2 の型解析結果
Fig. 6 stack/2 result of type analysis.

並列プロセス **stack/2** の型解析結果を，**図 6** に示す．前記のアルゴリズムにより，プロセス中の変数が取り得る型と入出力方向が得られ，たとえばファンクタ **push/1** は **stack/2** により参照のみ行われるが，その引数は参照も具体化もされないことや，ファンクタ **p/2** は **stack/2** により具体化が行われることが示されている．

5. 一括送信コードの生成

本章では，4 章の静的解析結果を利用して，3 章で述べた一括送信コードを生成する方法を述べる．以後の説明では，ノード N_p 上のプロセス P_p がノード N_c 上にプロセス P_c を生成する場合を考える．また，P_c の初期ゴールを $goal(v_1, \ldots, v_n)$ とする．

5.1 解析結果の意味

P_c に対する 4.3 節の型解析の結果，各 v_i について型集合 $T(v_i)$ が得られる．各 (t_j, m_j), $v_k(m_k) \in T(v_i)$ より，一括送信対象は次のように判定できる．

(1) $m_j=out$ なら，型 t_j のデータは P_c で参照のみ行われるので，P_p からの一括送信対象となる．$m_j=in$ なら P_c が具体化し，$m_j=?$ なら具体化も参照もしないので，送信の必要はない．

t_j が構造型データ型のときは，各引数についても同様に再帰的な判定を行う．

(2) $m_k=out$ なら，変数 v_k の具体値は P_c で参照のみ行われるので，$T(v_k)$ の各要素について同様に一括送信対象の判定を行う．$m_k=in$, ? のときは，(t_j, m_j) と同様に送信対象とならないので，$T(v_k)$ の判定は必要ない．

そこで，実行時に実際に生成された v_i の具体値に対

し，上記の判定を行って該当部分を一括送信する KL1 コードを生成する．コードを KL1 で記述することにより一括送信処理を KL1 ゴールの 1 つとして扱えるため，送信データの型判定やメモリ管理に KL1 ランタイムの機能を利用でき，実装が容易になる．

実行時には，P_c から read メッセージが送られると，P_p はこのコードより一括送信ゴールを生成し，その実行により一括送信を行う．

また，P_c が具体化し P_p が参照するデータについては，上記と逆に考えて P_p に対する型解析の結果より一括送信コードを生成し，P_c は unify メッセージ送信時に一括送信ゴールを実行すればよい．

5.2 一括送信コードの生成手順

一括送信ゴールは送信対象 v_i に対し，具体値の各要素について 5.1 節の判定を行い，送信対象となる場合は送信バッファに格納すればよい．

そこで，送信対象となる型や型変数について，それぞれ次の処理を行う送信データ格納節を生成する．

アトミックデータ型 b_j　具体値が b_j ならバッファに格納する．

構造型データ型 $f(T_1,\ldots,T_n)$　具体値が f/n なら，構造型データ型名 f/n を格納した後，各 T_i に対応する送信データ格納節を呼び出す．

型変数 $v_j(out)$　$T(v_j)$ に対応する送信データ格納節を呼び出す．

これ以外の場合は受信側で参照されないので，具体値は送信対象とせず，外部参照ポインタを格納する．

T_i や $T(v_j)$ に対応する送信データ格納節がまだ生成されていない場合は，同様にして生成を行う．

たとえばプロセス `stack/2` 第 1 引数の場合，4 章での解析結果（図 6）より，一括送信コードは**図 7** のようになる．コード中の '`inline:`' は，ガード部で変数の未具体化判定を行う C コードをインライン展開している．また，一括送信バッファは KLIC のデータ型拡張用の機能である generic object として実装されており，コード中の '`generic:`' は，この object のメソッド呼び出しを用いてバッファ操作を行っている．

このコードは，`push/1`, `pop/1` は格納するがそれらの引数は格納しない，という 3.2 節で述べた動作を実現している．また，解析時にはコンス・セルの再帰構造による線形リスト構造を認識していないが，コード生成時に格納節の再帰呼び出しが生成されるため，具体化されている分はすべて一度に送られるようになっている．

```
packsend_end(B)
 :- generic:sendbuf(B).
packsend_stack_2_1_top(V,BIn)
 :- packsend_stack_2_1(BIn,V,BO),
    packsend_end(BO).
packsend_stack_2_1(BIn,V,BOut)
 :- inline:"guard_unbound(%0,%f)":[V+any]
  | generic:put_in_ref(BIn,V,BOut).
otherwise.
packsend_stack_2_1(BIn,V,BOut)
 :- V = []
  | generic:put_atomic(BIn,V,BOut).
packsend_stack_2_1(BIn,V,BOut)
 :- V = [V0|V1]
  | packsend_stack_2_1_cons(BIn,V0,BO),
    packsend_stack_2_1(BO,V1,B1),
    generic:put_cons(B1,V,BOut).
packsend_stack_2_1_cons_top(V,BIn)
 :- packsend_stack_2_1_cons(BIn,V,BO),
    packsend_end(BO).
packsend_stack_2_1_cons(BIn,V,BOut)
 :- inline:"guard_unbound(%0,%f)":[V+any]
  | generic:put_in_ref(BIn,V,BOut).
otherwise.
packsend_stack_2_1_cons(BIn,V,BOut)
 :- V=push(V0)
  | generic:put_unused(BIn,V0,BO),
    generic:put_functor(BO,V,BOut).
packsend_stack_2_1_cons(BIn,V,BOut)
 :- V=pop(V0)
  | generic:put_unused(BIn,V0,BO),
    generic:put_functor(BO,V,BOut).
```

※ 送信バッファ実装の都合上，再帰構造の格納はボトムアップで行うようになっている

図 7 stack/2 第 1 引数の一括送信コード
Fig. 7 KL1 code for packing data transfer (first argument of stack/2).

6. 性能評価

6.1 評価対象

評価用の処理系として，次の 2 種類を用いた．

(1) AP1000 版 KLIC+64 台構成の AP1000

(2) PVM 版 KLIC+イーサネットで結合した 2 台のワークステーション（SS10）

(1) の処理系は，ICOT より公開されている (2) の処理系を基に，本研究室で移植を行ったものである．

また，評価プログラムとして，次の 5 種類（ベンチマーク 3 本，応用 2 本）を用いた．

`stack` 図 2 に示した，2 ノード間でスタック操作を行うプログラムである．繰返し回数は (1), (2) でそれぞれ 10000, 1000 とした．

`mastermind` 数当てゲームで正解に至るまでの推測の並びを全解探索するものであり，分岐が生じるたびに細粒度の負荷分散を行う．

`nqueen` 12-queen 問題の解を全解探索するものであり，負荷分散は最初の分岐のみで行われる．

`pia` 反復改善法を用いたマルチプル・シーケンス・ア

表2 AP1000での実行結果
Table 2 Result on AP1000.

プログラム		逐次	通常	バッチ	一括
stack	時間	1.56	51.6	30.1	21.1
(2ノード)	通信	—	136	70.0	30.0
	転送	—	3741	2266	1524
mastermind	時間	3.44	22.9	139	7.59
(32ノード)	通信	—	216	9.90	43.5
	転送	—	6697	48580	2885
nqueen	時間	412	52.4	41.2	36.2
(12ノード)	通信	—	43.4	0.27	0.21
	転送	—	1265	2096	276
pia	時間	248	23.7	25.7	28.2
(18ノード)	通信	—	65.7	55.1	44.5
	転送	—	2087	1891	1807
cmgtp	時間	43.4	42.4	15.2	14.2
(8ノード)	通信	—	28.9	0.22	0.61
	転送	—	953	411	376

表3 SS10+イーサネットでの実行結果
Table 3 Result on SS10+ethernet.

プログラム		逐次	通常	バッチ	一括
stack	時間	0.06	96.3	31.4	22.7
(2ノード)	通信	—	13.0	4.44	3.01
	転送	—	367	167	152
nqueen	時間	68.8	194	36.5	37.3
(2ノード)	通信	—	22.3	0.052	0.79
	転送	—	653	1084	152
pia	時間	109	404	174	163
(2ノード)	通信	—	59.1	20.8	18.7
	転送	—	1889	1209	1016

ライメント・プログラムであり，1回の改善ごとに，アライメント処理を並列実行する．問題のサイズは長さ30の配列17本とした．

cmgtp モデル生成型の定理証明システムで準群問題（QG5のオーダ8）を解くプログラムである．負荷分散はマスタースレーブ方式を用いており，暇なノードに仕事が割り当てられる．

6.2 実行結果

各プログラムについて，通常の送信，3.1節で述べたバッチ転送モード，一括送信，の3通りについて，それぞれ実行時間（単位：秒）と処理系全体の物理通信回数（単位：千回）および総転送量（単位：Kbyte）を測定した．結果を**表2**，**表3**に示す．ただし，通信オーバヘッドが非常に大きい**mastermind**と，マスタースレーブ方式のため並列実行に3台以上が必要な**cmgtp**については，AP1000上の結果のみをあげてある．

本手法を用いたことにより，**pia**以外では通信回数が数分の1から数十分の1に減少している．この結果，各メッセージごとに必要なヘッダ情報などが削減されるため，通信量も通常の送信より大幅に減少している．

stackや**mastermind**のように通信回数が多いプログラムの場合は，2〜4倍の速度向上が得られた．これらはベンチマーク用の極端なプログラムであるため，本手法を用いても逐次に対する速度向上は得られない．しかし，ある程度並列性の期待できる大規模プログラムでも局所的に通信ネックを生じる可能性があるから，本手法はそのような場合の改善に役立つと考えられる．

一方，相対的に通信回数の少ない他の3本も，多くの場合は速度向上が得られた．とくに，AP1000上の**cmgtp**，SS10上の**nqueen**では，それぞれ3倍，5倍の速度向上が得られ，通常の通信では得られなかった台数効果を引き出すことができた．**pia**についてはAP1000上でかえって速度が低下したが，これは一括送信では処理データが一度に届く代わりにそれまで若干の待ち時間ができるのに対し，通常の送信では送信完了までの時間は長くても届いた分から処理を開始できるため，と考えられる．実際に，このデータを一括送信対象からはずしたところ，実行時間は21.4秒まで改善できた．一方，通信速度の遅いSS10上では，この待ち時間よりも通信回数の減少の効果が大きいため，2倍以上の速度向上が得られている．

バッチ転送との比較でも，多くの場合は一括送信の方が良い性能を出している．**mastermind**や**cmgtp**のようにバッチ転送の方が通信回数を削減できているものもあるが，参照されないデータまで送信してしまうため，通信量は一括送信より多くなっている．**mastermind**などでは通常の通信よりも大幅に通信量が増えた結果，大きな速度低下を生じており，このような事態を避けるためには，本手法は有効であるといえる．

7. おわりに

本論では，並列論理型言語KL1のメッセージ通信最適化手法と，その性能評価について述べた．

本手法ではプログラムの静的解析を行い，通信対象となるデータ型や通信方向をコンパイル時に決定する．そしてこの情報を利用して，受信側で参照されるデータのみを一括送信するような通信コードを生成することにより，余分なデータ転送を生じずに粒度の高い通信を実現する．

ベンチマークや応用プログラムに本手法を適用した結果，大幅な通信回数の削減を達成できた．また，多くの場合において数倍の速度向上が得られた．とくに，イーサネットのように通信オーバヘッドが大きい環境

や，大規模なデータ転送が行われるプログラムでは，本手法により従来の通信方法では得られなかった並列効果を引き出すことができ，このような場合に対する最適化手法としての有効性が確認できた．

しかしながら今回の性能評価でも，pia のように一括送信によるデータ到着までの待ち時間が若干の性能低下につながる場合があった．これを解決するには，一括送信するデータサイズに上限を設け，大規模データの場合は何度かに分割して送るなどの工夫が必要である．また，細粒度のまま通信オーバヘッドを削減する単方向送信[7]の適用も有効と考えられる．

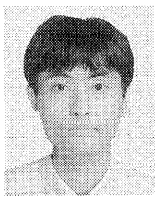
また，一括送信ではデータ要求を受けた段階で具体化されている部分だけを対象としているため，参照が具体化より先に届くことが多いプログラムでは効果が期待できない．この問題は，具体化がある程度進むまでデータ送信を遅らせる遅延送信により解決できると考えられる．現在，静的解析によりデッドロックを生じない範囲で遅延送信を行う手法の研究を進めており，前記の単方向送信なども併用した一括送信の改良を予定している．

謝辞 並列計算機 AP1000 の実行環境をご提供いただきました（株）富士通研究所，および数々の情報と助言をいただいた（財）新世代コンピュータ技術開発機構と KLIC タスク・グループの方々に感謝いたします．また，日頃ご討論いただく富田研究室の諸氏に感謝いたします．

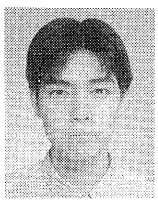
参考文献

1) Ueda, K.: Guarded Horn Clauses: A Parallel Logic Programming Language with the Concept of a Guard, Technical Report, ICOT (1986).
2) Ueda, K. and Chikayama, T.: Design of the Kernel Language for the Parallel Inference Machine, *Comput. J.*, Vol.33, No.6, pp.494–500 (1990).
3) Chikayama, T., Fujise, T. and Sekita, D.: A Portable and Efficient Implementation of KL1, *Proc. 6th Intl. Symp. PLILP'94 LNCS 844*, pp.25–39, Springer-Verlag (1994).
4) 近山 隆：KLIC ユーザーズマニュアル，ICOT (1995).
5) Ueda, K. and Morita, M.: Moded Flat GHC and Its Message-oriented Implementation Technique, *New Generation Computing*, Vol.13, No.1, pp.3–43 (1994).
6) Bansal, A.K. and Sterling, L.: An Abstract Interpretation Scheme for Logic Programs Based on Type Expression, *Proc. Intl. Conf on FGCS'88*, pp.422–429 (1988).
7) Nakashima, H. and Inamura, Y.: An Efficient Message Transfer Mechanism Bypassing Transit Processors, 並列処理シンポジウム JSPP'92, pp.123–130 (1992).

(平成 7 年 9 月 1 日受付)
(平成 9 年 6 月 3 日採録)



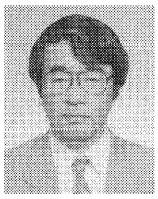
大野 和彦（学生会員）
1970 年生．1993 年京都大学工学部情報工学科卒業．1995 年同大学院修士課程修了．同年同大学院博士後期課程進学．現在同博士後期課程在学中．並列論理型言語の実装方式に関する研究に従事．



伊川 雅彦（正会員）
1973 年生．1995 年京都大学工学部情報工学科卒業．1997 年同大学院修士課程修了．同年三菱電機（株）入社．

森 眞一郎（正会員）
1963 年生．1987 年熊本大学工学部電子工学科卒業．1989 年九州大学大学院総合理工学研究科情報システム学専攻修士課程修了．1992 年九州大学大学院総合理工学研究科情報システム学専攻博士課程単位取得退学．同年京都大学工学部助手．1995 年同助教授．工学博士．並列分散処理．計算機アーキテクチャの研究に従事．IEEE-CS, ACM 各会員．

中島 浩（正会員）
1956 年生．1971 年京都大学大学院工学研究科情報工学専攻修士課程修了．同年三菱電機（株）入社．推論マシンの研究開発に従事．1992 年京都大学工学部助教授．1997 年豊橋技術科学大学情報工学系教授．並列計算機のアーキテクチャ，プログラミング言語の実装方式に関する研究に従事．工学博士．1988 年元岡賞，1993 年坂井記念特別賞受賞．IEEE-CS, ACM, ALP 各会員．



富田　眞治（正会員）

1945年生．1973年京都大学大学院博士課程修了，工学博士．同年同大学工学部情報工学教室助手．1978年同助教授．1986年九州大学大学院総合理工学研究科教授．1991年京都大学工学部情報工学科教授．計算機アーキテクチャ，並列計算機システムに興味を持つ．著書「並列コンピュータ工学」「並列処理マシン」「コンピュータアーキテクチャI」など．電子情報通信学会，IEEE，ACM各会員．本会理事．