静的解析による並列論理型言語 KL1 の通信最適化

大野 和彦　伊川 雅彦　森 眞一郎　中島 浩　富田 眞治
京都大学 工学部

概 要

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、要求駆動的なデータ転送により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、必要なデータのみを一括送信する最適化手法を提案する。本手法により不要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・実行時間ともに大幅に改善されることが確認できた。

Abstract

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 receiver process. With this information, packed messages are generated. As a result of evaluation, the execution time and the number of communications were considerably reduced.

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、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いる。また、移植性の高い KL1 処理系 KLIC [3] が開発され、様々な並列計算機やワークステーション上での利用を目指している。このため、将来的にはこれらのソフトウェアの利用や、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に決定し、必要になると要求・返信という形で通信が行われるためである。

そこで本研究では、静的解析によりデータフローやデータ型を抽出し、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手法を提案する。本手法ではプロセス単位の解析・最適化を行うため、処理コストは比較的小さくできる。解析段階ではモード・タイプ解析 [4],[5] により、生じるデータの型や依存関係を抽出する。続いてこの情報を利用し、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通信最適化の具体的手法を説明し、6 章で性能評価の結果について述べ、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

Efficient Message Communication of Concurrent Logic Programming Language KL1 Based on Static Analysis
Kazuhiko OHNO, Masahiko IKAWA,
Shin-ichiro MORI, Hiroshi NAKASHIMA,
Shinji TOMITA
Faculty of Engineering, Kyoto University

表される。

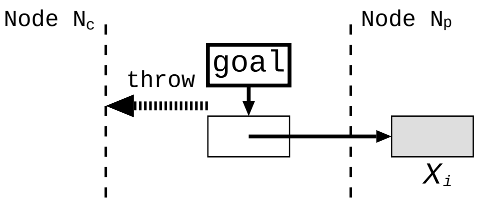
$$H \quad :- \quad G_1,\ldots,G_m | B_1,\ldots,B_n$$

H, G, Bは述語であり、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。述語は、ヘッドの述語名、および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。

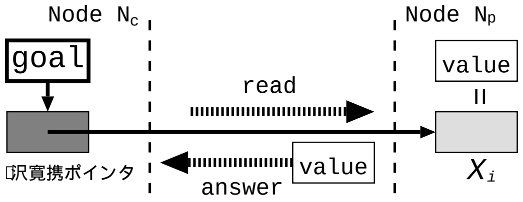
実行の単位はゴールと呼ばれる。与えられたゴール Hを述語と同一化し、その定義部を次の新たな実行ゴールとする。この過程を繰り返すことによって、プログラムの実行が行われる。

このとき、述語 Hを定義する各節のヘッド H_iとの同一化、ならびにガード部 $G_{i1},\ldots,G_{im}$の実行を並列に行うことができる。ただし、Hの引数を具体化することはできない。続いて、ガード部が成功した節のうち一つが選択され、そのボディ部 $B_{j1},\ldots,B_{jn}$が並列実行される。

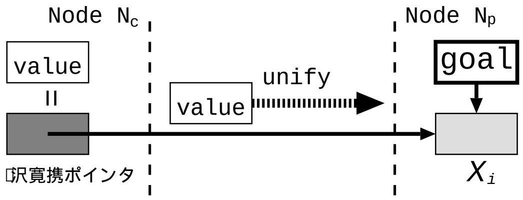
なお、KL1 では、ガードゴールに書けるのは組込述語のみという制限が付けられている。ボディゴールには、組込述語とユーザ定義述語を記述できる。また、実行開始時の初期ゴールとして、`main/0` が与えられる。



(a) ゴール索軒



(b) readメッセージ



(c) unifyメッセージ

図 1: ノード間通信

2.1.2 データ構造

KL1 のデータ構造は、アトムや整数などのアトミックデータ型、リストやファンクタなどの構造データ型、および変数の 3 種類に分類できる。このうち変数は、初期状態では未具体化で値や特定の型を持たず、アトミックデータ型や構造データ型との同一化 (具体化) により具体値を持つようになる。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。

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る。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。

現在の KLIC にはベクタやストリングなども実装されているが、本論では省略する。

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取っている。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。並列実行は`@node` プラグマを用いて明示的に記述し、$goal(X_1,\ldots,X_n)$`@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、未具体化の変数 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 1a)。

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 1b)。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 1c)。

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、それらのゴールは外部参照ポインタや X_iにフックし、他の

ゴールの実行に切り替わる。メッセージ到着によりこれらが具体化されると、このゴールの実行が再開される。

3　本手法の概要

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一般的には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データを扱う場合、どの部分を参照するかが判らないため個々の要素が参照される毎に通信を行ってしまい、通信が細粒度化する。

改善方法として最初の要素が参照された時点で構造データ全体を送信してしまうという実装も考えられる。しかしこの場合、通信回数は減らすことができるが、送信した構造データ内に受信側で参照しない要素があった場合、その部分の通信時間および受信側で占めるメモリが無駄になる。

3.2　通信最適化の方針

3.1 節で述べた問題点を改善するため、本研究では次のような一括送信手法を提案する。

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。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、構造データ中のすでに具体化された部分のうち P_cが必要とする部分のみをすべて送るようにすれば、余分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。

KL1 では変数が型を持たないため、文法上は引数のデータ型は判明しない。しかし、タイプ解析を行うことにより、実行時にある変数がとり得るデータ型をコンパイル時に決定できる。

```
main :- drive(10000,S),stack(S,none)@node(1).
drive(M,S) :- M=:=0 | S=[].
drive(M,S) :- M=\=0 |
  S=[push(M)|S0],S0=[pop(N)|S1],
  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

また、このタイプ解析では、プログラム中における変数の具体化・参照操作を元に解析を行う。したがって、タイプ解析をプログラム中の並列実行単位である並列プロセス毎に行った場合、ある変数 v_iの具体値であるデータ構造のうち並列プロセス P_jのタイプ解析でデータ型の判明した部分は、P_j内で具体化あるいは参照が行われることを表している。

また、モード解析を行うことにより、変数の各出現における具体化・参照の別が判明する。したがって判明したデータ型中の各要素について、プロセス内の操作が具体化・参照のいずれであるかも決定できる。

これらの情報を利用し、前記のようなデータ送信を行う KL1 コードを生成する。以下、これを一括送信コードと呼ぶ。

一括送信コードを KL1 で記述することにより、一括送信処理を KL1 ゴールの一つとして扱える。このため送信する KL1 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。

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。なお、例として図 2に示したスタックプログラムを用いる。

4.1　入力プログラムの条件

対象となる KL1 プログラムは、*Moded Flat GHC* [4] 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上で、プログラムは **well-moded** [4]、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。

以後の記述では述語 $p(v_1,\ldots,v_n)$ 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 $H_{p/n,i}$, $B_{p/n,i}$ と表記する。また、$f(v_1,\ldots,v_m)$ で変数 $v_1,\ldots,v_m$ を含む構造データを表すことにする。

4.2 並列プロセス分割

KL1 ではプラグマにより並列実行するゴールを指定し、それ以外の部分は単一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、それぞれプロセスとみなすことができる。

プログラムの初期ゴールである `main/0`、およびすべての`@node` プラグマ付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次実行される部分である。この集合を、並列プロセス G_I と呼ぶ。

例えば図 2のプログラムは、`main/0` と `stack/2` の二つの並列プロセスに分割できる。

4.3 モード解析

本手法では、[4] で提案されているモード解析を並列プロセス毎に適用する。このため解析コストを小さくできる。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、このような変数はそのプロセス内で具体化・参照は行われないため、本手法ではモード情報は不要である。

以下、本論では変数 v_i のモードを $m(v_j)$ で表し、入力出現、出力出現、入出力不明な出現をそれぞれモード in, out, $unknown$ で表す。また、逆モードを $\overline{m(v_j)}$ で表し、$\overline{in} = out$, $\overline{out} = in$, $\overline{unknown} = unknown$ とする。

```
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, `?`: $unknown$

図 3: `stack/2` のモード解析結果

モード解析の結果、プログラム中の変数の出現毎に入出力のいずれであるかが得られる。並列プロセス `stack/2` のモード解析結果を、図 3 に示す。

4.4 タイプ解析

2.1.2 節で述べたように、KL1 の変数は型を持たない。したがってタイプとは、ある変数が実行時にどのようなデータ型に具体化され得るかを示したものである。ただし、実行パスにより同じ変数が異なるデータ型に具体化されるようなプログラムも考えられるから、静的解析では具体値の型が一意に決まるとは限らない。そこで、タイプは可能性のある具体値の型の集合で表す。

変数は節毎にローカルであり、一度具体化された変数は値を変更できないから、同じ節内で同一変数が違う値をとることはない。したがって、節内の出現数にかかわりなく、変数毎に 1 個のタイプを考えればよい。

変数 v_i のタイプを $T(v_i) = \{t_1,\ldots,t_m\}$ で表すと、各 t_k は以下のいずれかの値をとる。

1. $atom$, $integer$, `[]`
2. `[`$t_{car}|t_{cdr}$`]`, ファンクタ名 $(t_1,\ldots,t_n)$
   構造データのタイプは、car, cdr や各引数を各々のタイプに置き換えた形で表現する。
3. $v_j(m_j)$
   構造データの要素として現れる変数をモード付きで表す。$v_j(m_j)$ が実際にとるタイプは、$T(v_j)$ の値である。
4. $unknown$
   最終的に判明しなかったタイプを表す。

さらに、同一化による同値関係を、集合 $U_1,\ldots,U_M$ で表すことにする。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、または述語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内の他の変数と同一化することを表している。以後、U_kを同一化集合、$\hat{U}_l$を同一化候補集合と呼ぶ。

また、具体値を b_j で表し、$T(b_j)$ は b_j の型を表すものとする。例えば、$T(10) = \{integer\}$, $T(\texttt{move(left)}) = \{\texttt{move}(atom)\}$ である。

以下、具体的なタイプ解析手順を述べる。

4.4.1　節内解析

節内の各ゴールに対し、次の操作を適用し、節毎に局所的な解析を行う。

1. 組み込み述語 $goal(\ldots, v_i, \ldots)$
   $\Rightarrow T(v_i) = \{$ その位置に来るべき型 $\}$
2. $v_i = b_m$
   $\Rightarrow T(v_i) = T(b_m)$
3. $v_i = v_j$
   $\Rightarrow$ $v_i \in U_k,\ v_j \in U_l$となる同一化集合 U_k, U_lがすでに両者とも存在すれば、$U_k \leftarrow U_k \cup U_l$ とし、U_lは削除する。
   U_kのみ存在する場合は、$U_k \leftarrow U_k \cup \{v_i, v_j\}$ 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 $M \leftarrow M+1$ とし、新たに $U_M = \{v_i, v_j\}$ を生成する。

上記の操作により判明しなかった $T(v_j)$ については、初期値として $T(v_j) = \emptyset$ とする。

4.4.2　プロセス内解析

プロセス全体について、ボディゴール・クローズヘッド間の同一化によるタイプ波及を解析する。

アルゴリズムは、以下のようになる。

1. $\forall v_h \in H_{p/n,m},\ \forall v_b \in B_{p/n,m}$に対して、
   $\hat{U}_h = H_{p/n,m},\ \hat{U}_b = B_{p/n,m}$とし、
   $\forall U_h \ni v_h,\ \forall U_b \ni v_b$について、
   $U_h \leftarrow U_h \cup \{\hat{U}_b\},\ U_b \leftarrow U_b \cup \{\hat{U}_h\}$
   として、同一化候補集合$\hat{U}$を各 Uに加える。
   ただし v_hに対し該当する U_hがない場合は、$M \leftarrow M+1,\ U_M = \{v_h\} \cup \{\hat{U}_b\}$ とする。
   v_bについても同様である。
2. U_kに含まれる各変数のタイプの集合 T_kを求める。以下これを同一化タイプ集合と呼ぶ。T_kは次のように定義できる。
   $$T_k = \textstyle\bigcup_{v_i \in U_k} T'(v_i),$$
   $$T'(v_i) = \begin{cases} \{v_i\} & \text{if } T(v_i) = \emptyset \\ T(v_i) & \text{otherwise} \end{cases}$$
   また、同一化候補集合$\hat{U}_l \in U_k$についても、同様に同一化候補タイプ集合
   $$\hat{T}_l = \textstyle\bigcup_{\forall u_m \in \hat{U}_l} T'(u_m)$$
   を求め、$T_k \leftarrow T_k \cup \{\hat{T}_l\}$ とする。
   ただし$\hat{U}_h$については、
   $u_m = f(\ldots, v_j, \ldots),\ m(v_j) = m_j$のとき、
   $T'(u_m) = f(\ldots, v_j(\overline{m_j}), \ldots)$ とする。
3. Uは同値関係要素の集合であるから、$v_i, v_j, \hat{U}_l \in U_k$のとき、$T(v_i)$ と $T(v_j)$、また$\hat{T}_l$の少なくとも一要素と $T(v_i)$ が、それぞれ同一化可能でなければならない。
   そこで、同一化タイプ集合 T_kの各要素に対し、以下の操作を適用する。
   (a) 同一要素は一方を削除する。
   $t_i, t_j \in T_k,\ t_i = t_j \Rightarrow T_k \leftarrow T_k - \{t_j\}$
   $\hat{T}_l \in T_k, t_i, t_j \in \hat{T}_l,\ t_i = t_j$
   $\quad \Rightarrow \hat{T}_l \leftarrow \hat{T}_l - \{t_j\}$
   (b) $v_i, v_j \in U_k$のとき、v_i, v_jは同タイプとなる。
   $t_i, v_j \in T_k,\ t_i \neq \forall v_m$
   $\quad \Rightarrow T(v_j) = \{t_i\},\ T_k \leftarrow T_k - \{v_j\},$
   $\qquad \forall \hat{T}_l(\ni v_j) \leftarrow \hat{T}_l \cup T(v_j) - \{v_j\}$
   (c) 構造データ $t_{n_1}, t_{n_2} \in T_k$の場合、両者は同一化可能でなければならない。
   $f(\ldots, t_i, \ldots) \in T_k,$
   $f(\ldots, v_j(m_j), \ldots) \in T_k,$
   $t_i \neq \forall v_m(m_m)$
   $\quad \Rightarrow T_k \leftarrow T_k - \{f(\ldots, v_j(m_j), \ldots)\},$
   $\qquad T(v_j) = \{t_i\},$
   $\qquad T_l(\ni v_j) \leftarrow T_l \cup t_i - \{v_j\}$
   $f(\ldots, v_i(m_i), \ldots) \in T_k,$
   $f(\ldots, v_j(m_j), \ldots) \in T_k$
   $\quad \Rightarrow$ 節内解析の操作 3と同様に
   $\qquad v_i, v_j$ について Uを更新
   (d) 引数の一つのみが異なる構造データは一つにまとめ、異なる部分をタイプ集合で表す。
   $f(\ldots, t_i, \ldots), f(\ldots, t_j, \ldots) \in \hat{T}_l$
   $\quad \Rightarrow \hat{T}_l \leftarrow \hat{T}_l \cup \{f(\ldots, \{t_i, t_j\}, \ldots)\}$
   $\qquad\qquad - \{f(\ldots, t_i, \ldots), f(\ldots, t_j, \ldots)\}$

(e) 同一化候補タイプ集合$\hat{T}_l \in T_k$の要素が一つになった場合、$v_i \in T_k$は必ずこれと同タイプになる。
$\hat{T}_l \in T_k,\ \hat{T}_l = \{t_i\}$
　$\Rightarrow T_k \leftarrow T_k \cup \{t_i\} - \{\hat{T}_l\}$
$\hat{T}_l \in T_k,\ \hat{T}_l = \{v_i\}$
　$\Rightarrow T_k \leftarrow T_k \cup \{v_i\} - \{\hat{T}_l\}$
　　$\forall T_m(\ni v_i)$について
　　$T_k \leftarrow T_k \cup T_m$, T_m, U_mは削除

4. 3での各操作で更新されたU_k, T_lがあれば、それぞれ2, 3から繰り返す。
新規情報が得られなくなったら、5へ。

5. T_kの要素がv_iおよび$\hat{T}_{l_j}$の場合、v_iのタイプはこの節内で一意に決定されない。したがって、$\hat{T}_{l_j}$の各要素のうち、型が判明しているものの集合で表す。

(a) $\forall v_i \in \forall T_k$について
$T(v_i) = \bigcup_{\forall \hat{T}_{l_j} \in T_k} \hat{T}_{l_j}$

(b) 決定した$T(v_i)$を、他の$T(v_j)$に波及させる。
　$\forall v_j$について$v_i \in T(v_j),\ T(v_i) \neq \emptyset$
　$\Rightarrow T(v_j) \leftarrow T(v_j) \cup T(v_i) - \{v_i\}$

(c) $T(v_i)$について、まだv_jが含まれている場合、この部分についてはタイプ不明とする。
$\exists v_j \in T(v_i)$
　$\Rightarrow T(v_i) \leftarrow T(v_i) \cup \{unknown\}$
　　$- \bigcup_{\forall v_j \in T(v_i)} \{v_j\}$

6. 5の適用によってもタイプが判明しなかったv_iについては、プロセス内情報だけではタイプ不明である。
$\forall v_i, T(v_i) = \emptyset$
　$\Rightarrow T(v_i) = \{unknown\}$
$\forall f(\ldots, v_i(m_j), \ldots) \in T(\forall v_m)$
　$\Rightarrow T(v_m) \leftarrow T(v_m)$
　　$\cup \{f(\ldots, unknown, \ldots)\}$
　　$- f(\ldots, v_i(m_j), \ldots)$

7. $\forall T(v_i)$について、3a, 3d と同様に集合要素の削減・融合を図る。
$t_k, t_l \in T(v_i),\ t_k = t_l$
　$\Rightarrow T(v_i) \leftarrow T(v_i) - \{t_l\}$
$f(\ldots, t_k, \ldots), f(\ldots, t_l, \ldots) \in T(v_i)$
　$\Rightarrow T(v_i) \leftarrow T(v_i) \cup \{f(\ldots, \{t_k, t_l\}, \ldots)\}$
　　$- \{f(\ldots, t_k, \ldots), f(\ldots, t_l, \ldots)\}$

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

変数は 節番号:変数名（モード）で表記。
モードは　i: *in*,　o: *out*,　?: *unknown*

図 4: stack/2 のタイプ解析結果

並列プロセス stack/2 のタイプ解析結果を、図 4に示す。

5　一括送信手法

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を実現する手法について説明する。

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　実現方法

P_cについて、4.3 節の結果より$m(v_i) = in$であるとすると、v_iはP_pで具体化され、P_cで参照される。また、4.4 節の結果より判明しているv_iのタイプのうち、$v_j(in)$である部分は、このプロセス内で具体化される。したがって、v_iの具体値のうちP_cで参照する部分は、v_iの判明タイプのうち$v_j(in)$を除いた部分である。

そこで、実行時に実際に生成されたv_iの具体値のうち、これに該当する部分のみを一括送信する KL1 コードを生成する。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、$m(v_i) = out$のときは、v_iはP_cで具体化され、P_pで参照される。この場合はP_pの解析結果から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して

一括送信を行う。

5.2 一括送信コード生成手順

一括送信ゴールは送信対象v_iに対し、v_iの具体値である構造データ中で送信対象となる要素を順に送信バッファに格納した後、バッファ内容を一括送信する。

これを実現するには、具体値の要素の型が得られたタイプ$T(v_i)$の各要素t_kのいずれかと一致した時、送信対象とすればよい。そこで、各t_kについて以下のいずれかの送信データ格納節を生成する。

1. t_kがアトミック型 $(atom, integer, \texttt{[]})$
   具体値がt_kならバッファに格納する。

2. t_kがコンス型
   具体値がコンスなら、car および cdr を新たな引数として再帰呼出を行い、各々をバッファに格納する。

3. t_kがファンクタ型
   具体値がファンクタ$functor/m$なら、ファンクタ名をバッファに格納し、各引数をゴールの新たな引数として再帰呼び出しを行い、各々をバッファに格納する。

4. t_kが不明 $(unknown)$
   その要素は送信対象とせず、外部参照ポインタを送信する。

5. t_kが$v_j(out)$
   v_jについても現在この一括送信処理を行っているプロセスが具体化するため、引続きデータ格納処理を行う必要がある。
   そこで、v_jの送信データ格納節を呼び出し、現在着目している具体値の格納処理を行わせる。呼び出される格納節は、$T(v_j)$の各要素に対し、本節の手法を適用して生成する。

例えばプロセス `stack/2` 第1引数の場合、4章での解析結果 (図4) より、一括送信コードは図5のようになる。

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。

```
packsend_stack_2_1_top(V,H)
 :- packsend_stack_2_1_count_init(V,S),
    packsend_stack_2_1_pack_init(V,S,H).
packsend_stack_2_1_count_init(V,S)
 :- packsend_stack_2_1_count(V,0,S).
packsend_stack_2_1_pack_init(V,S,H)
 :- wait(S) | generic:new(packbuf,B,S,H),
    packsend_stack_2_1(B,V,B1), packsend_end(B1).
packsend_end(B)
 :- generic:sendbuf(B).

packsend_stack_2_1(B,V,B1)
 :- inline:"guard_unbound(%0,%f)":[V+any]
  | generic:put_in_exref(B,V,0,B1).
otherwise.
packsend_stack_2_1(B,V,B1)
 :- V = [] | generic:put_atomic(B,V,B1).
packsend_stack_2_1(B,V,B3)
 :- V = [V0|V1] | generic:put_cons(B,V,B1),
    packsend_stack_2_1_cons(B1,V0,B2),
    packsend_stack_2_1(B2,V1,B3).

packsend_stack_2_1_cons_top(V,H)
 :- packsend_stack_2_1_cons_count_init(V,S),
    packsend_stack_2_1_cons_pack_init(V,S,H).
packsend_stack_2_1_cons_count_init(V,S)
 :- packsend_stack_2_1_cons_count(V,0,S).
packsend_stack_2_1_cons_pack_init(V,S,H)
 :- wait(S) | generic:new(packbuf,B,S,H),
    packsend_stack_2_1_cons(B,V,B1),
    packsend_end(B1).

packsend_stack_2_1_cons(B,V,B1)
 :- inline:"guard_unbound(%0,%f)":[V+any]
  | generic:put_in_exref(B,V,1,B1).
otherwise.
packsend_stack_2_1_cons(B,V,B2)
 :- V=push(V0) | generic:put_functor(B,V,B1),
    generic:put_unused(B1,V0,B2).
packsend_stack_2_1_cons(B,V,B2)
 :- V=pop(V0) | generic:put_functor(B,V,B1),
    generic:put_unused(B1,V0,B2).
```

※ バッファサイズの計算部分は省略

図 5: `stack/2` 第1引数の一括送信コード

1. PVM 版 KLIC+イーサネットで結合した SparcStation2 台
2. AP1000 版 KLIC+64 台構成の AP1000

前者は ICOT より公開されている処理系であり、後者は前者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3種類を用いた。

`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。

`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、分岐が生じる毎に細粒度の負荷分散を行う。

`nqueen` N-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

表 1: SparcStation+イーサネットでの実行結果

	一括送信なし		一括送信あり	
セル	時間 (秒)	通信回数	時間 (秒)	通信回数
`stack`				
1	0.07	0	—	—
2	85.8	6504	19.8	1503
`mastermind`				
1	0.23	0	—	—
2	222.7	19327	79.2	6691
`nqueen`				
1	39.8	0	—	—
2	98.3	4733	19.8	671

表 2: AP1000 での実行結果

	一括送信なし		一括送信あり	
セル	時間 (秒)	通信回数	時間 (秒)	通信回数
`stack`				
1	1.7	0	—	—
2	24.1	130004	10.3	30003
`mastermind`				
1	2.0	0	—	—
2	22.7	190762	11.2	49501
4	15.7	263667	7.9	47081
8	11.3	278663	5.2	48872
16	9.9	290703	4.5	52496
32	8.7	296751	4.3	55245
`nqueen`				
1	42.7	0	—	—
2	24.8	4104	23.2	369
4	12.6	6576	11.9	427
6	9.0	7439	8.7	760
8	8.6	7496	8.0	346
10	7.8	7713	7.4	193
11	5.7	8508	5.6	809

れる。

6.2 実行結果

一括送信の有無による実行時間と物理通信回数の変化を測定した。結果を表 1, 2 に示す。ただし処理系により問題のサイズは異なっている。

`stack` や `mastermind` のように通信回数が多いプログラムの場合、一括送信により通信回数が 1/3 ~ 1/6 に減少した。この結果、AP1000 では 2 倍程度、通信オーバヘッドの大きいイーサネット環境では 3 ~ 4 倍程度の速度向上が得られた。

これらのプログラムに対し、`nqueen` では負荷分散の粒度が大きく、通信は最初の負荷分散時および最後の解の個数収集時にしか行われない。このため通信回数は 1/7 ~ 1/40 と大幅に減少したにもかかわらず、AP1000 では数%の速度向上しか得られなかった。しかしイーサネット環境では速度向上が 5 倍になり、ほぼ台数に見合った実行時間が得られた。

7 おわりに

本論では、並列論理型言語 KL1 の通信最適化手法と、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、必要な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。

性能評価の結果、AP1000 では最大 2 倍、通信オーバヘッドが大きいイーサネット環境では 3 ~ 5 倍と、良好な性能向上が得られた。後者の場合には `nqueen` において解収集のボトルネックが解消されるなど、本手法は非常に効果的であった。

今後の予定として、大規模なプログラムでの本手法の性能評価や、手法の改良などを検討している。

謝辞

日頃御討論頂く富田研究室の諸氏に感謝致します。また、並列計算機 AP1000 の実行環境を御提供頂きました (株) 富士通研究所、および数々の情報と助言を頂いた (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致します。

参考文献

[1] K. Ueda. Guarded horn clauses:a parallel logic programming language with the concept of a guard. Technical report, ICOT, 1986.

[2] K. Ueda and T. Chikayama. Design of the kernel language for the parallel inference machine. *The Computer Journal*, Vol. 33, No. 6, pp. 494–500, 1990.

[3] T. Chikayama, T. Fujise, and D. Sekita. A portable and efficient implementation of KL1. In *Proc. 6th Intl. Symp. PLILP'94*, pp. 25–39, 1994.

[4] K. Ueda and M. Morita. Moded Flat GHC and its message-oriented implementation technique. *New Generation Computing*, Vol. 13, No. 1, pp. 3–43, November 1994.

[5] A. K. Bansal and L. Sterling. An abstract interpretation scheme for logic programs based on type expression. In *Proc. Intl. Conf on FGCS'88*, pp. 422–429, 1988.