特別研究報告書

並列論理型言語処理系 KLIC における 単方向通信の実装と評価

指導教官　　富田眞治 教授

京都大学工学部情報工学科

高山 雅陽

並列論理型言語処理系 KLIC における
単方向通信の実装と評価

高山 雅陽

内容梗概

近年、大規模なプログラムを高速に実行するために、並列計算機や、並列実行を前提としたプログラミング言語の開発が行なわれるようになってきた。そのうちの一つである並列論理型言語 KL1 は、知識情報処理プログラムなどを記述するために開発された言語で、シンタックスが単純であり、並列記述が容易であるなどの特徴がある。また並列推論マシン上の OS や多くの応用プログラムが KL1 を用いて記述されることにより、高度な並列処理への有効性も証明されている。

ところが、KL1 で記述されたプログラムの多くは並列推論マシン PIM 上でしか実行できなかった。そこで、KL1 で記述されたプログラムの動作環境をさまざまな計算機上に実現するために、(財) 新世代コンピュータ開発機構 (ICOT) により KLIC と呼ばれる処理系が開発された。この処理系は、KL1 言語をいったん C 言語に翻訳し、機械語への翻訳は機種ごとの C コンパイラに任せるという方式をとることによって低レベルの最適化を行なっている。そのことにより、KLIC は高い移植性と高性能を同時に実現している。KLIC は、当初は PVM 通信ライブラリを用いてワークステーション上に実装されたが、現在では、分散メモリ型並列計算機 AP1000 上などにも移植が行なわれ、実行が可能になっている。

KLIC では、上に述べたような方法をとることにより低レベルに関しては比較的効率の良いコードが得ることができる。ところが、実際には頻発する細粒度通信によるオーバヘッドが大きく、実行速度の面で手続き型言語に及ばないという問題がある。そこで、この処理系を実用的なものとするためには通信回数を減少させるような工夫が必要となる。実際、今までにも一括送信方式や無条件一括送信方式などのさまざまな方法が考えられ、通信の高速化が行なわれてきた。さらに、単方向通信という高速化の手法が提案され、いくつかの実験から、この手法を用いることにより通信の性能向上が期待できることが報告されている。

本研究では、単方向通信のKLIC上への実装を行なった。単方向通信は、1対1の単純な通信では非常に効果的であることがすでに実証されている。しかし、複雑な通信においては通信データの正確な受渡しを保証するためにさまざまな処理を必要とし、その際のオーバヘッドに関しての十分な考察も行なわれていない。そこで、そのような場合についても評価を行なうこととした。さらに、単方向通信の今後における拡張の可能性についても考察を行なった。

Evaluation for Implementation of *Oneway-send* on KLIC (Implementation of Parallel Logic Language)

Masaaki Takayama

Abstract

Nowadays, parallel computers and programming languages for parallel execution are being developed in order to run large programs speedy. KL1 is a concurrent logic programming language designed in order to write parallel processing programs. It has simple syntax, and we can describe parallel execution easily in the language. The operating system of the parallel inference machines(PIM) and various application programs are described in KL1. Then KL1 reduce the effort required to develop the parallel processing software.

However most of programs written in KL1 were able to run only on PIM. Hence, to execute KL1 programs on various computers, the portable implementation called KLIC is developed by ICOT. The KLIC compiles KL1 programs into C programs, then compiles them using C compilers of the host system, and then links the objects with runtime libraries. Therefore, KLIC has high portability. Originally, KLIC was implemented on workstations with PVM parallel comunication library, and now, it is ported on distributed-memory multiprocessor AP1000.

KLIC makes rather efficient codes in assembler level, but because of overhead caused by frequent fine-grained communications, it does not run faster than other sequential languages. To solve this problem, techniques for reducing messages are required. In fact, various method for fast communication, such as *pakking-send-method*, have been invented. And another method for fast communication called *oneway-send-method* is suggested, and it is reported that we can expect to considerably reduce overhead by using it.

We tryed to implement *oneway-send-method* on KLIC runtime system. *Oneway-send-method* is a very efficient system in simple comunication, but in complicated cases, it requires various procedure for correct communication. We also dealt with such cases and evaluate the performance of *oneway-send-method* with some simple programs. And we also gave consideration for *oneway-send-method*

in complicated cases and expansion of *oneway-send-method*.

並列論理型言語処理系 KLIC における 単方向通信の実装と評価

目次

1　はじめに

KL1 は、第五世代コンピュータプロジェクトの核言語として開発された並列論理型言語である。現在までに並列推論マシン上の OS やさまざまな応用プログラムが KL1 を用いて記述され、高度な並列処理に対する有効性が実証されている。

しかし、 KL1 で記述されたプログラムは並列推論マシン PIM 上でしか動作しなかったため、KLIC と呼ばれる KL1 の処理系が開発され、並列通信ライブラリを用いた分散処理系がワークステーション上などに実装されている。KLIC は、KL1 言語をいったん C 言語に翻訳し、機械語への翻訳は機種ごとの C コンパイラに任せる方式をとることにより低レベルの機械依存の最適化を行なっている。この結果、移植性を損なわずに高性能を実現している。しかし、従来の KLIC ではプログラムの並列実行過程において細粒度通信の頻発などによるオーバヘッドのため、実行速度の面では手続き型言語処理系に及ばず、実用的な処理系の実現のためには通信の方法に改良の必要があった。

KLIC において、通信回数を削減することによって高速化を達成する方法として、一括送信方式 [1] や単方向通信 [2] という手法が提案され、すでに前者は実装されている。本研究では、単方向通信を KLIC 上に実装し、その性能を評価した。

以下第 2 章では並列論理型言語 KL1 とその処理系 KLIC について簡単に紹介し、第 3 章では単方向通信の手法について述べる。第 4 章ではその性能の評価及び考察をそして、単方向通信の拡張における展望を述べる。

2　背景

本節では、並列論理型言語 KL1 [3] とその処理系 KLIC [4] について簡単に紹介する。

2.1　並列論理型言語 KL1

2.1.1　プログラム構造

KL1 は Flat GHC に基づく言語で, そのプログラムは以下の形をした節の集合で表される。

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

H,G,B は述語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、各々は述語名と引数からなる。

KL1 において、実行の単位はゴールと呼ばれる。与えられたゴールの述語名と引数の数が一致するクローズヘッドの引数を同一化し、その節のすべてのボディゴールを次の新しいゴールとする。ただし、ガードゴールにはその節が適用されるための条件が記され、そのすべての条件が成立した時にボディゴールが実行される。各ボディゴールは並列実行が可能で、この過程を繰り返すことによりプログラムが実行される。

また、実行中のゴールで未具体化変数への参照が行われるとそのゴールは実行を中断し、 別のゴールを選択して実行が続けられる。その後、その未具体化変数が具体化されると、中断したゴールが再開される。

2.1.2 データ構造

KL1 のデータ構造は次の 3 つのものからなる。

- 論理変数 ：
  他の変数や具体的な値を指す参照ポインタで表され、特定の型は持たない。論理変数は具体化によって対象へのポインタが格納され、一度具体化されると他の値に書き換えられることはない。
- アトミックデータ ：
  シンボリックアトムや整数のような値そのものに意味を持つデータである。
- 構造データ ：
  複数のセルからなり、それぞれのセルには任意の型のデータや参照ポインタを格納することができる。代表的なものにファンクタやリストがあげられる。

2.2 KL1 処理系 KLIC

KLIC は KL1 プログラムを様々な計算機上で動作させるために ICOT が開発した KL1 処理系である。KLIC は, KL1 プログラムをいったん C プログラムに変換しオブジェクトコードの生成はターゲットマシンの C コンパイラが行うという方式をとっている。そのため移植性の高い処理系となっており、現在では PVM[5] を用いたワークステーション上や AP1000 上などでの実行が可能になっている [1][6]。

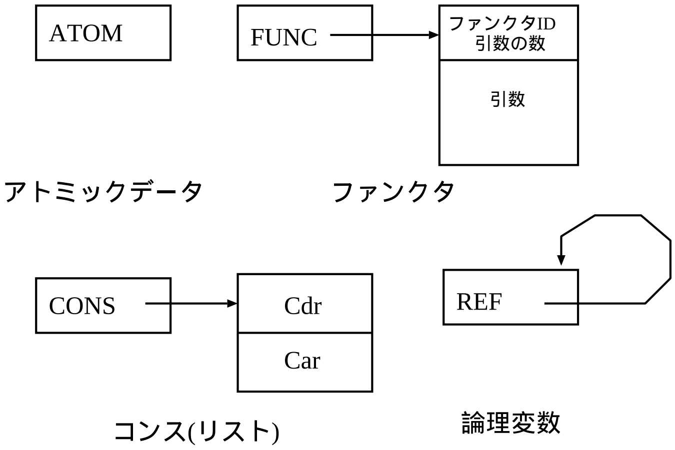


図 1: 基本データ構造

実行プログラムはKL 1プログラムから得られたオブジェクトコードとKLICのランタイムライブラリをリンクすることで作成される。実行ゴールの選択などのプログラム全体の制御はカーネルと呼ばれるランタイムルーチンが行なう。

2.2.1　データ構造とその働き

KLICには大別して以下の3種類のデータ構造が存在する。

- 基本データ構造 ：
  KL1のデータを表現するもので、タグつきセルに指された構造体で表される。タグには、REF(変数、generator, consumer object(後述)へのポインタ)、CONS(リストを実現するコンスセルへのポインタ)、FUNC(ファンクタまたはdata object(後述)へのポインタ)、ATOM(シンボリックアトムまたは整数へのポインタ)の4種類がある(図1)。

- Generic object ：
  KL1のさまざまなデータ構造を実装するために、KLICにはKL1の基本データの他にgeneric objectと呼ばれるデータ構造が存在する。 generic objectはデータ領域とそのデータを操作するメソッドと呼ばれる手続きから構成されている(図2)。generic objectは用途に応じて3種類に分けられ、それぞれdata object、generator object、consumer objectと呼ばれる。

  consumer objectとgenerator objectは、ともに未具体化変数を表すREFタグつきセルに指されており、このような状態をobjectがREFにフック

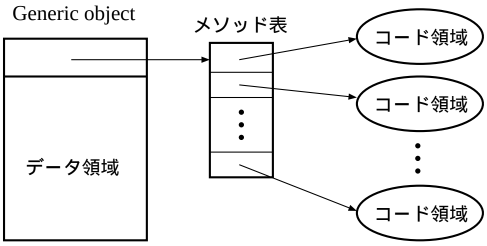


図 2: Generic object

していると呼ぶ。generator object は、自身がフックしている変数に対する読みだしに対して仕事を行ない、consumer object は、フックしている変数に対する書き込みに対して仕事をする。例えば generator object がフックする REF に対して参照が行なわれると GENERATE メソッドが起動され、consumer object がフックする REF に対して具体化が行なわれると consumer object の UNIFY メソッドが起動される。

- ゴールレコード：
  KL1 の実行の単位であるゴールは、ゴールレコードとして表される。ゴールレコードは次に実行されるゴールへのポインタ、対応する実行コードへのポインタ、引数からなる (図 3)。
  プログラムの実行は、ゴールキューにつながれたゴールレコードを 1 個取り出し、そのゴールレコードに対応するコードを実行し、得られたボディーゴールをゴールキューにつなぐことで行なわれる (図 4)。実行中ゴールで未具体化変数 X を参照すると、このゴールは X にフックして中断し、新たなゴールをゴールキューから取り出すことにより実行が続けられる。その後 X が具体化されると X にフックしていたゴールは実行可能ゴールキューに戻される。実行可能ゴールキューからゴールがなくなるとプログラムは終了する。

2.2.2 分散環境における KLIC の働き

計算モデル

KLIC 分散処理系の計算モデルは、局所メモリを持ったノードがネットワークを介して互いにメッセージ通信を行ないながら処理をするものである。負荷分散はプログラム中に `goal(`$Args$`)@node(`N_c`)`(ゴールをノード N_c に送って実行させる) と明記することによって行なわれ、自動負荷分散は行なわれない。すべて

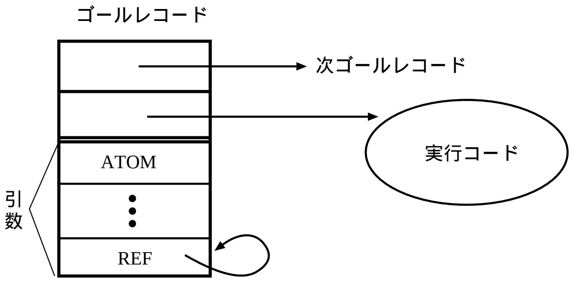


図 3: ゴールレコード

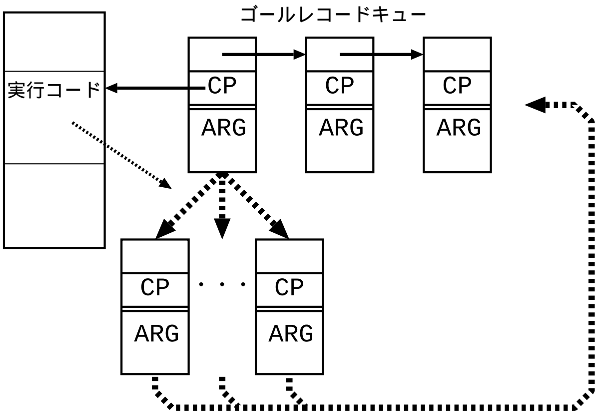


図 4: ゴールの実行

のノードで実行可能なゴールがなくなるとプログラムは終了する。

外部参照ポインタ

KLIC では変数や構造データの送信が行なわれるとノード間に渡るポインタが生成される。これは外部参照ポインタと呼ばれる。外部参照ポインタは具体的には次のような場合に生成される。

- ゴール送信時に引数に未具体化変数が含まれていると、この引数を指す外部参照ポインタが生じる。
- 外部参照ポインタに対して参照が生じると参照先に読みだし要求を行ない、データを送信してもらう。この時、データがポインタを含む構造データの場合外部参照ポインタが生じる。
- 外部参照ポインタに具体化が行なわれた時、具体値がポインタを含む構造データの場合外部参照ポインタが生じる。

外部参照ポインタへの読みだし

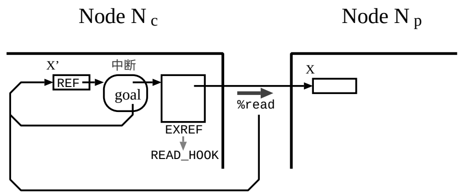
外部参照ポインタに対してデータの参照が起こった場合の処理について、ゴール `goal(X)` がノード N_p からノード N_c に送られた場合を用いて説明する。

ノード N_c が `goal(X)` を受信すると generator object(`EXREF`) を作成し、外部参照ポインタを実現する。`EXREF` は N_p 上の `X` を指している。ゴール `goal(X)` の実行中に外部参照ポインタに対して参照が起きた場合の処理系の動作は以下の通りである。ただし、以下では `EXREF` がフックする REF を `X'` とする。

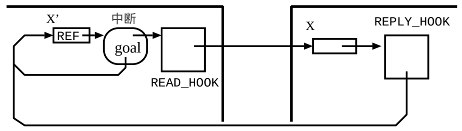
1. 外部参照ポインタへの参照によって `EXREF` の `GENERATE` メソッドが起動される。`GENERATE` メソッドは返信先を用意し、外部参照ポインタが参照するノードに読みだし要求メッセージ (read メッセージ) を送る。ゴール `goal(X)` は実行を中断し `X'` にフックする。この後、外部参照ポインタを表現する generator は consumer object(`READ_HOOK`) に変化する (図 5(a))。
2. read メッセージをノード N_p が受信すると、`X` が具体化済であれば answer_value メッセージによって `X` の具体化された値を返信する。`X` が未具体化変数であれば `X` に consumer object (`REPLY_HOOK`) をフックし、値の返信は中断される (図 5(b))。その後 `X` が具体化されると `REPLY_HOOK` の `UNIFY` メソッドが起動され answer_value メッセージが送られる (図 5(c))。
3. answer_value メッセージをノード N_c が受信すると `X'` とメッセージを同一化し、その結果、中断ゴール `goal` は実行可能ゴールキューにつながれる。そして、ノード間ガーベジコレクションのためにノード N_p に対して release メッセージを送り、`X` が `goal` から参照されなくなったことを知らせる (図 5(d))。

送信先での変数の具体化

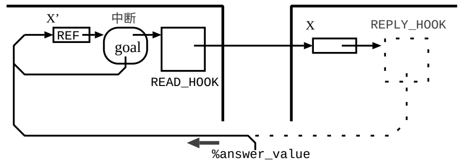
`EXREF` に対し、具体化が行われると `UNIFY` メソッドが起動し、具体値を相手ノードに送信する。このようなメッセージを unify メッセージと呼ぶ。



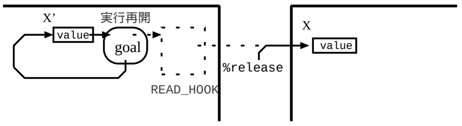
(a)readメッセージの送信



(b)Xの具体化待ち



(c)具体値の返信



(d)外部参照ポインタの解放

図 5: 外部参照ポインタへの参照

3　通信の高速化手法

3.1　KLIC の通信における問題点

KLIC では、変数の具体化や参照は必要となった時点で送信を要求するという方式をとっているため、粗粒度のメッセージ通信を前提とした並列計算機を対象としたとき、頻発する細粒度通信によるオーバヘッドが大きな問題となる。実際、KLIC には次のような問題点がある。

- 外部参照ポインタを介した参照が行なわれた場合、1 回の参照につき read, answer_value, release という 3 回のメッセージ通信が必要であり、効率が悪い。
- answer_value メッセージや unify メッセージは、送られるデータが構造データの場合最外側の値しか送信されず、構造データの参照の際にはメッセージ数が増大する。(図 14)

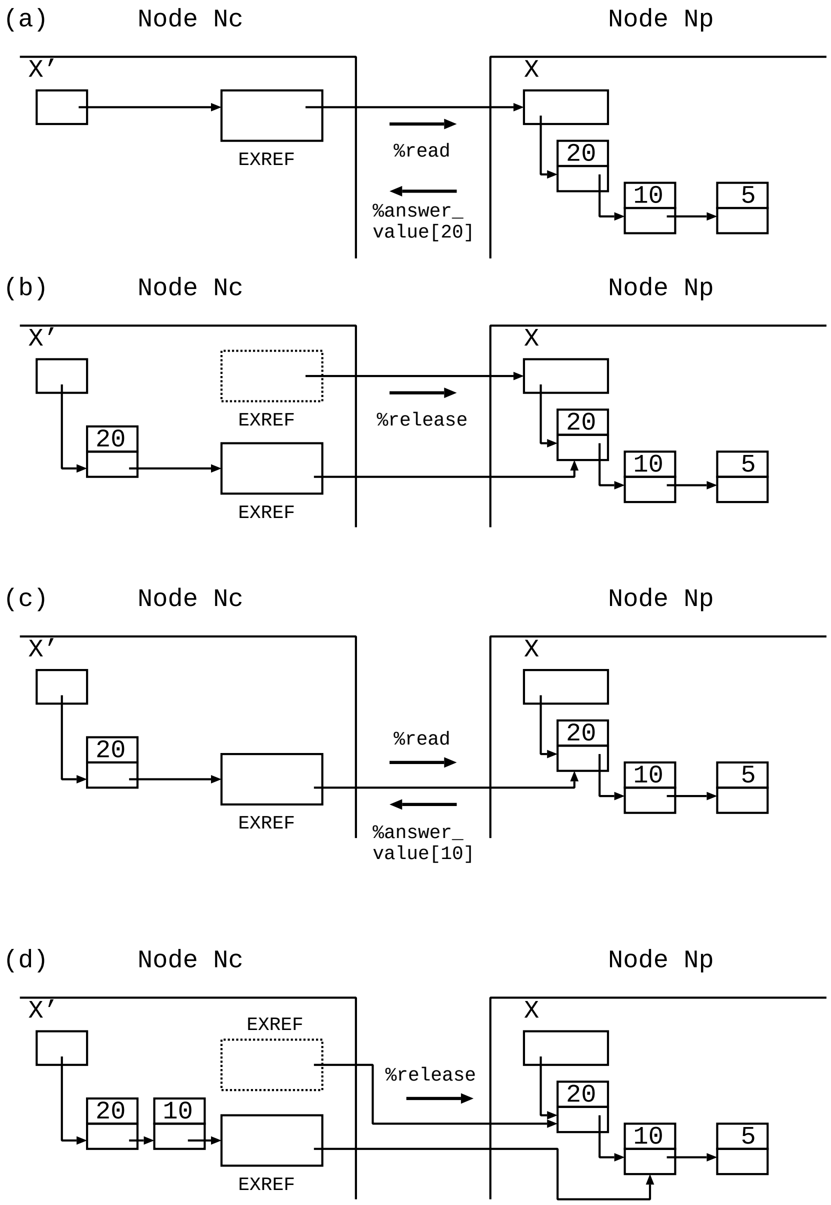

(a)
Node Nc
Node Np
X′
X
EXREF
%read
%answer_
value[20]
20
10
5
(b)
Node Nc
Node Np
X′
X
EXREF
%release
20
EXREF
20
10
5
(c)
Node Nc
Node Np
X′
X
%read
20
EXREF
%answer_
value[10]
20
10
5
(d)
Node Nc
Node Np
X′
X
EXREF
%release
20
10
EXREF
20
10
5

図 6: 構造データの参照

3.2　通信回数の削減

以上のような問題を解決するためにメッセージ数を削減する方法として構造データを参照する際、次のような処理を行なうことが考えられる。

- 送信要求に対して具体化されているデータを必要なだけバッファリングして送信する方法。
- 送信要求を待たずに送信者が具体値を送信する方法。(単方向通信)

前者は送受信者の独立性が高く、バッファに多数のメッセージが溜る場合に特に有効であり、すでに一括送信方式[1][7](図7)として実装されている。一方後者は、送受信者の依存性が強い場合でも比較的有効な場合がある。そこで、単方向通信[2]をKLIC上に実装し、その効果を評価することとした。

3.3　単方向通信の実現

3.3.1　単方向通信の手順と有効な条件

受信者がリスト型の変数を待つプロセスであることが明らかであれば、通信を単方向で行なうことができる。すなわち、受信者が送信者に変数の参照を行なうと、送信者がその変数を具体化するたびにその値を受信者に送信すれば良い。この方法で、参照の回数と物理メッセージの個数は等しくなる。KL1のプログラムの特徴として、リスト型のデータは連続して次の値を参照する場合が多く、単方向通信を行なうことにより、

単方向通信は、リスト型データ以外にも引数にファンクタを1個以下しか持たないファンクタのようなデータにも有効であるが、解析や処理が困難であるので、今回の実装はリスト型のデータの通信に限るものとした。

以下に、どのように単方向通信を実現するかを具体的に述べる。

単方向通信用 object

単方向通信実現のため、次の3つのgeneric objectを用意した。

- `ONEWAY_EXREF` ：

  単方向通信用の外部参照ポインタを実現するためのgenerater object。

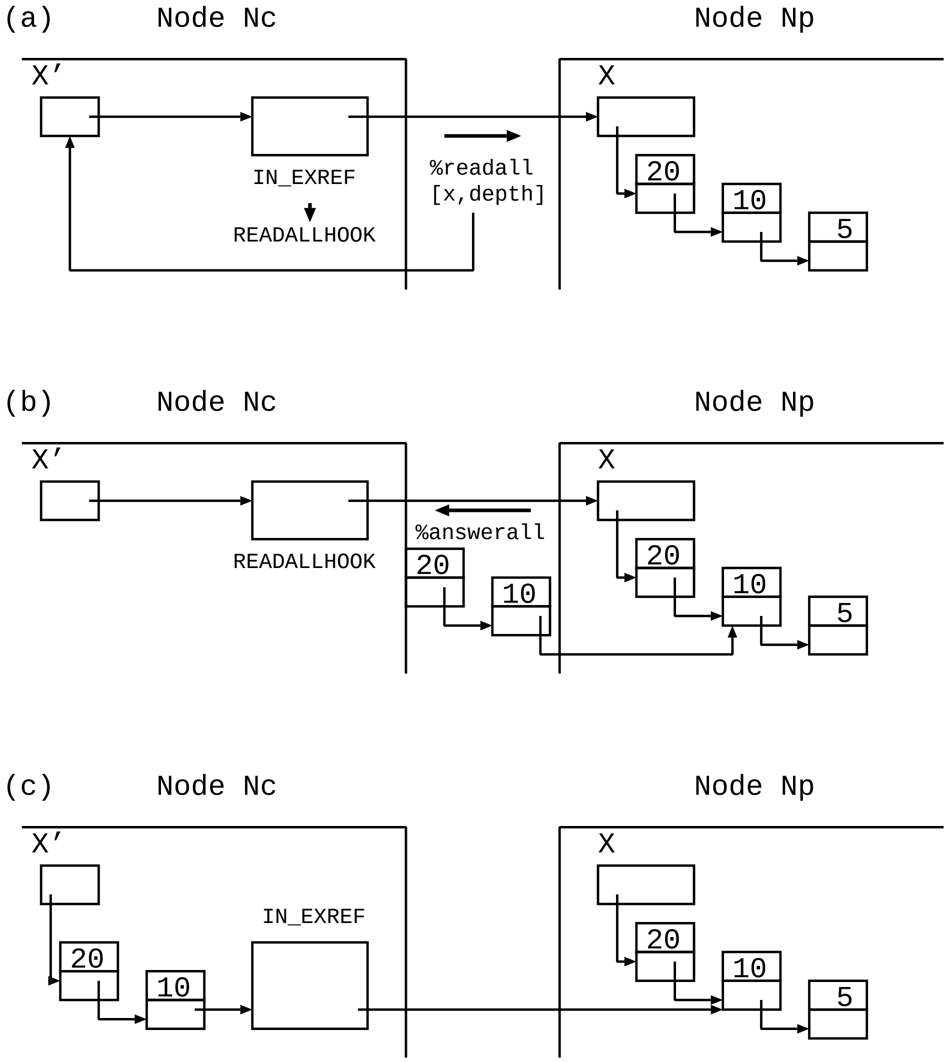

(a)
Node Nc
Node Np
X′
X
IN_EXREF
READALLHOOK
%readall
[x,depth]
20
10
5
(b)
Node Nc
Node Np
X′
X
READALLHOOK
%answerall
20
10
20
10
5
(c)
Node Nc
Node Np
X′
X
IN_EXREF
20
10
20
10
5

図 7: 一括送信方式

- `ONEWAY_READ_HOOK` ：
  単方向通信の受信処理をするための consumer object。次節に述べるような理由から内部データとして参照先以外から受けとったデータへのポインタなどを持つ。

- `ONEWAY_REPLY_HOOK` ：
  単方向通信の送信処理をするための consumer object。内部データとして送信先を記した表などを持つ。

単方向通信の起動

ノード N_cが `goal(X)` を受信すると generator object(`ONEWAY_EXREF`) を作成し、単方向通信用の外部参照ポインタを実現する。`ONEWAY_EXREF` は N_p上のデータ `X` を指している。また、`ONEWAY_EXREF` がフックする REF を `X'` とする。

1. `ONEWAY_EXREF` への参照によって `ONEWAY_EXREF` の `GENERATE` メソッドが起動される。`GENERATE` メソッドは返信先を用意し、外部参照ポインタが参照するノードに読みだし要求メッセージ (oneway_read メッセージ) を送る。ゴールは実行を中断し `X'` にフックし、外部参照ポインタを表現する generator は consumer object (`ONEWAY_READ_HOOK`) に変化する。(図 8(a))

2. oneway_read メッセージをノード N_pが受信すると、`X` が具体化済のレベルまで oneway_answer_value メッセージによって `X` の具体化された値を 1 個ずつ返信する。`X` の末端が未具体化変数であればこの未具体化変数に consumer object (`ONEWAY_REPLY_HOOK`) をフックし、値の返信は中断される。その後 `X` が具体化されると `ONEWAY_REPLY_HOOK` の `UNIFY` メソッドが起動され、oneway_answer_value メッセージが送られる。また、この具体化された変数に未具体化変数へのポインタが含まれていると、さらに `ONEWAY_REPLY_HOOK` をその未具体化変数にフックして具体化されるのを待つ。(図 8(b),(c))

3. oneway_answer_value メッセージをノード N_cが受信すると `X'` とメッセージを UNIFY する。この際、未具体化変数が含まれていると、その未具体化変数に `ONEWAY_READ_HOOK` をフックし、さらに次の oneway_answer_value

メッセージが届くのを待つ。(図 8(b),(c))

3.4 単方向通信の問題点

単方向通信では、通常では送信側から受信者側へのメッセージの通信しか行なわれない。そのため通常の通信では read メッセージと release メッセージによって正確に保証されている外部参照ポインタの内容が、送信されたデータが受信側に到着するまでの時間のずれなどから曖昧なものとなる。そこで、単方向通信を実現するには、送信者か受信者に不都合が起こった時に通信データの正確なやりとりを保証しなければならない。具体的には、特に次のような場合が問題である。

- 受信側ゴールが終了し、途中でデータを必要としなくなった場合。
- 送信側ゴールが他のノードに移動した場合。
- 受信側ゴールが他のノードに移動、分割した場合。

それぞれの場合についての解決方法を以下に述べることにする。

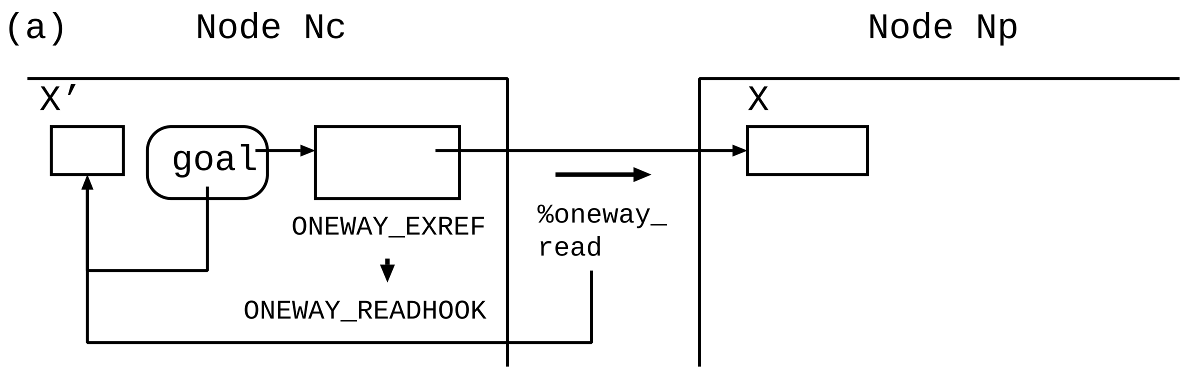
受信側ゴールの終了

1. 受信側ゴールが終了すると `ONEWAY_READ_HOOK` の `READSTOP` メソッドが起動される。`READSTOP` メソッドは、送信先に単方向通信の終了を要求する oneway_read_stop メッセージを送信する。(図 9(a))

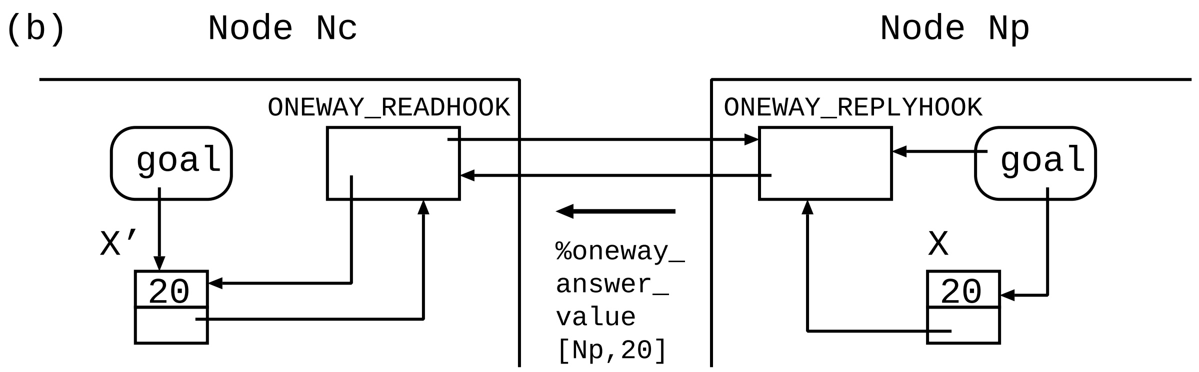
2. `ONEWAY_REPLY_HOOK` は、oneway_read_stop メッセージを受信するとノード N_c への送信を終了し、ノード N_c に送信を終了したことを知らせる oneway_answer_stop メッセージを送信する。(図 9(b))

3. N_c の `ONEWAY_READ_HOOK` は、oneway_answer_stop メッセージを確認するまで N_p の `ONEWAY_REPLY_HOOK` から単方向通信によって送られてくる oneway_answer_value メッセージの受信処理を行なう。この処理によって N_c は N_p が送信したデータをすべて受けとることが保証されるので、受信されないメッセージによるメモリの占有などが避けられる。

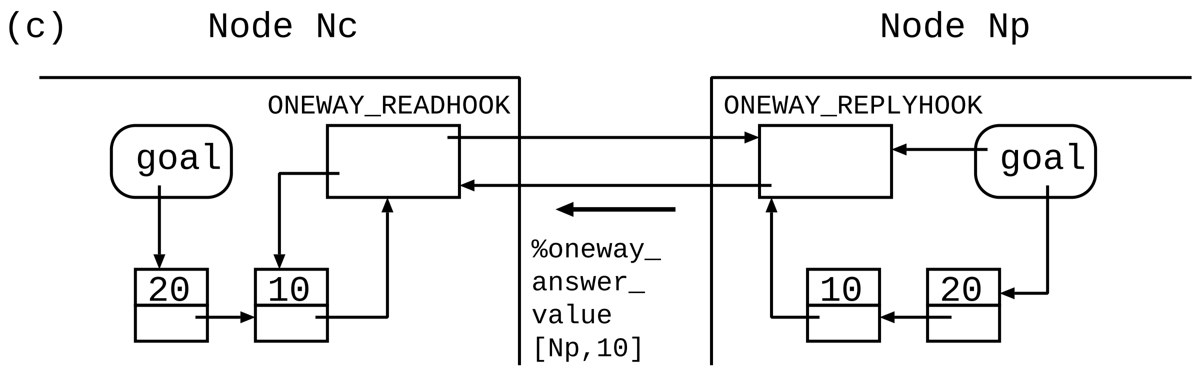
送信側ゴールの移動



oneway_readメッセージの送信は最初の参照の機会にのみ行なわれる。

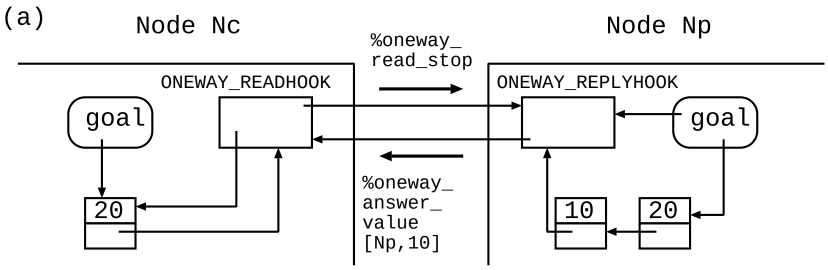


Xが具体化されその値が送信されても通信は続行される。

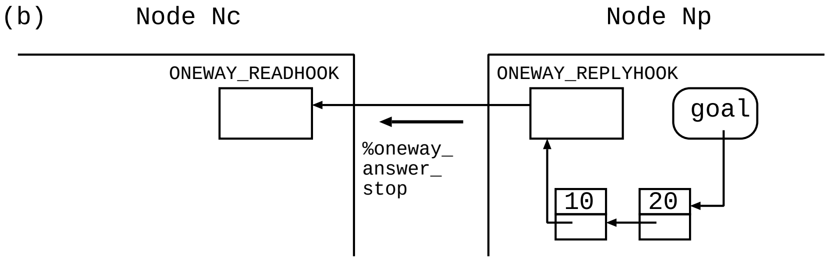


ONEWAY_REPLYHOOKは、未具体化変数が具体化されるたびに
oneway_answer_valueメッセージによってその値を送信する。

図 8: 単方向通信



ONEWAY_READHOOKがoneway_read_stopメッセージを送信する間にも
ONEWAY_REPLYHOOKがoneway_answerを送る可能性がある。



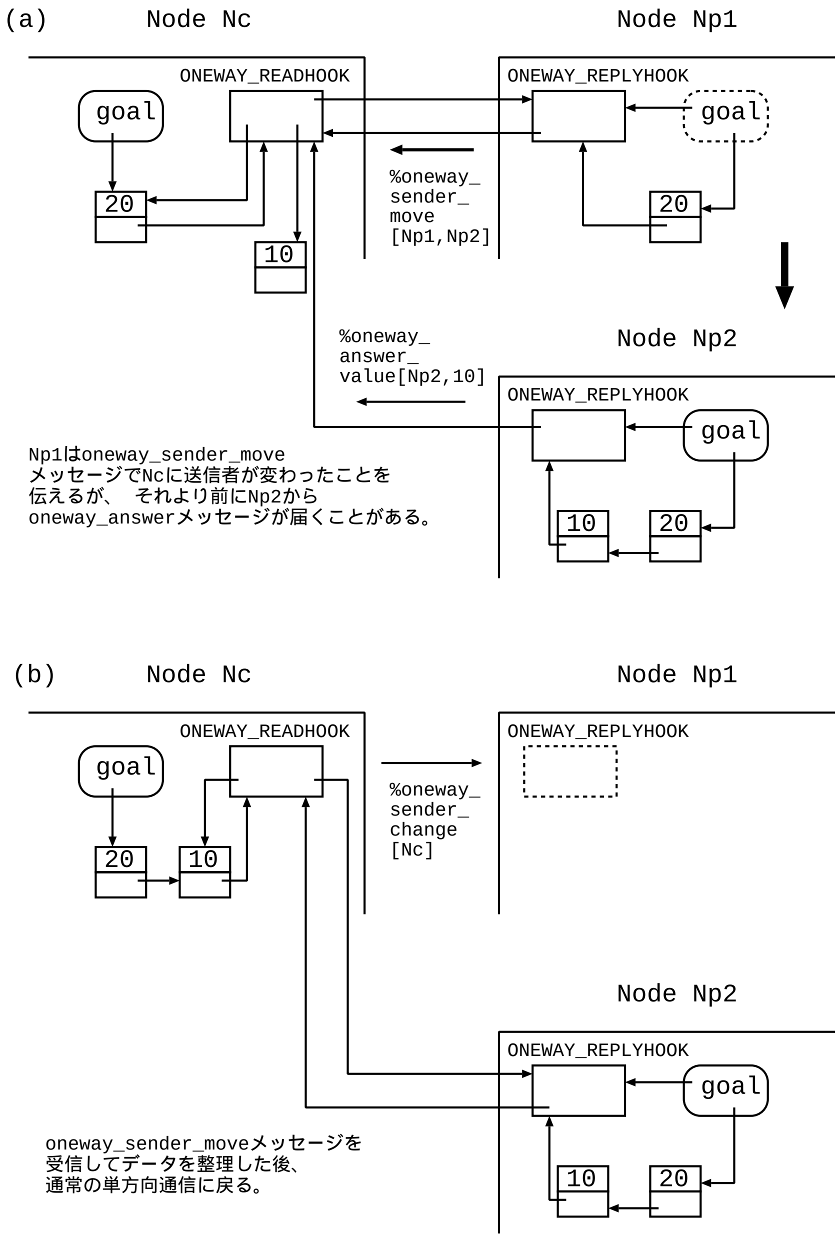
ONEWAY_REPLYHOOKはoneway_answer_stopメッセージを送って送信を終了し、
ONEWAY_READHOOKはこのメッセージを確認してから終了する。

図 9: 受信側ゴールの終了

1. 送信側ゴールが N_{p1} から N_{p2} に移動すると、送信側ノードが変わったことを知らせるための oneway_sender_move メッセージが N_{p1} から N_c へ送信される。(図 10(a))

2. N_c では、oneway_sender_move メッセージを受けとる前に `ONEWAY_READ_HOOK` の参照先 N_{p1} とは異なるノード N_{p2} から oneway_answer_value メッセージを受けとる可能性がある。その時はそのデータをいったん `ONEWAY_READ_HOOK` の内部データとして格納しておく。(図 10(a))

3. N_c が oneway_sender_move メッセージを受けとると、`ONEWAY_READ_HOOK` の外部参照ポインタを N_{p2} に書き換え、そのことにより N_{p1} の変数がすでに N_c から参照されなくなったことを知らせる oneway_sender_change メッセージを N_{p1} に送信する。また、N_c の `ONEWAY_READ_HOOK` に先だって N_{p2} から受けとって格納しておいたデータがあれば、それが今までに N_{p1} から送られてきたデータの続きであるとわかるので、自身のフックする変数と UNIFY



Np1はoneway_sender_move
メッセージでNcに送信者が変わったことを
伝えるが、 それより前にNp2から
oneway_answerメッセージが届くことがある。

oneway_sender_moveメッセージを
受信してデータを整理した後、
通常の単方向通信に戻る。

図 10: 送信側ゴールの移動

する。(図 10(b))

受信側ゴールの移動

ここでは受信側ゴールが移動した場合について述べる。

1. N_{c1}から N_{c2}へ受信側ゴールが移動すると、N_{c2}に作られた `ONEWAY_READ_HOOK` は、受信ノードの変更を N_pに知らせる oneway_receiver_move メッセージ

を送信する。N_pの ONEWAY_REPLY_HOOK はその間も変数の具体化が起これば N_{c1}にデータを送信するので、受信側ゴールが N_{c1}から N_{c2}へ移動してから N_pが oneway_receiver_move を受信するまでに N_{c1}に送られたデータは N_{c1}が中継する必要がある。そこで、N_{c2}の ONEWAY_READ_HOOK は、この時点では N_{c1}の変数を指している。(図 11(a))

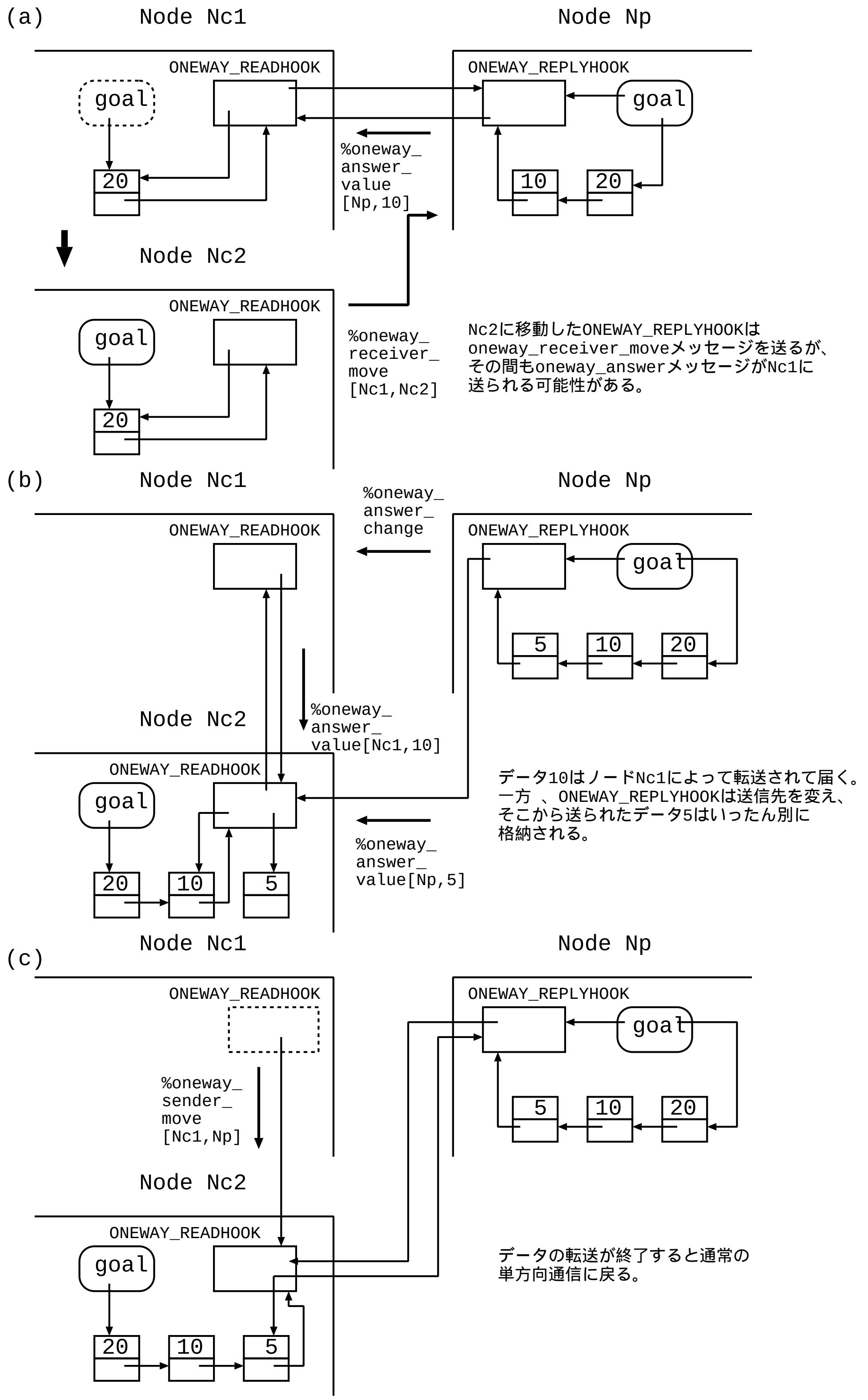
2. N_pの ONEWAY_REPLY_HOOK は oneway_receiver_move メッセージを受けとると送信先を N_{c1}から N_{c2}に変更する。さらに N_{c1}に送信先を変更したことを告げる oneway_answer_change メッセージを送り、以降は N_{c1}への oneway_answer_value メッセージの送信は行なわない。(図 11(b))

3. N_{c1}の ONEWAY_READ_HOOK は、受信側ゴールが N_{c2}に移動した後は、oneway_answer_value メッセージを受信するごとに N_{c2}にその oneway_answer_value メッセージを転送する。また、N_pから oneway_answer_change メッセージを受けとると、これより後のデータは N_pから N_{c2}に直接送られることがわかるので、送信ノードが移動したことを知らせるためのメッセージ oneway_sender_move を N_{c2}に送る。(図 11(c))

4. N_{c2}には N_pと N_{c1}の両方から oneway_answer_value メッセージが送られてくることになるが、はじめ、N_{c2}の ONEWAY_READ_HOOK の参照先は N_{c1}にあるので、自身の未具体化変数は N_{c1}からのメッセージと UNIFY し、N_pからのデータは ONEWAY_READ_HOOK の内部データとして格納しておく。その後、N_pからの oneway_sender_move を受信すると、送信側ゴール移動時と同様の手続きを行ない、通常の単方向通信に戻る。(図 11(c))

短期間における送信側ゴールの連続移動

送信側ゴールが短期間に N_{p1}から N_{p2}、N_{p2}から N_{p3}へ移動した場合、受信側ノード N_cには参照先以外のノード N_{p2}や N_{p3}からデータを受信することがある。この場合、受信されたデータは、送信先ごとに区別されて ONEWAY_READ_HOOK の内部データとして格納されるので、N_{p1}、N_{p2}それぞれのノードから送られてくる oneway_sender_move を受信することにより、正しくデータを整理することができる。(図 12)

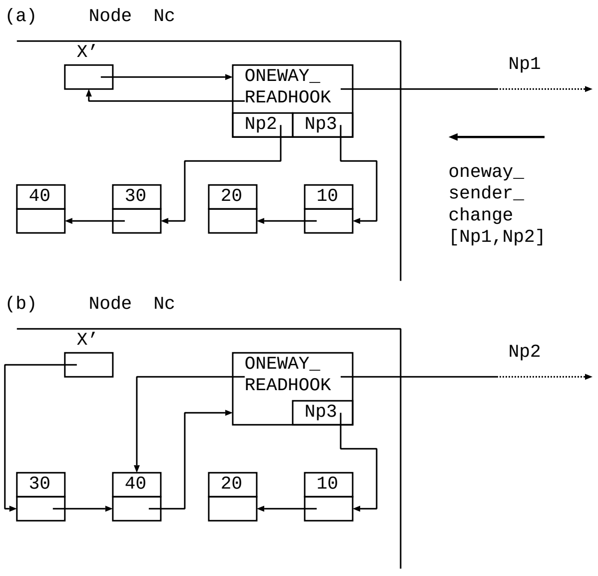


Nc2に移動したONEWAY_REPLYHOOKは
oneway_receiver_moveメッセージを送るが、
その間もoneway_answerメッセージがNc1に
送られる可能性がある。

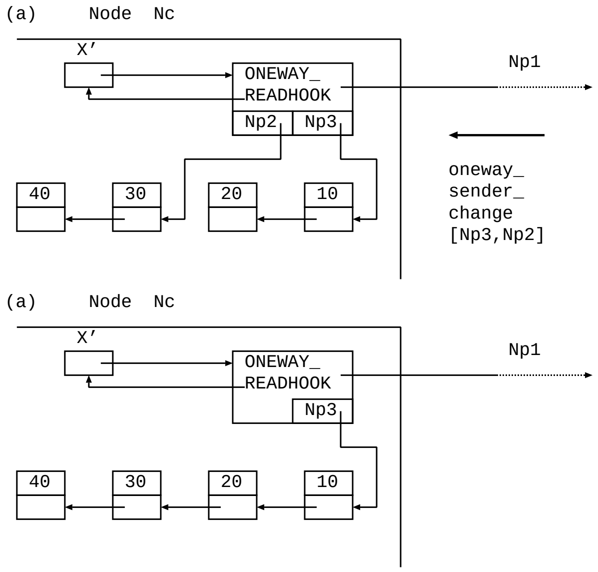
データ10はノードNc1によって転送されて届く。
一方 、ONEWAY_REPLYHOOKは送信先を変え、
そこから送られたデータ5はいったん別に
格納される。

データの転送が終了すると通常の
単方向通信に戻る。

図 11: 受信側ゴールの移動



送信側ノードの連続移動(1)



送信側ノードの連続移動(2)

図 12: 送信側ゴールの連続移動

短期間における受信側ゴールの連続移動

受信側ゴールが短期間に N_{c1} から N_{c2}、N_{c2} から N_{c3} へ移動した場合、N_{c3} で必要となるデータが N_{c1} に送信されることがある。このデータは、いったんノード N_{c2} に送られた後、そこを経由して N_{c3} へ送られる。(図 13)

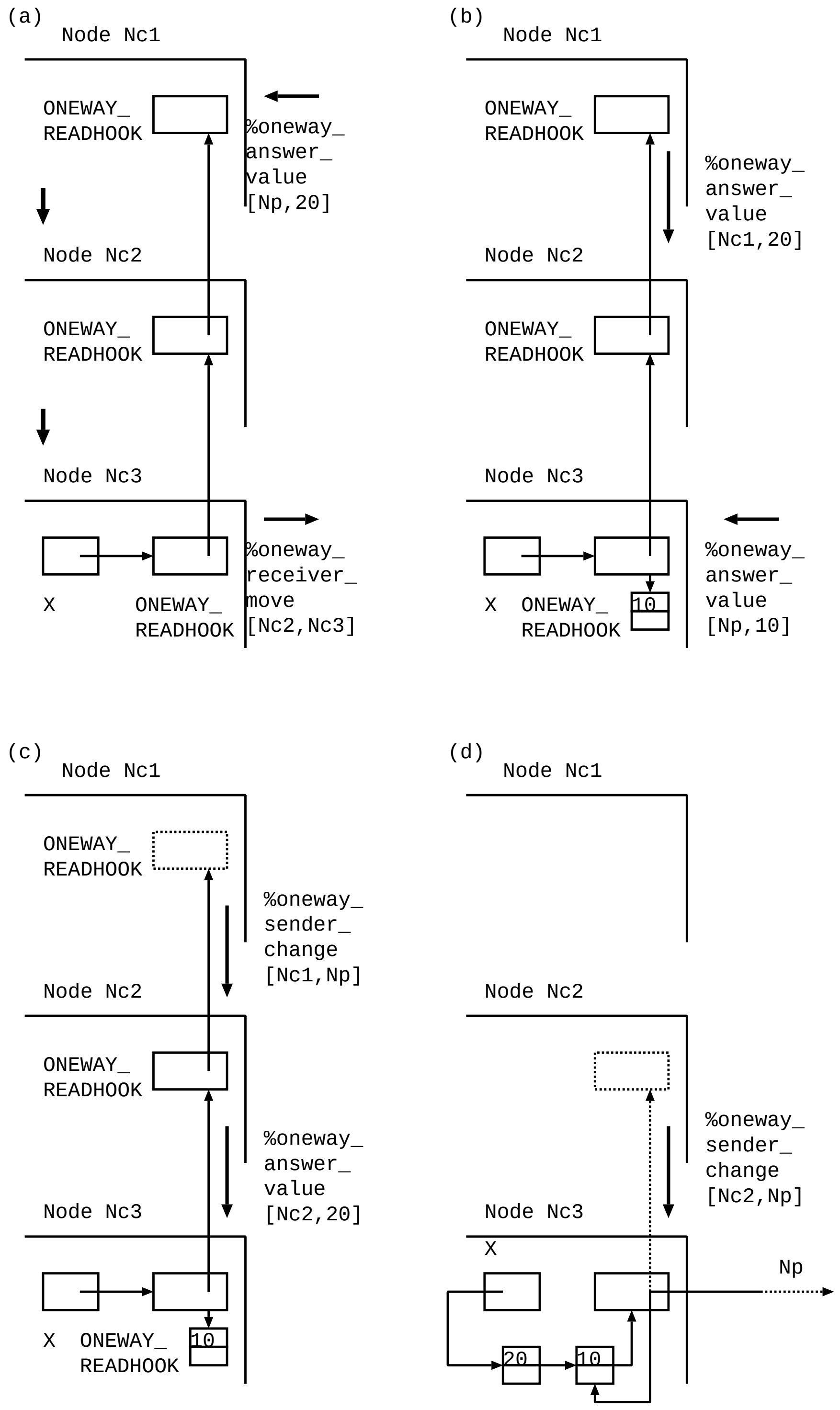


図 13: 受信ゴールの連続移動

4　評価と考察

本研究ではKLIC上に単方向通信を実装し、その上でKL１プログラムを実行させる予定であったが、完全に動作させせることができなかった。そこで、単方向通信の仕様に関する検証と考察を行なうことにする。

4.1　オーバヘッドの考察

単方向通信は1対1の通信で、一方が送信者、他方が受信者と明確に決まっている時は間違いなく有効である。ただし、送信者、受信者が移動することにより、従来の通信では必要なかったメッセージ(具体的にはoneway_sender_move, oneway_receiver_moveなど)を送るなどの処理が必要になる。また、受信者の終了の場合では、通常の通信では送信されることのなかったはずのデータがメッセージとして送信されていることになる。
これらの処理によるオーバヘッドは事態の変更を知らせるメッセージ(oneway_read_stop, oneway_sender_move, oneway_receiver_move)が送信されてから受信されるまでの時間に特に大きく依存すると考えられる。なぜなら、その時間が多いほど余計な処理を受けなければならないoneway_answer_valueメッセージの数が増えるからである。このことより、単方向通信は通信時間が速いハードウェアにより向いていると言える。

KLICでは、他のノードから参照される変数は自ノードの輸出表に登録される。外部参照ポインタは、参照する先のノードとそのノードの輸出表における参照先の変数が登録されているアドレスで表される(図14)。そのため実際には次のような問題が起きている。

1. 送信側ゴールがN_{p1}からN_{p2}に移動すると、N_{p1}はoneway_sender_moveメッセージを送信するが、この時点ではN_{p2}における変数の輸出表のアドレスがわかっていない。
2. 受信側ゴールがN_{c1}からN_{c2}へ移動すると、ノードN_{c1}はN_{c2}へのデータの転送を行なうことになるが、この時点ではN_{c2}における変数の輸出表のアドレスがわかっていない。

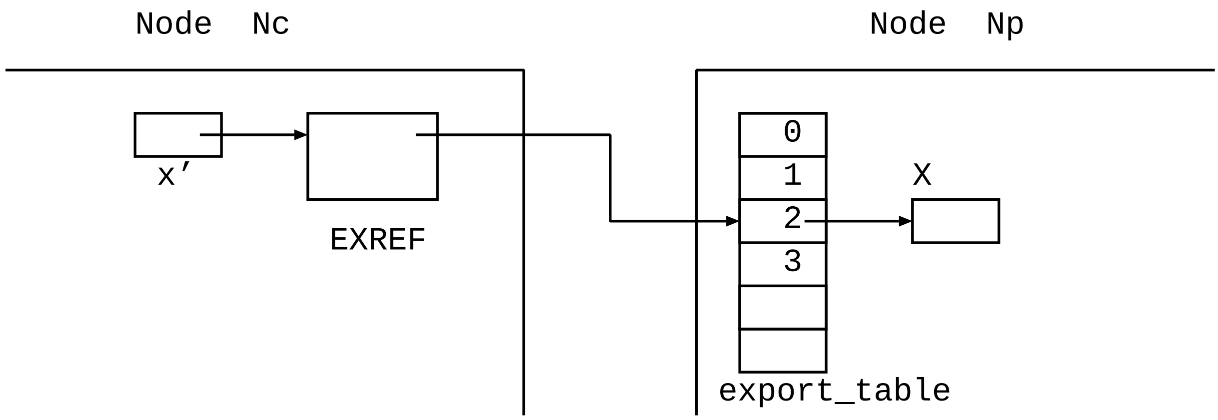


図 14: 輸出表

これらの問題を解決するために現実にはそれぞれ、

1. N_{p2}における変数の輸出表のアドレスを N_{p1}に知らせるメッセージ
2. N_{c2}における変数の輸出表のアドレスを N_{c1}に知らせるメッセージ

を送ることとなった。これにともない、これらのメッセージの到着までの時間もオーバヘッドに加わることになり、当初よりもオーバヘッドの見積りが増加した。もし、さらにゴールのノード間移動が連続して起こるようになると、さらに複雑な処理が必要となり、オーバヘッドが加速度的に増加し、単方向通信の効果が得られないどころかかえって性能が落ちることも考えられる。これを避けるためには、単方向通信を用いる場合ゴールのノード間移動などが頻発しないようなプログラムに限る必要がある。

4.2 単方向通信の拡張

ファンクタへの応用

単方向通信の拡張の方法の一つとしてファンクタに応用することが考えられる。引数の中に常に 1 つのファンクタしかないようなファンクタはリストと同様に単方向通信が非常に効果的である。しかし、そのようなファンクタを分析することは困難である。また、引数に 2 個以上のファンクタを持つようなファンクタでは、そのたびに consumer object を増やさなければならず、通信のインターフェイスが困難となる。そこで、単方向通信をファンクタに適用するには、前者のようなファンクタをユーザによるオプション指定により、行なうような形にするのがもっとも簡単であろう。

一括送信方式との併用

一括送信方式は前にも触れたとおり、送信要求に対して具体化されているデータを必要なだけバッファリングして送信する方法である。この方式と単方向通信を併用するためには、送信要求に対し、ある程度バッファにデータをためてから送信する方法が考えられる。この場合、バッファにデータをためるために先に具体化されたデータの転送を遅らせる遅延送信 [7] を用いることになる。ただし、送受信者のデータに依存関係がある場合には、バッファにデータがたまらず送信が滞るため、解析が必要であろう。

5 おわりに

本論では、並列論理型言語 KL1 の処理系 KLIC に単方向通信の実装を試みた。単方向通信の本来の目的はメッセージ数の削減による通信の高速化である。ところが、ゴールのノード間移動などに対処するためさまざまな通信や処理が必要となった。そのためプログラムによっては性能の低下するおそれもある結果となった。

しかし、単方向通信は大抵の場合は有効であり、さらなる高速化の手法として、遅延送信を含めた一括送信方式との併用は有力な考え方である。このような拡張でメッセージ数をさらに削減することも可能であろう。

謝辞

本研究の機会を与えてくださった富田眞治教授に深甚な謝意を表します。適切な私事を賜わった中島浩助教授、折りに触れ有用な御意見をいただいた森眞一郎助教授に心から感謝致します。

また、親切に御指導頂いた大野和彦氏、伊川雅彦氏をはじめ、日頃御討論頂いた京都大学工学部情報工学科富田研究室の諸兄に感謝致します。

最後に、筆者にいろいろと助言を下さった友人諸氏に御礼致します。

参考文献

[1] 伊川雅彦, 大野和彦, 森眞一郎, 中島浩, 富田眞治: 並列論理型言語処理系

KLIC における通信の高速化 (1994).

[2] 中島浩: 並列論理型言語 (KL1) の実装最適化に関する調査研究報告書, 技術報告 5-R-1(5), 財団法人 新世代コンピュータ技術開発機構 (1994).

[3] Ueda, K. and Chikayama, T.: Design of the Kernel Language for the Parallel Inference Machine, *The Computer Journal*, Vol. 33, No. 6, pp. 494–500 (1990).

[4] Chikayama, T., Fujise, T. and Sekita, D.: A Portable and Efficient Implementation of KL1, *Proc. 6th Intl. Symp. PLILP'94*, pp. 25–39 (1994).

[5] Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., Vaidy Sunderam (訳 村田英明): PVM3 ユーザーズガイド &リファレンスマニュアル (1994).

[6] 田沼仁: 並列論理型言語 KL1 の AP1000 への実装 (1994).

[7] 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 の最適化手法, 情報処理学会研究報告, pp. 183–190 (1994).