並列論理型言語処理系 KLIC における通信の高速化

伊川 雅彦

平成 7 年 2 月

Abstract

近年、大規模なプログラムを高速に実行するために、並列計算機や並列言語の研究が盛んになっている。並列論理型言語 KL1 は、知識情報処理プログラムを記述するために開発された言語で、シンタックスの単純さや並列性を自然に記述できるなどの特徴を持つ。また現在までに並列推論マシン上の OS や多くの応用プログラムが開発され、高並列処理への有効性が実証されてきている。

しかし、KL1 で記述されたプログラムの多くは並列推論マシン上でしか動作しなかった。そこで、KL1 により実現されているプログラム動作環境を様々な計算機上に実現するために ICOT で KLIC と呼ばれる KL1 の処理系が開発され、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ている。

本研究では、KLIC の分散メモリ型並列計算機 AP1000 の上への移植ならびに、通信の高速化を行った。

KLIC の移植では通信のインタフェースが最大の問題となる。特に AP1000 への移植の場合、受信方式としての割り込み処理がセル OS の制約によりサポートされておらず、単純な移植を行うことができない。今回の移植ではメッセージの到着を定期的にチェックするポーリング方式を採用することでこの問題を解決した。

通信の高速化の一つとして、AP1000 版 KLIC 特有の問題であるポーリング方式によるメッセージの受信チェックにかかるオーバヘッドを削減するため、高速なポーリング用のマクロ命令を作成した。

さらに、頻発する細粒度通信によるオーバヘッドを軽減するため、メッセージの一括送信を行う KLIC のランタイムの拡張を行った。

これらの通信の高速化手法の効果を評価用プログラムを用いて調査した結果、ポーリングマクロ `Cprobe()` の実行速度は、従来のポーリング用関数 `cprobe()` の約 12 倍であった。また、KLIC に実装し、`nqueen` プログラムを実行したところ、その実行速度は約 2 倍となった。

一方、メッセージの一括送信に関しては、ノード間でスタック操作を行う `stack` プログラム、数字の並びを推測するゲームである `mastermind` プログラム、および `nqueen` プログラムを用いて、実行時間と物理通信回数の評価を行った。その結果、実行時間に関しては、`stack` プログラムで 2.4 倍、`mastermind` では最大 2.2 倍の速度向上が得られた。しかし、`nqueen` については演算時間が実行時間を支配するため、5 ~ 9%の速度向上しか得られなかった。また、物理通信回数については、一括送信ありの場合、一括送信なしに比べて最良で 1/40、最悪の場合でも 1/3 に減少しており、良好な結果が得られた。

Abstract

Nowadays, parallel computers and parallel languages are used to run large program fastly. KL1 is a programming language designed for ease of writing parallel processing programs. Its characteristics are the simplicity of language rules and the expressive power of parallelizm. KL1 is a language for parallel symbolic processing. The operating system of the parallel inference machines(PIMs) and various application programs are described in this language. KL1 reduce the effort required to develop the parallel processing software.

However almost KL1 programs were running only on PIM. To execute KL1 program on various computers, the portable implementation of KL1 called KLIC is developed. KLIC with PVM, pallarel communication library, is currently implemented on work stations.

In our laboratory, we ported KLIC on AP1000, a message passing parallel computer, and improve inter-PE communications.

In implementing on AP1000, the largest problem is placed in the interface for inter-PE communications. Since AP1000's Cell OS does not provide interrupt function for receive system. For that reason, porting is not simple. In order to deal with this problem, we use polling method which checks periodically if any message is arrived.

One method to improve communication is to decrease overhead of polling with faster macro instruction for polling. Another method is to expand KLIC runtime system, which we call *packing-send* method, to decrease inter-PE communications by sending messages in a lump.

We evaluate the performance of these methods with some programs. We confirm that the macro instruction for polling runs 12 times faster than original function for polling. And by using macro instruction with nqueen program, we get twice speedup.

About *packing-send* method, we examine execution speed and the number of inter-PE communications with stack, mastermind, and nqueen programs. We obtain the following results.

Execution Speed:

For stack program, we get 2.4 times speedup. For mastermind program, we get 2.0 ~ 2.2 times speedup. However, for nqueen program, we get only 5 ~ 9% speedup. The reason is that operation time dominate execution time on nqueen program, because of a small number of inter-PE communications.

The number of Inter-PE communications:

The number of Inter-PE communications becomes a maximum of 1/40 of its original value.

1 はじめに

並列論理型言語 KL1 は、第五世代計算機計画の核言語として開発され、シンタックスの単純さや並列性を自然に記述できることなどの特徴をもつ。そして、現在までに並列推論マシン上の OS や様々な応用プログラムを記述することで高並列処理への有効性を実証して来た。

しかし、従来 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。そこで、KL1 により書かれたプログラムを様々な計算機上で動作させるために、(財) 新世代コンピュータ開発機構 (ICOT) で KLIC と呼ばれる移植性の高い 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に実装されている。そこで、この KLIC 分散処理系の、分散メモリ型並列計算機 AP1000 上への移植を行った。

KLIC では 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。そのため低レベル部分の最適化に関しては、比較的効率のよいコードが得られる。しかしながら、頻発する細粒度通信などによるオーバヘッドのため、実行速度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形式の改良が必要である。そこで本研究ではこのオーバヘッドを減少させることを目的とし、メッセージの到着チェックにかかる時間の短縮と、一括送信による通信回数の削減を行うランタイムの拡張を行った。

以下第 2章では背景として並列論理型言語 KL1, その処理系 KLIC について説明し、第 3章で KLIC 分散処理系の分散メモリ型並列計算機 AP1000 への移植方式を説明する。続いて第 4章で KLIC の高速化について説明する。次に第 5章で各種の評価結果を述べ、第 6章で結論を述べる。

2 KL1 と KLIC

本節では、並列論理型言語 KL1[1][2] 及びその処理系 KLIC [3][4] の概要について述べる。

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、以下の形をした節の集合で表される。

$$H :- G_1 \cdots, G_m | B_1 \cdots, B_n$$

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 '述語名/引数の個数' で表記される。
実行の単位はゴールと呼ばれる。与えられたゴールをヘッドと同一化し、その各ボディゴール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。ガードゴールに書けるのは組込述語のみで、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。ボディゴールはそれぞれが並列に実行される。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。

2.1.2 データ構造

KL1 のデータ構造には、次の 3 つのものがある。

- 論理変数
  論理変数は参照ポインタで表現され、特定の型は持たない。具体化によって対象へのポインタが格納される。論理変数は一度具体化されると、別の値に再び具体化することはできない。

- アトミックデータ型
  アトミックデータは内部構造を持たないデータであり、値そのものが意味を持っている。アトミックデータ型にはシンボリックアトムや整数がある。

- 構造データ型
  構造データは複数のセルからなり、これらのセルには任意の型のデータ (へのポインタ) を格納することができる。構造データ型の一般的なものとしてはファンクタやリストがある。

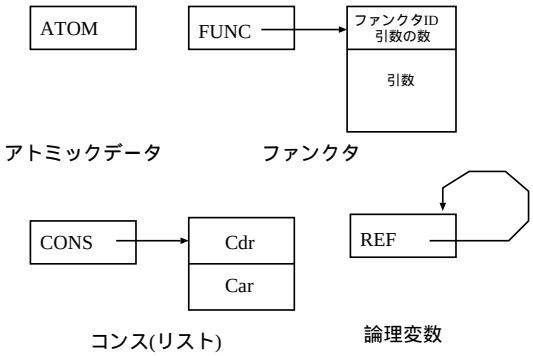

図 1: 基本データ構造

2.2 KL1 処理系 KLIC

KLIC は、ICOT で開発された KL1 の処理系であり、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 コンパイラが行う方式をとっている。よって、通信関係などの機種依存部を除き移植性の高い処理系となっている。現在汎用並列通信ライブラリである PVM[5] を用いたワークステーション版 KLIC が稼働中である。

実行プログラムは KL1 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う。

2.2.1 データ構造

基本データ構造　KLIC では KL1 のデータをタグ付きセルまたは、タグ付きセルに指された構造体で表している (図 1)。タグには REF, CONS, FUNC, ATOM の 4 種類があり表 1の様に用いられる。

表 1: セルにおけるタグ

タグ	意味
REF	変数, generator, consumer object へのポインタ。未具体化変数は自分へのポインタで実現する。
CONS	コンスセルへのポインタ。
FUNC	ファンクタまたは data object へのポインタ。
ATOM	シンボリックアトム又は整数。

ゴールレコード
実行の単位であるゴールは、ゴールレコードとして表現され、次に実行されるゴールへのポインタ、対応するコードへのポインタ、引数からなる (図 2)。

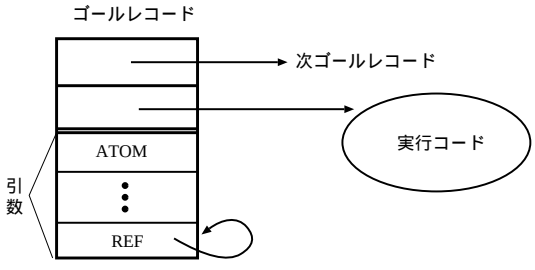

図 2: ゴールレコード

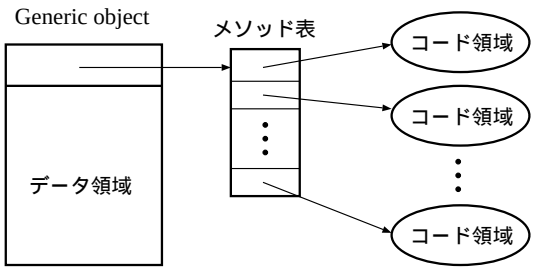

図 3: Generic object

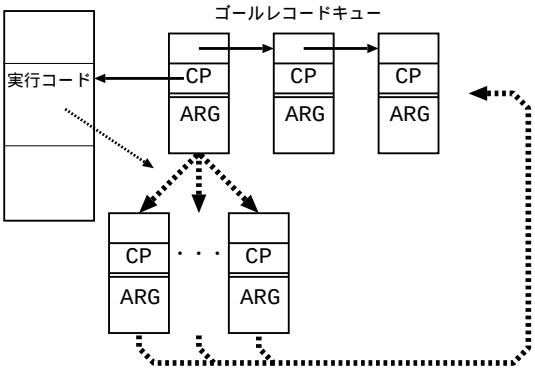

図 4: KLIC 逐次実行方式

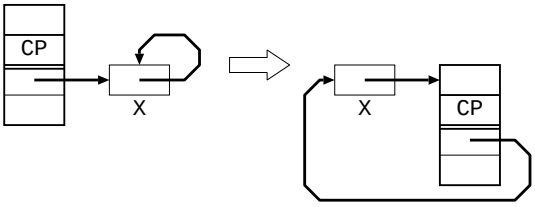

図 5: 実行ゴールの中断

Generic object

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 はデータ領域とデータを操作するメソッドと呼ばれる手続きからなり、図 3のような実装方式になっている。

generic object には data, consumer, generator の３つのタイプがあり、用途に応じて使い分けられる。consumer object と generator object は共に一種の未具体化変数を表す REF タグ付きセル (以下 REF) に指されている。このような状態を本論では object が REF にフックしているという。consumer object がフックする REF に対して具体化が行われたとき UNIFY メソッドが起動される。また generator object がフックする REF に対して、具体化が行われると UNIFY メソッドが、参照が行われると GENERATE メソッドが起動される。

2.2.2 逐次実行方式

プログラムの実行は、カーネルが選択したゴールに対応するコードを実行し、その中で得られたボディゴールをゴールキューにつなぐことで行われる (図 4)。これをリダクションという。実行中のゴールで未定義変数 X を参照すると、このゴールは X につながれ中断し (図 5)、新たなゴールがゴールキューから取り出され実行が続けられる。その後 X が具体化されると、X につながれ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無くなるとプログラムは終了する。

2.2.3 KLIC 分散処理系

KLIC 分散処理系の計算モデルは、局所メモリを持ちネットワークで結合されたノードが、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。自動負荷分散は行われず、負荷分散を行うには明示的にプログラム中に指定しなければならない。$\texttt{goal}(X_1, \cdots, X_n)\texttt{@node}(N_c)$ 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。全てのノードで実行可能なゴールがなくなると、プログラムの実行は終了する。終了の検出はいずれかのノードに割り付けられた荘園と呼ばれるプロセスが行う。

外部参照ポインタ

変数や構造データの送信が行われると、ノード間に渡る参照ポインタが生まれる。これを外部参照ポインタという。外部参照ポインタは次の場合に生成される。

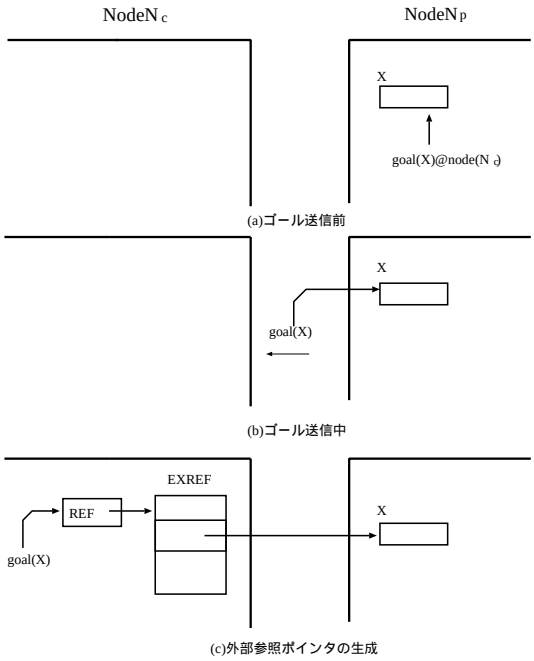

図 6: 外部参照ポインタの生成 (ゴール送信時)

- ゴールの送信時には引数も同時に送信される。このとき引数にポインタが含まれていると、この引数を指す外部参照ポインタが生じる (図 6)。
- 外部参照ポインタに対して参照が生じると参照先に対し読みだし要求を行い、データを送信してもらう。このとき、データがポインタを含む構造データであった場合に外部参照ポインタが生じる (図 7)。
- 外部参照ポインタに対して具体化が行われると、具体値を外部参照ポインタが指している変数に対して送信する。このとき、具体値がポインタを含む構造データであった場合に外部参照ポインタが生じる (図 8)。

このように外部参照ポインタを経由した参照や具体化にはメッセージ通信という機種に依存した処理を伴う。そこで外部参照ポインタを generic object を用いて実装し、カーネルの移植性を高めている。

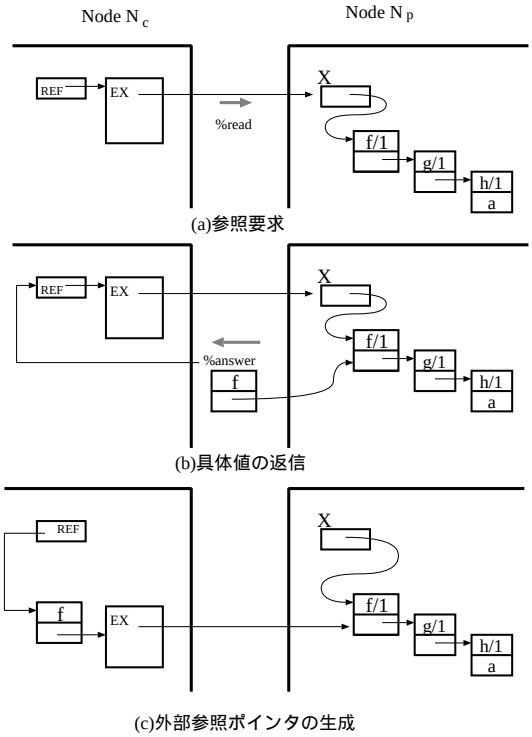

図 7: 外部参照ポインタの生成 (answer_value 送信時)

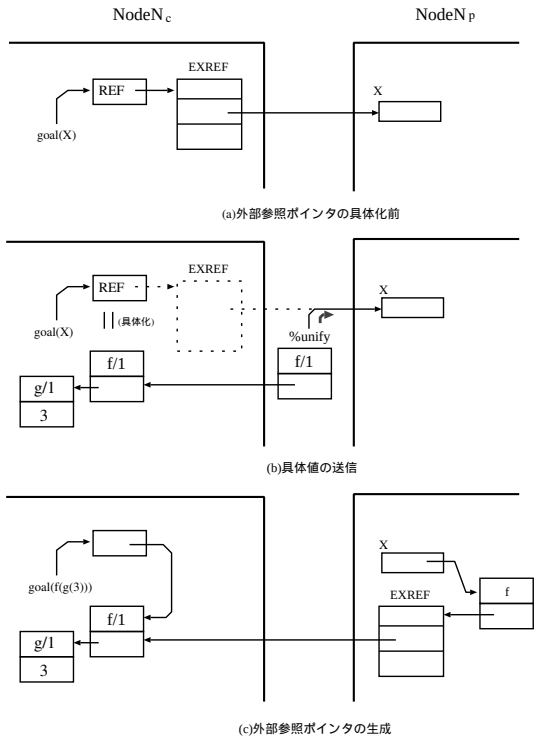

図 8: 外部参照ポインタの生成 (unify 送信時)

なおメッセージの種類やメッセージに含まれるデータ型ごとに、メッセージをデコードする関数 (デコード関数) が用意されている。メッセージにはそれらの関数へのポインタが付加されており、受信側でデコード関数を呼び出すことで受信処理が行われる

外部参照ポインタの参照値の読みだし
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' にフックし、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 9(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 9(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 9(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 9(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。これを unify メッセージと呼ぶ (図 10)。

3 KLIC の AP1000 への移植

本章ではオリジナル KLIC を基にした、AP1000 への移植について述べる。

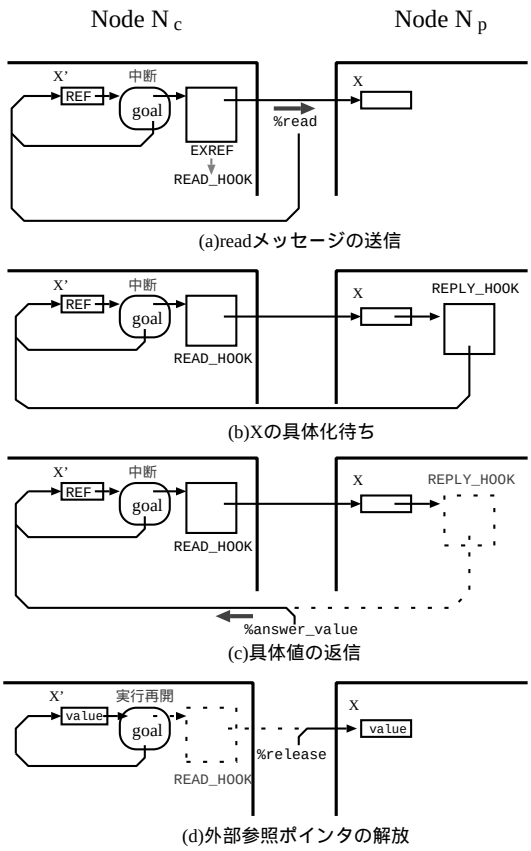

図 9: 外部参照ポインタの参照

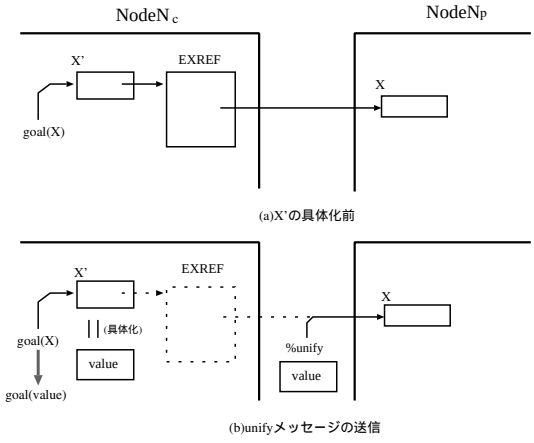

図 10: 外部参照ポインタの具体化

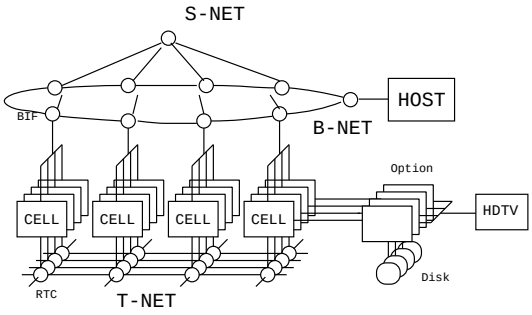

図 11: AP1000 のアーキテクチャ

3.1 AP1000 の通信機構

AP1000 は最大 1024 台のプロセッサを二次元トーラス網で接続した分散メモリ型並列計算機で、通信用のライブラリ関数と複数の通信機構を持っている (図 11)[6]。

3.1.1 標準ライブラリ

AP1000 の標準ライブラリ [7][8] として、プロセッサ間通信のための関数が用意されている。しかし、これらの通信関数はデータの送受信のタイミングをプログラマが意識することを前提としたものであり、KLIC のように不定期にメッセージが到着するシステムには適さない。また、メッセージの到着を知らせる割り込み機能は用意されておらず、ポーリング用の関数はシステムコールに伴うオーバヘッドが大きい。

3.1.2 通信機構

AP1000 ではトーラスネットワークを用いてのセル間通信はルーティングコントローラ (RTC) に対するデータの読み書きにより行われる。RTC には内部情報を格納している幾つかのレジスタがあり、受信用バッファのライトポインタ (WTP) やリードポインタ (RDP) の値がこれらのレジスタに格納されている [9]。

3.2 移植上の問題点

AP1000 には以下に述べるようなハードウェアおよび OS 上の制約があり、単純に移植を行うことはできない。

1. オリジナル KLIC は Unix 系 OS 上での動作を前提としているため、1 プロセッサ上に複数のプロセスをマッピングすることが可能である。しかし、AP1000 のセル OS のマルチプロセス機構は極めて貧弱であるため、現実的には 1 セルに 1 プロセスしかマッピングできない。

2. オリジナル KLIC では PVM のシグナル送信機能を用いて、送信者から受信者にメッセージが送られたことを知らせている。AP1000 にも PVM が実装されているが、AP1000 のセル OS の制約によりシグナル送信の機能が削除されている。

1については解決方法が見当たらず、1 セル 1 プロセスのマッピングを行うこととした。2については、受信方式として定期的にメッセージのポーリングを行う方式を用いることによって解決した。以下にそれぞれについて詳述する。

3.3 プロセスマッピング

KLIC は I/O,shoen,master,slave の 4 種類のプロセスからなる (図 12)。AP1000 版 KLIC では以下のようなプロセスマッピングを行った。以下セル n 台で実行する場合を考える。

I/O プロセス: このプロセスはホストマシンで実行される。shoen, master, slave の各プロセスを起動し、その後 I/O 処理を行う。

shoen プロセス: このプロセスは n-1 番のセルプロセッサで実行される。shoen はプログラム全体の管理を行うプロセスで、終了の検出などを行う。

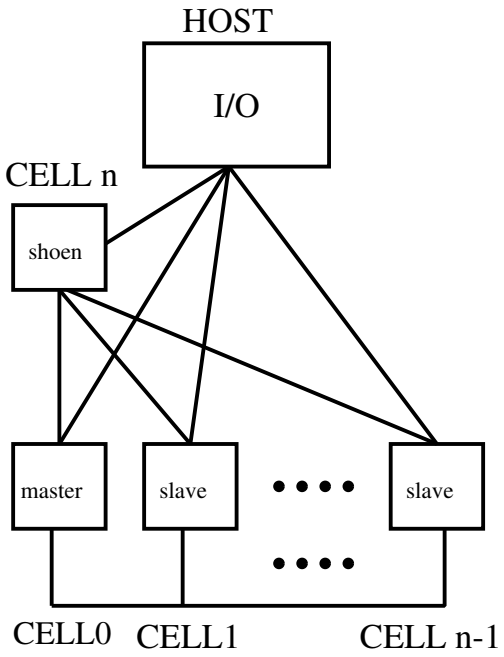

図 12: AP1000 版 KLIC における各プロセスのマッピング

master プロセス: このプロセスは 0 番のセルプロセッサに置かれる。初期ゴールである main/0 はこのプロセスで実行される。slave プロセスにゴール送信を行うことにより負荷分散が行われる。

slave プロセス: このプロセスは 1 ~ n-2 番のプロセスで実行される。起動直後は待ち状態であり、master や他の slave プロセスからゴールを受信するとそのゴールの実行を開始する。

荘園プロセスでは実際の仕事は行わないため、並列動作可能な最大ノード数は全セル台数−1 台である。

3.4 プロセス間通信機能の実装

通信処理の実装方式として AP1000 版 PVM を用いる方式と標準通信ライブラリを用いる方式の 2 通りが考えられる。オリジナル KLIC で用いている PVM の機能のうち、割り込み以外の機能に関しては AP1000 版 PVM、標準通信ライブラリ共に実現可

能である。したがって、この部分に関してはオリジナル KLIC と同等のコードを使用できる AP1000 版 PVM を用いるほうが有利である。しかし、標準通信ライブラリで AP1000 版 PVM の機能を実現するのはさほど困難なことではなく、通信性能も良い。さらに AP1000 版 PVM では PVM 以外の通信機能を用いた場合の動作が保証されておらず、将来、AP1000 のセル OS が改良されて割り込み処理がサポートされても、使用できない恐れがある。

以上のことから、本移植では機種依存部である通信関係の部分には、AP1000 の標準通信ライブラリを用いることとした。

受信方法についてはメッセージが到着しているかどうかを定期的にチェック (ポーリング) することにより解決した。ポーリングは毎リダクション後に行う。また実行可能ゴールが存在しない場合には、一定時間毎に行われる。

また本移植ではポーリング用の関数として AP1000 標準通信ライブラリ関数である `cprobe()` を用いた。

4 通信高速化手法

4.1 KLIC の問題点

KLIC の受信方式をポーリング方式で実装すると、システムは受信メッセージの有無に係わらず一定時間毎に受信チェックを行う。そのため、ポーリングのオーバヘッドが問題となる。

また、KLIC には KL1 の実行モデルによる速度低下が存在している。KLIC では変数の具体化や参照を行うノードが一般的には実行時まで判明せず、必要と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。このため、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場合、細粒度通信の頻発が大きなオーバヘッドになる。具体的にこの種の問題としては次に述べるようなものがある。

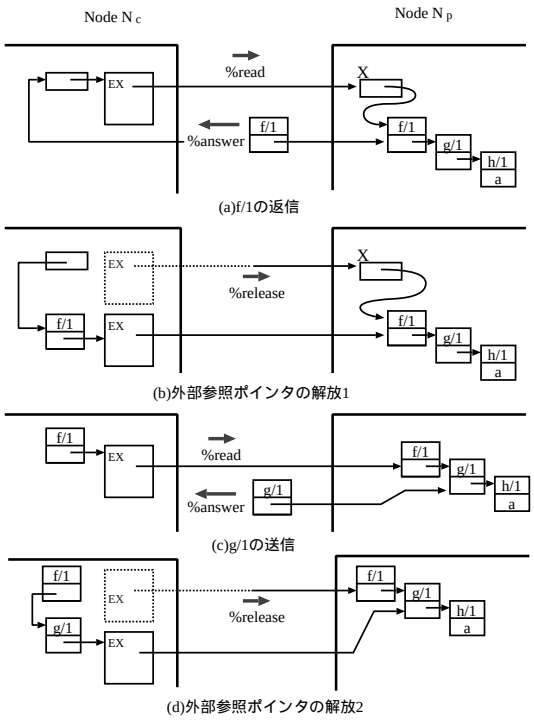

図 13: 構造データの送信 (一括送信なし)

1. 第 2.2節で述べたように実体が他ノードにある変数に対し参照が行われた場合、1 回の参照につき 3 回のメッセージ通信が必要であり効率が悪い。

2. オリジナル KLIC ではある変数 X が構造データに具体化されていた場合には、answer_value メッセージや unify メッセージは構造データの最外側の値しか返信されない。すなわち複雑な構造データを参照する場合にはネスト毎にメッセージ通信が必要になり、メッセージ数が増大する (図 13)。

以下にそれぞれの問題点の解決法について述べる。

4.2 ポーリングの高速化

受信方式をポーリングで実装した場合に、ポーリング用の関数の速度が大きな問題になってくる。しかし第 3章で述べた KLIC の実装に用いたポーリング用の関数 `cprobe()`

は、システムコールを伴うことによるオーバヘッドのために非常に低速である。これでは cprobe() の実行時間が全体の処理時間の多くの部分を占めてしまうことになりかねない。そこでポーリングのオーバーヘッドを減らした専用マクロ Cprobe() を作成した。

第 3節で述べたように AP1000 では、RTC のステータスレジスタの値を読み出すことによって、メッセージの到着の確認ができる。この方法を用いることで Cprobe() ではレジスタに対する読み出し 2 回と書き込み 1 回でポーリングを行うことができ、cprobe() に比べて大幅なポーリング時間の短縮が期待できる。

4.3 一括送信による通信頻度の低減

4.3.1 手法の概要

本手法は外部参照ポインタの参照によって生じる answer_value メッセージおよび、具体化によって生じる unify メッセージが構造データである場合に、相手プロセスの必要とするデータ全てを一括して送信する。これにより不要なメッセージを送信することなしにメッセージ数を減少させることが可能になる。従来の方法では、構造データ f(g(h(a))) のうち f(g(X)) を N_cが必要だとすると、6 回の物理通信が必要であった (図 13)。そこで本手法を用いると、必要な物理通信は 2 回ですみ高速化が実現できる (後述する理由により release メッセージも不要)(図 14)。

以下では、ノード N_pにおいて $\mathrm{goal}(X_1, \cdots, X_n)@\mathrm{node}(N_c)$ というボディゴールが実行され、ノード N_cにこのゴールが送信された場合を前提に、一括送信に関して詳述する。

4.3.2 送信モード

引数 $X_i(i = 1, \cdots, n)$ がノード N_p上で具体化される場合は N_p側が、N_c上で具体化される場合は N_c側が、それぞれ一括送信を行う。また、Xの具体値がアトミックデータである場合は一括送信を行う意味がないので、余計なオーバヘッドを避けるため、従来と同じ送信を行う。そこで、ゴールの各引数に対し送信モードを以下の様に定義する。

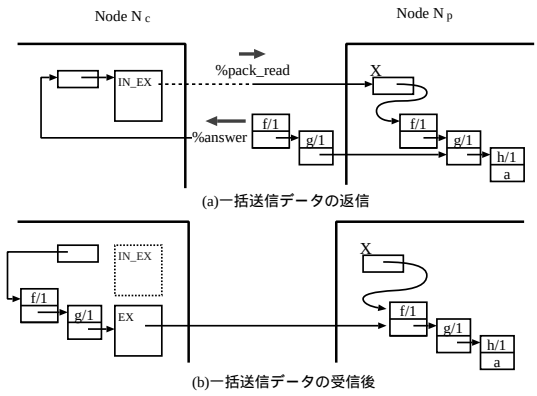

図 14: 構造データの送信 (一括送信あり)

in: 引数が送信側で具体化され、その具体値が構造データである場合。

out: 引数が受信側で具体化され、その具体値が構造データである場合。

normal: 具体化の行われるプロセスが判明しない場合、または具体値がアトミックデータである場合。

これらのモードは KL1 のソースプログラムを静的に解析することにより判明する。また静的解析によって、構造データのとり得る要素の型を調べ、実行時に生成されたデータ型に応じて参照される部分を格納する “一括送信コード” が生成される [10]。なお、ゴール goal(X) が X として f(g(Y)) を必要としている場合の一括送信コードは図 15のようになる。

4.3.3 一括送信処理

goal/n のある引数 X について、送信モードにより以下の処理を行う。

```
1 packsend_goal_1_1_top(V,H)
  :- packsend_goal_2_1_count_init(V,S),
 packsend_goal_2_1_pack_init(V,S,H).
2 packsend_goal_2_1_count_init(V,S1)
  :- S0 := 0,
 packsend_goal_2_1_count(V,S0,S1).
3 packsend_goal_2_1_pack_init(V,S,H)
  :- wait(S)
  |  generic:new(pack_buf,B,S,1,H),
 packsend_stack_2_1(B,V),
     packsend_end(B,V).
4 packsend_goal_2_1_count(V,S0,S2)
  :- inline:"guard_unbound(%0,%f)":[V+any]
  | S1 := S0+5.
  otherwise.
5 packsend_goal_2_1_count(V,S0,S2)
  :- V = f(V0)
  | S1 := S0 + 2,
packsend_goal_2_1_count(V0,S1,S2).
6 packsend_goal_2_1_count(V,S0,S2)
  :- V = g(V0)
  | S1 := S0 + 2, S2 := S1 + 4.
7 packsend_goal_2_1(B,V)
  :- inline:"guard_unbound(%0,%f)":[V+any]
  | generic:put_in_ref(B,V,0).
  otherwise.
8 packsend_goal_2_1(B,V)
  :- V = f(V0)
  |  generic:put_functor(B,V),
 packsend_goal_2_1(B,V0).
9 packsend_goal_2_1(B,V)
:- V = g(V0)
  |  generic:put_functor(B,V),
    generic:put_unused(B,V0).
10 packsend_end(B,V) :- generic:send(B).
```

各節の前の番号は説明のために付加。

図 15: 一括送信コードの例

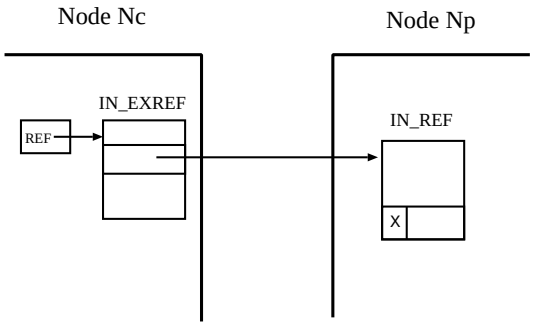

図 16: in モード外部参照ポインタの生成

(a) 送信モードが in の場合

- in モード外部参照ポインタの生成
  X は N_p の側で具体化されるため、N_c からの読み出し要求メッセージの受信により、answer_value メッセージとして具体値を一括送信する。そこで、一括送信コードへのポインタを持った外部参照ポインタを作成する (図 16)。この外部参照ポインタを in モード外部参照ポインタと呼び、generator object (IN_EXREF) で実現される。IN_EXREF がフックする REF を X' とする。

- 参照値の読みだし
  in モード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説明する。

  1. N_c での X への参照によって IN_EXREF の GENERATE メソッドが起動する。GENERATE メソッドは返信先を用意し、参照するノード N_p に対して、一括送信コードを持った読みだし要求メッセージ (pack_read メッセージ) を送る。goal/n は実行を中断し、X' にフックする。これ以後、goal/n は X に対して参照も具体化も行わないため、pack_read メッセージには release メッセージの機能も持たせてやる。こうすることで release メッセージが不要になり、さ

らなるメッセージ数の削減が実現できる。

2. read_all メッセージをノード N_p が受信すると、X が未具体化変数であれは X に consumer object(PACK_REPLY_HOOK) をフックし、値の返信は中断される (図 17(b))。PACK_REPLY_HOOK は内部データとして返信先の位置および一括送信ゴールへのポインタを持つ。その後 X が具体化されると PACK_REPLY_HOOK の UNIFY メソッドが起動し、次の処理を行う。また read_all メッセージ受信時に X が具体化済の場合も、受信処理として次の処理を行う。

3. 返信処理を行う consumer object ANSWER_HOOK を作り、pack_read メッセージに含まれていた一括送信ゴールを実行可能ゴールキューの先頭につなぐ (図 17(c))。ANSWER_HOOK は内部データとして返信先の位置などを持つ。

   一括送信ゴールは X および ANSWER_HOOK を引数として持ち、次の処理を行う。以下では図 15のプログラムリストを用いてその動作を説明する。

   X の具体値のサイズを数え (2,4,5,6 行)、一括送信用バッファである data object PACK_BUF を生成する (3 行)。続いて X の具体値をたどりながら、PACK_BUF のメソッドにより具体値の各要素を PACK_BUF に格納していく (7,8,9 行),(図 17(d))。最後に PACK_BUF と ANSWER_HOOK を同一化して終了する (10 行)。

   この最後の同一化により、ANSWER_HOOK の UNIFY メソッドが起動し、PACK_BUF を answer_value メッセージとして、N_c に送信する (図 17(e))。

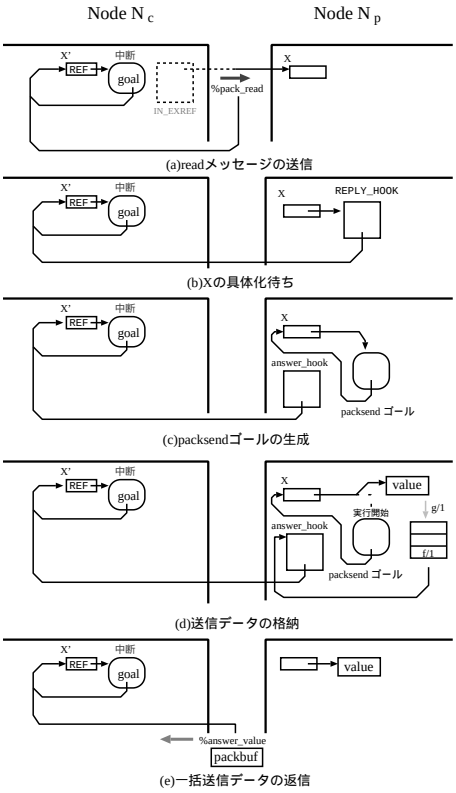

図 17: 送信モード in における参照値の読みだし

(b) 送信モードが out の場合

- out モード外部参照ポインタの生成
  X は N_cの側で具体化されるため、N_cは unify メッセージとして具体値を N_pに一括送信をする必要がある。そこで、N_c側がゴール goal/n を受信した時、EXREF の代わりに一括送信を行う generator object OUT_EXREF を用いて外部参照ポインタを実現する (図 18)。この外部参照ポインタを out モード外部参照ポインタと呼ぶ。OUT_EXREF は内部データとして参照先および X に対応する一括送信コードへのポインタを持つ。N_p側では、従来と同様の処理を行う。

- N_cにおける X の具体化
  N_cにおいて X の具体化が起きると、

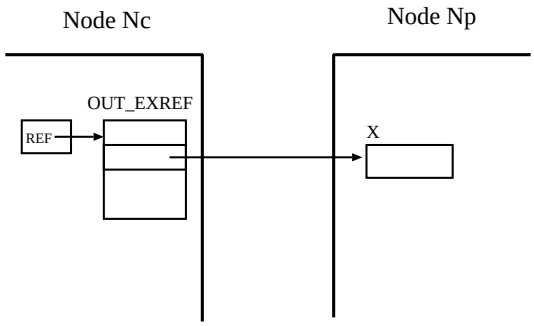

図 18: out モード外部参照ポインタの生成

OUT_EXREF の UNIFY メソッドが起動する。このメソッドでは返信処理を行う consumer object UNIFY_HOOK を作り、一括送信ゴールを実行可能ゴールキューの先頭につなぐ (図 19(a))。UNIFY_HOOK は内部データとして参照先の位置などを持つ。

一括送信ゴールは X の具体値および UNIFY_HOOK を引数として持ち、送信モードが in の場合と同様に PACK_BUF を作成して、送信データを格納し、UNIFY_HOOK と PACK_BUF の同一化を行う (図 19(b))。

この同一化により、UNIFY_HOOK の UNIFY メソッドが起動し、PACK_BUF を unify メッセージとして、N_cに送信する (図 19(c))。

(c) 送信モードが normal の場合
従来と同様に 2.2節で述べた外部参照ポインタの処理を行う。

4.3.4 未具体化変数を含む構造データの送信処理

一括送信ゴールが X の具体値を PACK_BUF に格納する際に、具体値が未具体化変数 Y を含む構造データの場合がある。この場合にもノード間に外部参照ポインタが生じる。このとき、Y の具体化がいずれの側で行われるかが判明していれば、Y の具体値につ

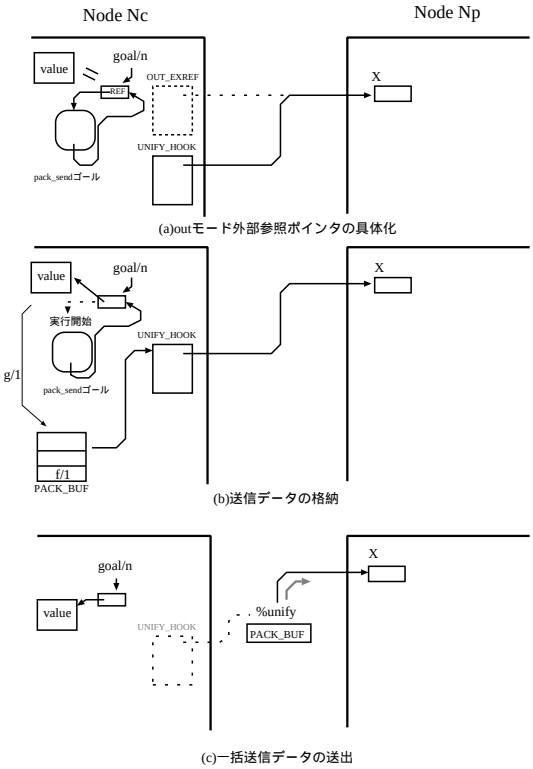

図 19: 送信モード out の変数の具体化

いても一括送信を行うことが望ましい。そこで、次のような処理を行う。

X の具体化と Y の具体化が同じノードで行われる場合は送信モードが in である引数の送信と同様の処理を行う。逆に具体化が行われるノードが異なる場合は、送信モードが out である引数の送信と同様の処理を行う。また Y がいずれのノードで具体化されるか判明しないときは一括送信は不可能であり、Y の具体値がアトミックデータであるときは、一括送信を行う意味がない。よって、従来と同様の処理を行う。

4.3.5 一括送信用バッファ

本研究では一括送信ゴールで送信データを格納するバッファは data object PACK_BUF を用いて実現した。そのためバッファに対するデータの格納やバッファの送信処理は PACK_BUF に対するメソッド呼び出しによって行われる。そこで本節では一括送信用バッ

ファPACK_BUF およびその各メソッドについて説明する。

一括送信ゴールはデータ格納中に中断する可能性があるため、送信用バッファは各一括送信ゴール毎に動的な確保が行われる。

次に PACK_BUF の各メソッドについて説明する。

- データ格納用メソッド
  PACK_BUF には次のデータ格納用メソッドを用意した。格納データを V とする。

  put_atomic: アトミックデータデコード関数へのポインタおよび V をバッファに格納する。

  put_functor: ファンクタ型データデコード関数へのポインタおよび V のファンクタ名を格納する

  put_cons: コンス型データデコード関数へのポインタを格納する。

  put_unused: V がアトミックデータの場合は put_atomic と同様の操作が行われる。その他の場合は EXREF 生成関数へのポインタ、および V の位置を格納する。

  put_in_ref: V が未具体化変数で送信側プロセスで具体化される場合に呼ばれる。IN_EXREF 生成関数へのポインタおよび V の位置を格納する。

  put_out_exref: V が未具体化変数で受信側プロセスで具体化される場合に呼ばれる。OUT_EXREF 生成関数へのポインタおよび V の位置を格納する。

- バッファ送信用メソッド

  send_buf: ANSWER_HOOK または UNIFY_HOOK と自分との同一化を行う。これによってメッセージの送信が行われる。

5 性能評価

本章では 4節で述べた通信の高速化手法の効果を AP1000 上での評価プログラムの実行により検証する。

5.1 評価プログラム

今回の性能評価では、次の 4 種類のプログラムを使用した。

loop(図 **20**) 定められた回数だけ、再帰呼び出しを行うプログラムである。

stack(図 **21**) ノード間でスタック操作を行うプログラムである。実行によりスタックゴール stack と操作ゴール drive が生成され、drive はリスト要素として操作命令 push(N) および pop(N) が交互に現れるようなストリームを、stack に送信する。stack はスタックとして動作し、push(N) に対しては N の値を構造データの形で順に格納し、pop(N) に対しては N をスタックトップの値で具体化する。

mastermind(図 **22**) マスターマインドとは $1 \sim N$ までの任意の数字 L 個の並びを推測するゲームである。ある試行での推測解が正解でなかった場合は、推測解と正解で数字および位置が一致したものの個数、数字は合っているが位置が異なっているものの個数、のそれぞれがヒントとして与えられ、次の試行での推測解を決定する際の情報として利用する。

本プログラムでは、このゲームにおいて M 回以内の試行で正解を当てるような推測の並びを全解探索する。探索の途中で分岐が生じた場合は、ノード番号が隣のノードから順に仕事を割り当てていく。

nqueen(図 **23**) $N-queen$ 問題は、チェスの駒であるクイーンを、互いに相手を取る

```
main:-p(10000000).
p(X):-X>0|New_X := X-1,p(New_X).
p(0).
```

図 20: プログラム loop

```
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 21: プログラム stack

ことのできない位置を占めるよう、$N \times N$ の盤面上に N 個配置するという問題である。

本プログラムでは、この条件を満たす配置を全解探索し、見つかった解の個数を出力する。上記の条件を満たすには、盤面上の各列には必ずクイーンが 1 個ずつ配置されなければならないという性質がある。これを利用し、盤面の状態を "i 列目のクイーンが存在する行" を表す整数 $(1, \ldots, N)$ を N 個並べたリストで表現する。1 列目のクイーンが存在する行により問題を N 個に分割し、各々を各ノードに割り当てて並列探索を行う。

5.2 専用ポーリングマクロ Cprobe() の評価

loop および nqueen を用いて、ポーリング用の標準関数 cprobe() と専用マクロ Cprobe() の比較を行う。

loop　繰り返し回数を 10000000 回とした。セル 1 台だけを用いてポーリングの純粋なオーバヘッドを測定した。表 2にそれぞれの実行時間と loop 1 リダクション当たりに占めるポーリングの割合を示している。Cprobe() は cprobe() に比べ約 12 倍の速度で実行できた。しかし 1 リダクション当たりに占める割合は Cprobe() でも 42%もあり、かなり大きなオーバヘッドになっている。これは、loop の 1 リダクション当たりの演算量が極端に少ないからで、一般的には、リダクション時間はもっと長く、また受信したメッセージの処理なども行われるため全実行時間に占める割合はもっと小さくなると考えられる。

表 3: nqueen 実行結果 (ポーリングの評価)

ポーリング手段	実行時間 (秒)
cprobe()	47.3
Cprobe()	24.8

nqueen　$N = 11$ としてセル台数 2 台で実行し、実際に通信が行われるプログラムでの実行時間の評価を行った。結果を表 3に示す。cprobe() を用いた場合に比べ、1.9 倍の速度向上が得られた。

5.3 一括送信の評価

loop を除く上記の 3 プログラムについて、AP1000 用 KLIC ランタイム上、および一括送信コードを用いた拡張版ランタイム上のそれぞれで実行し、性能評価を行った。計測項目は各セル台数での実行時間、および物理通信回数である。

なお、各プログラムは 64 セル構成の AP1000 上で、-LSEND -R 0 オプションを指定して実行した。ポーリングの手段としてはいずれも Cprobe() を用いた。

stack　ノード数 1 および 2 について、push&pop 操作を 10000 回繰り返す実行時間を測定した。その結果を表 4に示す。

セル 2 台で一括送信ありの場合、一括送信なしの場合に比べ物理通信回数が約 1/4 に減少し、2.4 倍の速度向上が得られた。これは stack/2 第一引数がファンクタを要素としたリストであり、一括送信なしでは 4 回かけて送信されるのに対し、一括送信ありでは最良の場合、一回で送信されるためである。

また、本プログラムは演算量が少なく、通信時間が実行時間を支配してしまうので、上記の通信回数減少がそのまま速度向上に反映されている。

mastermind　解く問題は $(L, M, N) = (5, 3, 2)$ とした。

本プログラムでも負荷分散単位である推測解の個数を超えるノード数で実行すると、負荷にばらつきが生じてしまう。今回の問題では推測解として考えられる組合せが高々32 通りなので、セル 32 台までについて評価を行った。結果を表 5および図 24に示す。

本プログラムでは分散される負荷の単位が小さく、通信回数が多いため stack と同様に通信時間が実行時間を支配してしまう。そのため台数効果はあまり得られていない。一括送信の効果は、物理通信回数

```
main :- solve(3,2,A), display(A).
solve(NG,NC,Out)
 :- code(C), length(C,0,CL), gen(NC,Clist), genGuess(Guesses,Clist,CL),
    spawn_cl0(Guesses,Guesses,CL,[],0,NG,Out,[]).
code(C) :- C=[1,2,1,2,1].
genGuess(Out,L,CL) :- N:=CL-1, addmyself([[]],L,In,[]), genGuess(N,In,L,Out).
genGuess(N,In,L,Out) :- N>0 | N1:=N-1, addmyself(In,L,Temp,[]),
                              genGuess(N1,Temp,L,Out).
genGuess(0,In,_,Out) :- Out=In.
addmyself([I|Is],Myself,O1,O3) :- combo(Myself,I,O1,O2),
                                  addmyself(Is,Myself,O2,O3).
addmyself([]    ,_     ,O1,O2) :- O1=O2.
combo([C|Cs],L,O1,O3) :- O1=[[C|L]|O2], combo(Cs,L,O2,O3).
combo([]    ,_,O1,O2) :- O1=O2.
spawn_cl0(Guesses,Plist,CL,A,N,NG,S0,S2)
 :- nextnode(Node), spawn_cl1(Guesses,Plist,CL,A,N,NG,S0,S2,Node).
spawn_cl1(Guesses,Plist,CL,A,N,NG,S0,S2,Node)
 :- wait(Node) | spawn(Guesses,Plist,CL,A,N,NG,S0,S2)@node(Node).
nextnode(Node) :- builtin:current_node(CC,TC), CL:=(CC+1) mod TC.
spawn([]    ,_    ,_ ,_,_,_ ,S0,S1) :- S0=S1.
spawn([G|Gs],Plist,CL,A,N,NG,S0,S2) :- test(A,G,Plist,CL,A,N,NG,S0,S1),
                                        spawn_cl0(Gs,Plist,CL,A,N,NG,S1,S2).
test([p(OldBulls,OldCows,Tried)|Gs],Guess,Plist,CL,A,N,NG,S0,S1)
 :- eval(Guess,Tried,Bulls,Cows),
    check(OldCows,Cows,OldBulls,Bulls,Gs,Guess,CL,Plist,A,N,NG,S0,S1).
test([]                            ,Guess,Plist,CL,A,N,NG,S0,S1)
 :- code(Code),eval(Guess,Code,Bulls,Cows),
    loop(Bulls,Cows,Guess,CL,Plist,A,N,NG,S0,S1).
eval(X,Y,Bulls,Cows) :- exact(X,Y,Bulls,0,X1,Y1), inexact(X1,Y1,Cows,0).
exact([]    ,T2    ,N,I,S1,S2) :- N=I, S2=T2, S1=[].
exact([H|T] ,[H|T2],N,I,S1,S2) :- I1:=I+1, exact(T,T2,N,I1,S1,S2).
otherwise.
exact([X|T1],[Y|T2],N,I,X1,Y1) :- X1=[X|S1], Y1=[Y|S2], exact(T1,T2,N,I,S1,S2).

inexact(_      ,[]    ,N,I) :- I=N.
inexact([]     ,_     ,N,I) :- I=N.
otherwise.
inexact([Ht|Tt],Choice,N,I) :- delete(Choice,Ht,F,F,Choice,Tt,N,I).
delete([]    ,_,_,_        ,Choice,Tt,N,I) :- inexact(Tt,Choice,N,I).
delete([X|Y] ,X,F,NewChoice,_     ,Tt,N,I) :- I1:=I+1
                                              | F=Y, inexact(Tt,NewChoice,N,I1).
otherwise.
delete([X1|Y],X,F,NewChoice,Choice,Tt,N,I)
 :- F=[X1|Z], delete(Y,X,Z,NewChoice,Choice,Tt,N,I).
check(Sum,Sum,Bulls,Bulls,Gs,Guess,CL,Plist,A,N,NG,S0,S1)
 :- test(Gs,Guess,Plist,CL,A,N,NG,S0,S1).
otherwise.
check(_  ,_  ,_    ,_    ,_ ,_    ,_ ,_    ,_,_,_ ,S0,S1) :- S0=S1.
loop(Bulls,Cows,Guess,CL,Plist,A,N,NG,S0,S1)
 :- Bulls<CL, N1:=N+1, N1<NG
  | spawn_cl0(Plist,Plist,CL,[p(Bulls,Cows,Guess)|A],N1,NG,S0,S1).
loop(CL   ,_   ,_    ,CL,_    ,A,_,_ ,S0,S1) :- S0=[A|S1].
otherwise.
loop(_    ,_   ,_    ,_ ,_    ,_,_,_ ,S0,S1) :- S0=S1.
display(A) :- count(A,N),io:outstream([print(N),nl]).
gen(N,X) :- N>0   | M:=N-1, X=[N|Xs], gen(M,Xs).
gen(N,X) :- N=:=0 | X=[].
length([_|T],X,Y) :- X1:=X+1, length(T,X1,Y).
length([]   ,X,Y) :- X=Y.
count(L,N) :- count(L,0,N).
count([]    ,M,N) :- M=N.
count([X|Xs],M,N) :- M1:=M+1, count(Xs,M1,N).
```

図 22: プログラム mastermind

```
main :- generate(11,[],L), nqueenD(L,[],X,[]).
generate(N,T,L) :- N>0  | T1=[N|T], N1:=N-1, generate(N1,T1,L).
generate(N,T,L) :- N=<0 | L=T.
nqueenD([N|T],R,I,O) :- worknode(N,M), nqueenD(M,N,T,R,I,O).
nqueenD([]   ,R,I,O) :- I=O.
worknode(N,M) :- builtin:current_node(_,TC), M:=N mod TC.
nqueenD(M,N,T,R,I,O) :- M>0   | append(T,R,NN), nqueenD(T,[N|R],I,X),
                              c1(N,1,NN,[],[],X,O)@node(M).
nqueenD(M,N,T,R,I,O) :- M=:=0 | append(T,R,NN), nqueenD(T,[N|R],I,X),
                              c1(N,1,NN,[],[],X,O).
c1(T,D,N,[P|R],B,I,O):- T=\=P+D, T=\=P-D | D1:=D+1, c1(T,D1,N,R,B,I,O).
c1(T,D,N,[P|_],B,I,O):- T=:=P+D | I=O.
c1(T,D,N,[P|_],B,I,O):- T=:=P-D | I=O.
c1(T,D,N,[]   ,B,I,O):- queen(N,[],[T|B],I,O).
queen([P|U],C    ,L,I,O) :- append(U,C,N), c1(P,1,N,L,L,I,X),
                            queen(U,[P|C],L,X,O).
queen([]   ,[_|_],_,I,O) :- I=O.
queen([]   ,[]   ,L,I,O) :- I=[L|O].
append([A|X],Y,Z) :- Z=[A|Z1], append(X,Y,Z1).
append([]   ,Y,Z) :- Z=Y.
```

図 23: プログラム nqueen

表 2: ポーリングのオーバヘッド

ポーリング手段	実行時間 (μsec)	loop での 1 リダクションに占める割合 (%)
cprobe()	8.33	90.0
Cprobe()	0.64	42.0

表 4: stack 実行結果

	一括送信なし		一括送信あり	
セル台数	実行時間 (秒)	通信回数	実行時間 (秒)	通信回数
1	1.7	0	—	—
2	24.1	130004	10.3	30003

表 5: mastermind 実行結果

	一括送信なし		一括送信あり	
セル台数	実行時間 (秒)	通信回数	実行時間 (秒)	通信回数
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

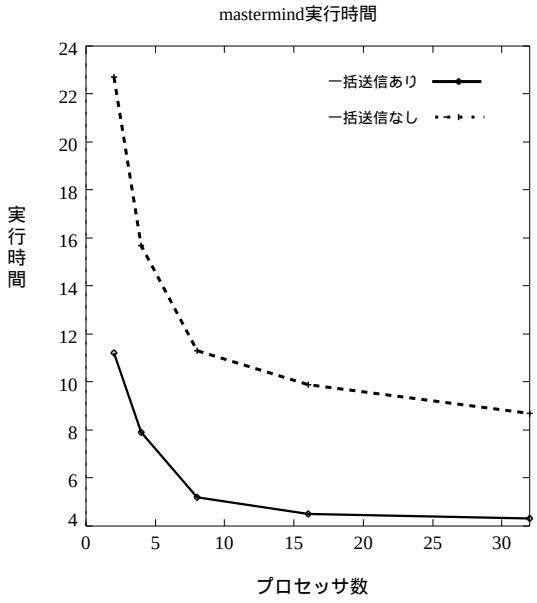

図 24: 評価結果 (mastermind)

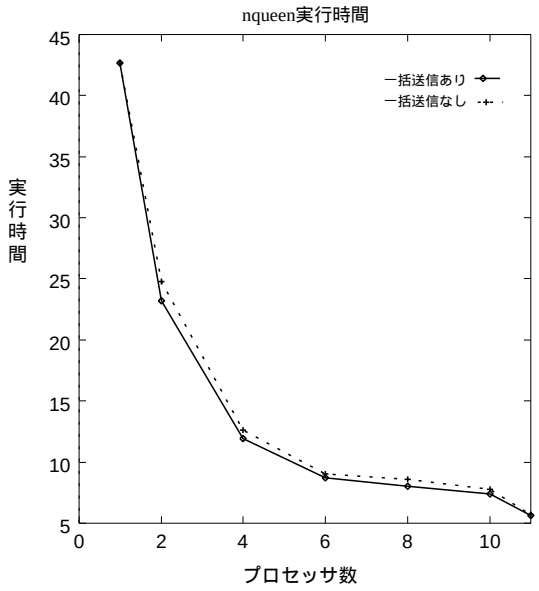

図 25: 評価結果 (nqueen)

については $1/4 \sim 1/6$ に減少し、実行時間については全セル台数で約 $1/2$ に減少した。

nqueen　$N = 11$ として、実行時間を測定した。

本プログラムではノード数が、負荷分散の単位である列の数を超えると、負荷にばらつきを生じる。そこで 11 セルまでについて評価を行った。結果を表 6 および図 25に示す。

セル 11 台の場合、一括送信なしで 9.1 倍、一括送信ありで 9.7 倍と良好な台数効果が得られている。ただし、一括送信の効果に関しては、通信回数が $1/10 \sim 1/40$ に減少したにもかかわらず実行時間は $5 \sim 9\%$程度しか短縮されていない。これは最初と最後以外には通信が行われず、全体として通信回数が少ないため、stack の場合とは異なり演算時間が実行時間を支配してしまうためである。

6　おわりに

本論では、並列論理型言語 KL1 処理系 KLIC の AP1000 への移植と、AP1000 用 KLIC 上への一括送信による通信の高速化手法について述べ、その効果を AP1000 上でプログラムを実行し評価した。受信判定をポーリングで行う場合のオーバヘッドは cprobe で $8.3\mu sec$、Cprobe で $0.64\mu sec$ であった。受信判定として Cprobe を用いることでポーリングのオーバヘッドを減少させることができた。

一括送信ゴールを用いたデータの一括送信については、5章で述べたように、通信時間が実行時間を支配するような場合では stack で 2.5 倍、mastermind で最大で 2.2 倍の速度向上が得られた。また nqueen のような演算時間が実行時間を支配するような例でも一応の速度向上が得られた。以上の結果から当初の目的はほぼ達成されたと言える。

しかし、loop における 1 リダクション当たりのオーバヘッドが 42%を占めた結果から、通信回数が少なく 1 リダクション当たりの実行時間が短いプログラムの場合、ポーリングのオーバヘッドがかなり大きくなると考えられる。これについては今後の課題として割り込み処理を用いた受信方式の実装を検討中である。また通信のさらなる高速化として、単方向通信や遅延通信 [10] [11] などのランタイムの拡張を行い、さらにメッセージ数の削減を行うことを考えている。

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。適切な御指導を賜わった中島浩助教授、および数々の有用な御意見を戴いた森眞一郎助手に心から感謝致します。

また、各種資料を提供して戴いた (財) 新世代コンピュータ開発機構、および並列計算機 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。さらに、共同研究者である大野和彦氏をはじめとして、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いたします。

最後に、筆者と労苦を共にした友人諸氏、及び日頃世話をかけている両親に御礼致します。

表 6: nqueen 実行結果 (一括送信の評価)

	一括送信なし		一括送信あり	
セル台数	実行時間 (秒)	通信回数	実行時間 (秒)	通信回数
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

参考文献

[1] Takashi Chikayama, “Introduction to KL1”, August. 1994.

[2] 木村 康則, “並列論理型言語 KL1 の処理系に関する研究:1-24”

[3] Takashi Chikayama, “KLIC User’s Manual”, October. 1994,

[4] 六沢 一昭, “KLIC 分散処理系プログラミングガイド”, October. 1994.

[5] Al Geist,Adam Beguelin,Jack Dongarra,Weicheng Jiang,Robert Manchek, Vaidy Sunderam, “PVM3 User’s guide & reference manual”, August. 1994.

[6] “AP1000 プログラム開発手引書 C 言語インタフェース第三版” July. 1993.

[7] 富士通株式会社, “AP1000 ライブラリマニュアル [セル] C 言語インタフェース第三版”, July. 1993.

[8] 富士通株式会社, “AP1000 ライブラリマニュアル [ホスト] C 言語インタフェース第三版”, July. 1993.

[9] 富士通アーキテクチャ研究部第 2 研究室, “AP1000 セルプロセッサ論理仕様書 第 04 版”, July. 1992.

[10] 大野 和彦, “静的解析による並列論理型言語 KL1 の通信最適化手法”, February. 1995.

[11] 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”, 財団法人新世代コンピュータ技術開発機構, March. 1994.

[12] 大野 和彦,伊川 雅彦,森 慎一郎,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 94-OS-67:183-190, December. 1994.

[13] 田沼 仁, “並列論理型言語 KL1 の AP1000 への実装”, March. 1994.

[14] 仲瀬 明彦、藤瀬 哲朗、近山 隆, “ポータブル KL1 処理系 KLIC の概要”, 第 47 回全国大会 講演論文集:5-55 ~ 5-56, October. 1993.

[15] K. Nakajima and Y. Inamura and N. Ichiyoshi and K. Rokusawa and T. Chikayama, “Distributed Implementation of KL1 on the Multi-PSI/V2”, “Proc. 6th Intl. Conf. and Symp. on Logic Programming:436-451”, 1989.

[16] Takashi Chikayama and Tetsuro Fujise and Daigo Sekita, “A Portable and Efficient Implementation of KL1”, “Proc. 6th Intl. Symp. PLILP’94 :25-39”, 1994

[17] K.Ueda, “Guarded Horn Clauses:A Parallel Logic Programming Language with the Concept of a Guard”, ICOT, 1986.