分散言語処理系DKLICの設計と実装

Design and Implementation of
Distributed Language System DKLIC

松村 量[†] 高山 啓[†] 高木 祐介[†*] 加藤 紀夫[†] 上田 和紀[†]

Ryo MATSUMURA Hiraku TAKAYAMA Yusuke TAKAGI Norio KATO Kazunori UEDA

[†] 早稲田大学理工学部情報学科

Dept. of Information and Computer Science, Waseda University

lang@ueda.info.waseda.ac.jp

並行論理型言語は、暗黙の並列性および自動同期の機構を持ち、静的解析によって通信プロトコルの一貫性が保証できるなど特徴を持つため、分散アプリケーションの記述に適している。

私達は、並行論理型言語KL1の処理系KLICに分散拡張のためのAPIを提供することによって、宣言型広域分散プログラミング環境を実現することを目指すDKLICプロジェクトを進めている。現在までに、未定義変数同士の単一化が検出できるようにKLICを拡張し、分散論理変数の無駄な中継ノードを自動的に排除して最適な通信路を形成する機構を実装した。

1 はじめに

分散プログラムは逐次プログラムより難しい。逐次プログラムはアプリケーションロジックの記述だけで完成させることが可能だが、分散プログラムはアプリケーションロジックと通信処理の記述が必要だからである。通信機能が言語に隠蔽されていることで、アプリケーションロジックの記述に専念できるならば、分散プログラミングの困難さは逐次プログラムに近付くと考えられる。

並行論理型言語KL1は、並行プロセスと自動同期、非同期プロセス間通信をプリミティブとしてサポートする。遠隔プロセスは並行プロセスの一種であると考えられる。並行論理型言語を分散拡張することで、分散プロセスの協調処理の記述に適した処理系を実現することが可能であると考えられる。

KL1の言語処理系の一つにKLIC[1][2][3]がある。KLICはKL1をC言語に変換することで高い実行効率と可搬性を実現した処理系である。しかし、KLICの提供する分散機能はUNIXレベルのソケットのみである。そこで、私達はKLIC上に分散拡張のためのAPIを提供することで、ネットワーク透過性に優れ、高い記述性をもった分散言語処理系を実現することを目標に研究を進めてきた。この言語処理系をDKLICと呼ぶ。

本論文では、DKLICの構成要素と中継ノードの排除について述べる。

*現在 (株) ジャステック

2 DKLICのプログラム例

DKLICによるサンプルプログラムとして簡単なチャットプログラムを提示する。このチャットは、チャットルームを一つだけ持ち、複数のユーザが接続することができる。

```
% サーバ側
server :-
   % サービス管理API（登録用）
   register(chat,Res),
   chat(Res,Data,Data). % チャット本体

chat(accept(Chan),Msgs,Data) :-
   % Msgs1とMsgs2を併合してMsgsが出力される
   merge(Msgs1,Msgs2,Msgs),
   stub(Chan,Msgs1,Data),
   register(chat,Res),
   chat(Res,Msgs2,Data).

% クライアントごとの入出力を処理する述語
stub(join(Name,In,Out),Msgs,Data) :-
   % Msgs : Nameによる発言のストリーム
   % Data : 各クライアントの発言をまとめたもの
   Out=Data, label(Name,In,Msgs).

label(Name,[Text|In],Msgs) :-
   % 発言者名と発言内容で構造体を構成
   Msgs = [msg(Name,Text)|Msgs1],
   label(Name,In,Msgs1).

% クライアント側
client :- username(Name), % 発言者名を取得
```

```
    lineinput(In), % 標準入力から入力
    lazyprint(Out), % 標準出力へ出力
    lookup(chat,Res), % サービス管理 API（検索用）
    proxy(Res,Name,In,Out).

proxy(normal(Chan),Name,In,Out) :-
    % チャットを利用するための構造体を生成
    Chan = join(Name,In,Out).
```

register/2 の第一引数は登録するサービス名である。第二引数はクライアントと接続すると accept 構造体に具体化する。accept 構造体の要素はクライアントサーバ間のチャネルである。chat/2 内で register/2 と chat/2 を呼んでいるのは、マルチクライアント対応用のために再び chat をサービス管理プロセスに登録し、クライアントとの接続を取得するためである。

lookup/2 の第一引数は検索するサービス名である。第二引数はサーバとの接続に成功すると normal 構造体に具体化する。normal 構造体の要素はクライアント サーバ間のチャネルである。

3 DKLIC が前提とする環境

広域分散プログラミング環境とは、ノードが動的に増減し、ノードのアーキテクチャや OS が不均一であるような環境である。ノードが増減するということは、ノードがいつまで存在するかを知ることができず、事前に特定のノードの位置を知ることや全てのノードに全順序を付けることが困難であることを意味する。ノードのアーキテクチャや OS が不均一であるということは、ノードによって実行効率が異なったり、サービスの実行に別ノードの機械語コードを使用することはできないということを意味する。

DKLIC はこれらの条件を前提に設計を行った。

4 DKLIC の構成

DKLIC は、図 1 のような構成をしている。

DKLIC はサービス管理層と分散論理変数層の 2 層で構成される。

サービス管理層は、遠隔サービスの仲介を担当するサービス管理プロセスと、ローカルのサービス管理プロセスに接続するためのサービス管理 API の 2 つで構成される。

サービス管理より下位な層として、分散論理変数層が存在する。ノードをまたいだ変数の輸出入および変数の具体化などを管理する層である。分散論理変数層は、サービス管理プロセスによって隠蔽されている。このことによって、ユーザは分散論理変数を直接利用することなく外部サービスとの通信を行うことができる。

DKLIC のノードには 2 つの種類が存在する。サービスを提供するサーバノードと、サービスを利用するクライアントノードである。サーバノードは全ての処理を自前で用意する必要はなく、一部機能を他のサーバノードに委託することができる。つまり、サーバノードも外部サービスを利用するときにはクライアントノードとして動作する。

5 サービス管理

第 3 章で述べたような環境では、プログラム中にサーバの位置情報をハードコーディングするようなプログラミングスタイルは不適切である。DKLIC では、サービスの利用時に利用可能なサービスへのチャネルを取得する。この仕組みをサービス管理 (サービス検索) という。

サービス管理プロセスは各ホストに必ず一つだけ存在する。サービス提供側は、サービス管理 API を用いてサービスをローカルのサービス管理プロセスに登録し、サービス利用者側は、サービス管理 API を用いてサービスの検索を行う。

各サービス管理 API は検索要求があると、ローカルのサービス管理プロセスにアクセスし検索を行う。DKLIC を利用した分散アプリケーションは、必ずサービス管理プロセスを通じてのみ外部サービスの利用を行うことになる。ローカルのサービス管理プロセスで検索要求に適合するサービスが見つからなかったときには、マルチキャストで他のサービス検索プロセスに検索要求を転送する。

サービス提供側では、サービス管理プロセスに、提供するサービス全体のモード情報を登録する。サービス利用側では、自分が利用する範囲内のサービスのモード情報を与える。モード情報とは、データ構造と引数の流れの向きを記述したものである。サービス管理プロセスでは、提供側と利用側のモード情報のマッチングを試み、矛盾の生じなかったサービスをクライアントに紹介する。

これによって、通信プロトコルの一貫性のとれた、サービス提供側、利用側双方にとって安全なサービスの実行を可能とする。

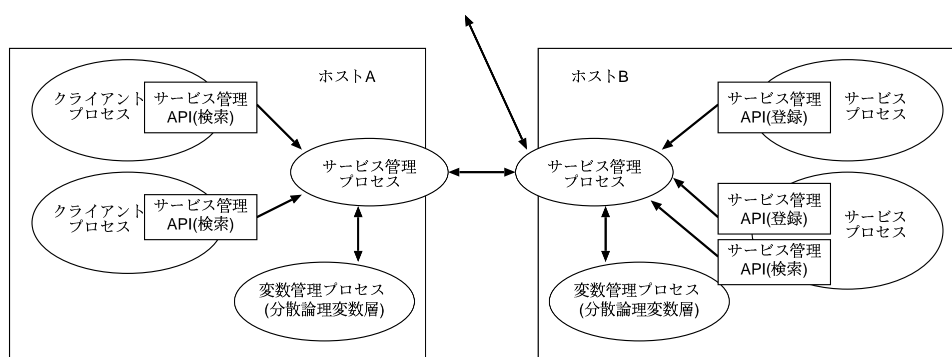

図 1: DKLIC 構成図

6 分散論理変数

KL1 の変数は未定義/具体化済の状態をもつ論理 (単一代入) 変数である。よって DKLIC では、ノードをまたいだ論理変数を管理する機能を提供する。異なる変数の区別、ノードをまたいで行われる変数の具体化、変数の送受信を管理するプロセスを変数表プロセス (以下、変数表) と呼んでいる。

変数表では、変数自体および変数に割り当てた ID と変数を生成したホスト識別子を対にした組を記録している。ID は分散論理変数を生成したホストが、管理している分散論理変数を区別するために用いる識別子として利用している。ID は、他の変数の ID と衝突が起きない範囲で任意に割り当てる。

6.1 中継排除

KL1 プログラムでは、変数を使って変数を送ったり、変数どうしを単一化によって接続したりすることで、複雑な通信ネットワークを動的に構成できる。しかし、このようなネットワークを形成すると、送信側と受信側の間には具体値を中継するだけの働きしかしないノードが介在することになる。これは、時間的、空間的効率を損なう。

このような第三者ノードによる中継の排除を行なうには、中継にのみ使われている変数を検知し、通信路を形成し直すことが必要である。変数の中継を検知するには、未定義の分散論理変数同士の単一化を検出し、その変数がそのノードの他のプロセスから参照されていないことを確認する必要がある。

現在の KLIC の実装では、未定義変数同士の単一化、およびある変数がプロセスから参照されているかどうかを、言語上からは知ることが出来なかった。そこで、処理系の単一化機構に手を加えることで未定義変数同士の具体化を検出できる機能および、GC 時に変数が自ノードのプロセスに参照されているかを検出する機能を追加した。

サービス管理プロセスでは、条件に整合するサービスが見つかると、出力チャネルの単一化を行い、クライアント-サーバ間での通信が可能になる。入出力チャネルの単一化を行うと、サービス管理プロセス内にはそのチャネルを参照するプロセスが存在しなくなる。このとき、中継排除を実行し、クライアント-サーバ間で直接通信を行う。

これによって動的にネットワークが形成される環境でも、無駄な中継が排除された最適な通信路の形成が可能になる。

変数表はホストに一つだけ存在する。従来の DKLIC の実装では、接続ごとに一つの変数表を生成していた [4]。従来の DKLIC では複数の変数表にまたがって存在していた未定義変数同士の単一化を検知する必要があったが、変数表を一つに統合することで、一つの変数表内だけを検査すればよくなり、未定義変数の検出を効率良くできるようになった。

6.2 中継排除プロトコル

中継排除を実現するプロトコルは図 2 に示す。

中継排除プロトコルの流れには、他のホストの中継排除と衝突が起きる場合と起きない場合に分類できる。

まず、衝突の起きない場合の動作を黒い枠と線で表す。

ホスト A、B、C があり、AB 間のチャネルを X 、BC 間のチャネルを Y とする。X と Y が単一化したとき、X、Y が両方未定義変数であり、かつホスト B 内に、X、Y を参照しているプロセスが存在しな

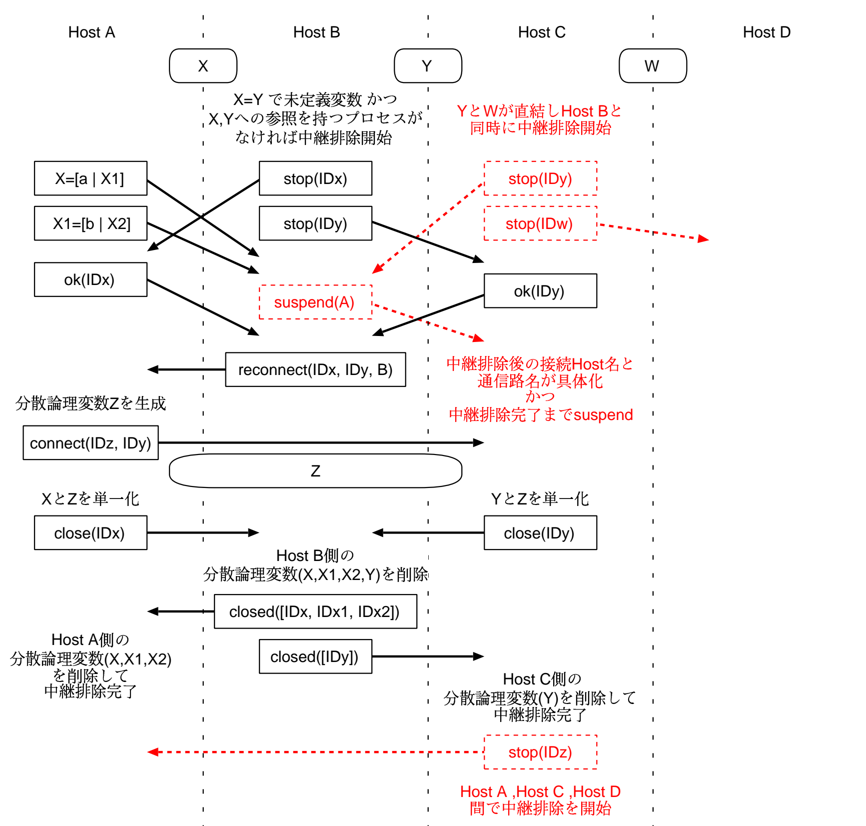

図 2: 中継排除プロトコル図

いとき、ホスト B 内の変数 X、Y は排除できる。

1. stop メッセージは、引数が指す分散論理変数およびその分散論理変数から派生する分散論理変数の具体化メッセージの送信を停止させるメッセージである。

2. stop メッセージを受信し、送信停止処理が完了すると ok メッセージをホスト B に返信する。

   ホスト B では、stop メッセージと行き違いに変数の具体化メッセージが送られてくる場合がある。ホスト B では、stop メッセージを送信後、ok メッセージを受信するまでに受信した、送信を停止するはずだった分散論理変数の ID を記憶しておく。ここで記憶した ID は、後に分散論理変数を削除する際 (5) に利用する。

3. 両方のホストから ok メッセージを受信すると、reconnect メッセージを片方のホストに送信する。

4. reconnect メッセージを受信したホストは新しい分散論理変数 Z を生成し、指定されたホストに対して新しい通信路を開くためのメッセージ connect を送信する。

5. 新しい通信路を開いたら、それぞれ X と Z、Y と Z を単一化する。単一化後は X、Y を通じて送信していたメッセージは Z を通じて送信する。ホスト A-ホスト C 間で Z が開通したら、X、Y は不必要となるので GC する。

   Z の両端では、X、Y が不必要になったことを通知するために、ホスト B に分散論理変数の削除を求めるメッセージ close を送信する。

6. close メッセージを受信したホストは、引数が示す分散論理変数から派生した分散論理変数を削除する。削除後に、削除した分散論理変数の ID を closed メッセージとして返信する。

7. closed メッセージを受信したホストでは、引数が示す分散論理変数を削除したら、中継排除は終了となる。

次に、衝突の起きる場合の、例外的な動作を赤い枠と線で表す。

X=Y によりホスト B が中継排除され、同時に Y = W によりホスト C が中継排除されるとき、ホスト B ホスト C 間を同時に stop 命令が送受信されることがある。この場合、互いに通信相手が中継排除されることで、通信路が破壊されることになる。

通信路の破壊を防ぐためには、中継排除を行うホストに優先順位をつける必要がある。すべての通信路には、つないだホストと、つながれたホストの 2 種類が存在する。この場合、通信路 Y は、ホスト B がつないだものとする。

1. 同じ通信路に対して stop メッセージが行き交った場合には、通信路をつないだ側の中継排除を優先することで通信路の破壊を防ぐ。

   もし、中継排除が進行した状態 (ok メッセージの受信以降) で stop メッセージを受け取った場合には、進行中の中継排除を優先する。

2. 中継排除を優先して行うホストは、中継排除を待つホストに対して、suspend メッセージを送信する。suspend メッセージは引数に、中継排除を待つホストが中継排除を再開するときに、自分の代わりにメッセージを送る相手ホスト名を与える。

3. suspend メッセージを受け取ったホストは、新しくメッセージを送信するホスト名と、中継排除の対象となる新しい通信路の名前が具体化し、優先ホストの中継排除が終了まで中継排除を停止する。中継排除を再開するときには、新しいホストに改めて stop メッセージを送信する。

並行論理型言語における中継排除プロトコルの研究には、本研究以外に [8] がある。[8] は、生産者が消費者に対して、完全に具体化した要素のリストを送信する one-way message プロトコルである。本研究は、要素に未定義変数を含む構造体を持つことを考慮したプロトコルになっている。

7 まとめと今後の課題

本論文では DKLIC の構成要素であるサービス管理層および分散論理変数層の紹介を行った。

従来の DKLIC[4] では、サービス管理機能は、付加的なサービスであったため、ネットワーク透過性は不十分であった。外部サービスを利用する際には、サービス管理の利用を必須とすることで、サービスの位置情報を完全に排除し、ネットワーク透過性をより高めた設計に改めた。

また、分散論理変数層による中継排除を導入することで、動的に通信路が形成される環境でも、ユーザが意識することなく自動的に最適な通信路を形成することが可能になった。

他の分散処理機構としては Java Applet に代表されるコード移送がある。DKLIC では Treecode インタプリタ [6] を用いたコード移送の研究を行っている [7]。KL1 プログラムコードから Treecode へのコンパイラを実装し、現在の DKLIC に統合することが課題である。

また、不特定多数に公開する分散アプリケーションにとってサービスの安全性の確保は重要な課題である。DKLIC では静的解析 [5] を用いることでプロトコルの一貫性を保証し、安全なサービスの提供を目標としている。KL1 の静的解析を DKLIC に応用するための記述的課題は解決されつつある。実装は今後の課題である。

謝辞

本研究の一部は、文部科学省科学研究費基盤 (C)(2)11680370 の補助を得て行った。

参考文献

[1] 関田大吾, Inside KLIC Version 1.0. *KLIC Task Group*, AITEC/JIPDEC,1998, http://www.klic.org/software/klic/inside/master.html

[2] http://www.klic.org/

[3] 瀧和男 編, 第 5 世代コンピュータの並列処理. bit 別冊, 共立出版, 1993.

[4] 高木祐介, dklic: KL1 による分散 KL1 言語処理系の実装. In 第 *4* 回プログラミングおよび応用のシステムに関するワークショップ *SPA2001*, 日本ソフトウェア科学会 2001, http://www.dcl.info.waseda.ac.jp/SPA2001/

[5] Kazunori Ueda, Resource-Passing Concurrent Programming. In *Proc. TACS 2001*, LNCS 2215, Springer, 2001, pp. 95–126.

[6] Kazunori Ueda, A Pure Meta-interpreter for Flat GHC, a Concurrent Constraint Language, In *Computational Logic: Logic Programming and Beyond*, LNAI 2407, Springer, 2002, pp. 138–161.

[7] 五十嵐宏, メタインタプリタに基づく分散並行論理型言語処理系, 早稲田大学大学院理工学研究科修士論文, 2001.

[8] Hiroshi Nakashima and Yu Inamura, An Efficient Message Transfer Mechanism Bypassing Transit Processors. In *Proc. Joint Symposium on Parallel Processing*, 情報処理学会, 1992. pp. 123–130.