dklic: KL1による分散KL1言語処理系の実装

高木祐介
早稲田大学大学院理工学研究科

上田和紀
早稲田大学理工学部情報学科

1 はじめに

分散プログラミングには逐次プログラミングにない多くの問題が存在する。プロセスの分散、セキュリティ、フォールトトレランスといった大きな問題領域から、ソケットの使い方、プロトコルに従った通信の実装といった些末な問題までである。このうち、些末な問題は適切なプログラミング言語によって解決できるものと思われる。

並行計算は並列・分散計算から物理的な属性を取り去った包括的な概念であり、逆に並行計算の上に物理的な属性を付加することによって並列・分散計算を表現できる。並行論理型言語は、並列・分散を含む並行計算をプロセス群とプロセス間通信によって宣言的に記述することができる。

並行論理型言語には Concurrent Prolog, Parlog, GHC などがあるが、KL1 [1] は最も単純な Flat GHC に実用的な拡張を施した言語である。KL1では Prolog のような探索機能が排除されており、各ゴールが並行プロセスを表現する。

dklic [10] は KL1 によって分散 API を記述し、既存の KLIC 処理系に分散プログラミングのための拡張を施す。KL1 によって KLIC を拡張するのは実装の単純さのためである。実働する分散 KL1 言語処理系を早急に実現するため、実装の単純さを重視した。また、KL1 によって記述できる API を切り分けることによって、今後実装されるべきノード実行系では KLIC 実行系の機能に加えて未定義変数の同一性の検査が行なえる必要があることが明らかになった。

dklic は場所の概念を導入するプラグマと、ミドルウェアに当たるサービス検索を提供する。プラグマは並行計算のネットワーク透過性を保つように設計してある。その一方、プラグマを考慮することによって物理的な実行状態を予測することも可能である。プラグマによってプログラムの論理的意味が変わることはないので、逐次プログラムを分散化する時にバグが導入されることは原理的にない。

以下、2, 3 節で関連研究と KL1 言語の概要を述べ、4, 5 節で dklic の設計目標と実装方針を述べる。6, 7 節で dklic が実現した遠隔述語呼出しの例とそのための分散論理変数の実装、8, 9 節で dklic の今後の目標である遠隔サーバの KLIC による例と dklic で遠隔サーバへのアクセスを実現するための実行系を説明する。最後に 10 節で本論文をまとめる。

2 関連研究

分散計算を目指した論理型言語としては、分散 KL1 の他に Distributed Oz [13,14,15] や DRL [16] などがある．本節では KL1 と Distributed Oz を簡単に紹介する。

これまでの KL1 処理系は主に並列記号処理を指向していた。並列推論マシン PIM 上の KL1 処理系 [2] や汎用並列機上の KLIC 処理系 [3] は、並列機上における KL1プロセスや論理変数、および論理型言語の基本操作である単一化を透過的に実現している。KLIC の分散メモリ版は、PVM の上で全順序付けされた複数のノードにまたがって処理を行なうが、アーキテクチャの異なるプロセッサ要素からなる分散計算環境には対応していない。広域分散機能は最低限の Unix ソケットインタフェイスが提供されている。

分散計算を意図した KL1 処理系の最初のものは KL1/J [8] である。分散 KL1 の計算モデルを Java で表現したが、分散 KL1 から Java へのコンパイラが実装されていない。

Distributed Oz は、並行論理計算の発展である並行制約計算を中心としたマルチパラダイム言語である。KL1が微粒度の並行性を暗黙に記述するのに対して、Distributed Oz は並行スレッドの生成をプログラマが

明示する。この点で Distributed Oz は、既に普及している手続き型言語の並行拡張に KL1 よりも近いと考えられる。各並行プロセスの実行順序が命令の時系列でなく、論理変数の依存関係によって表される点は KL1 と同じである。

Distributed Oz の実装は KLIC の分散メモリ版を参考にして、分散論理変数を表現している。各分散変数は、1 つのマネジャと複数のプロクシから成るアクセス構造によって表現される。分散変数を参照する各ノードにプロクシが置かれ、各プロクシはマネジャを参照する。マネジャは分散単一化や分散 GC の管理を行なう。

Distributed Oz では分散環境上での有限木の単一化を完全に実装することを目指しているのに対し、dklic では、通信路としての論理変数を効率的に実装することを目指している。論理型でない分散言語へ論理変数の考え方を導入する方法は [15] に論じられている。

3 KL1 言語とプログラミング

本節では KL1 言語とプログラミングスタイルの概要を述べる。

KL1プログラムは述語の集合から成り、**述語** (predicate) は節の集合から成る。**節** (clause) は次のような形をしている。

$$Head \text{ :- } Guard \mid Body \text{ .}$$

ここで、*Head* は原子論理式であり、*Guard* と *Body* は原子論理式のマルチ集合である。*Head* と *Guard* はその節が実行されるための条件を表し、*Body* は実行されるゴール（述語呼出し）を表す。

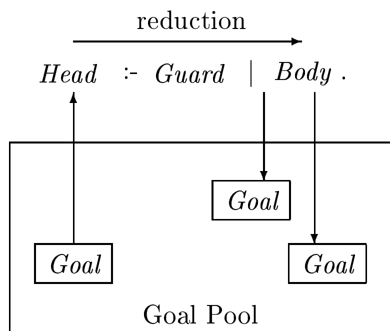

図 1: KL1 の実行モデル

ゴールが実行されて *Body* 中のサブゴール群に置換されることを、**リダクション**と言う（図 1）。ゴールが条件を満たす節が存在しない場合、リダクションの失敗が起きる。

論理変数は単一代入変数であり、**単一化** (unification) によって具体値が代入されるまでは**未定義**値を持つ。異なる具体値同士を単一化しようとすると単一化の失敗が起きる。ポインタ操作は論理変数によって隠蔽されており、変更や捏造ができないという意味で安全である。

Guard で未定義変数の具体値を参照するとゴールの実行が中断され、その変数が**具体化** (instantiation) すなわち具体値が代入されるまで待つ。KL1 プログラムの逐次性はこのデータフロー同期によってのみ導入され、各ゴールは並行に実行される。

並行に実行されるゴール、ないしデータフロー同期によって実行順序の決定された一連のゴール群は、それぞれ**プロセス**と見なすことができる。すなわち、KL1 では述語呼出しによって動的にプロセスが生成される。KLIC では全てのゴールについて（Unix プロセスより軽量な）KL1 プロセスが生成されるが、データフロー同期によって完全に実行順序が決定されたゴール群についてはプロセスを共通化する研究 [7] が行なわれている。

KL1 のデータは**項** (term) と呼ばれる有限木で表される。項は整数や記号アトムのようにアトミックな値であっても良いし、配列 (vector) やリストのようにデータ構造であっても良い。データ構造は完全に具体化されているとは限らず、未定義変数を含んでいても構わない。未定義変数を含むデータ構造を未完成データ構造と言う。

ストリームは未完成データ構造の一種で、頭から徐々に要素が具体化されていくリストである。ストリームの端点は、実行時コンフィギュレーションにおける論理変数の出現 (occurence) である。サーバプロセスや Unix のファイルストリームなど KL1 の全てのオブジェクトに対するインタフェイスはストリームとして提供されている。

KL1 ではグローバル変数が存在せず、プロセス外部と通信を行なうにはゴールの引数を共有しなければならない。このようなプロセス間通信のための変数は通信路であると見なすことができる。未完成データ構造にストリームを受け取ることによって動的に通信路を

構成することができ、モバイルプロセスを表現できる。

手続き型言語による並行処理の誤りが同期や通信のための手順に現れるのに対して、KL1 の誤りの多くは変数の書き誤りである。このような誤りを静的解析によって自動的に検出・修正する研究 [6] が行なわれている。

オブジェクト指向の観点から見ると、プロセスはオブジェクト、プロセスへの通信路はオブジェクト ID と見なすことができる。プロセスへの通信路を指す変数を持っていることは、オブジェクトへのアクセス権を持っていることに等しい。

複数のプロセスから並行にアクセスされるオブジェクトを**サーバ**と言う。これに対して、サーバにアクセスするプロセスは**クライアント**である。

サーバが各クライアントに対して個別に提供する処理を**サービス**と言う。サーバプロセスが複数のクライアントに応対する必要があるのに対して、サービスプロセスは 1 つのクライアントの要求に応えるだけで済み、対応するクライアントが要求を終えた時点で消滅して良い。

4 dklic の設計目標

本節では、まず dklic が前提とする分散環境について述べ、次に dklic 言語の設計目標について述べる。

1. ノードが動的に増減し、全順序を付けるなどの全域的な把握は困難である。また、全てのノードで同じ時刻を示すような全域的な時計は存在しない。KLIC の分散メモリ版はノードに全順序が付けられていることを前提としているため、広域分散環境にそのまま適用することはできない。
2. 各ノードの機種、OS は統一されていない。KLIC の分散メモリ版は全てのノードで機械語コードを共有するため、ヘテロな分散環境にそのまま適用することはできない。
3. ノード間の通信は予測できない遅延時間を要する。
4. 各ノードは TCP/IP 接続されており、パケットの損失や追い越しが起きることはない。

正しい広域分散アプリケーションを容易に記述するため、dklic の設計にあたっては、ネットワーク透過性を保ちながら遠隔ノードへ計算を依頼できる機能の提供を第一目標とした。具体的には、次の 3 つの機能である。

1. 遠隔述語呼出し
2. 述語移送
3. サービス検索

この 3 つの機能はそれぞれ Java RMI、Java クラスローダ、CORBA や Jini に相当する。

dklic では分散論理変数を実装することによって、ネットワーク透過性を保つ遠隔述語呼出しを実現した。本論文では遠隔述語呼出しの実現を主に論じる。

述語移送 [9] とサービス検索 [8,12] は現在の dklic には含まれていないが、いずれも試作が存在する。サービス検索については 9 節で触れる。

5 dklic の実装方針

dklic を実装するまで、コンパイラを備えた分散 KL1 言語処理系は存在しなかった。このため、dklic の実装にあたって次の方針を掲げた。

1. 100% KL1 による記述で移植性を保つ。
2. 単純な実装によって実装コストを抑える。ノード実行系として KLIC を用いる。KLIC 実行系自体には手を入れず、実行時ライブラリによって KLIC の拡張を行なう。

ノード実行系として KLIC 実行系をそのまま利用することによって、ノード内の変数管理や GC を考慮する必要はなくなった。しかし、この方針のため、dklic は KL1/KLIC の制限に捕らわれている。具体的には、未定義変数同士の同一性の検査が行なえず、変数や値に対する参照数を感知することができない。

現在は dklic を次の 3 つのファイルから成る実行時ライブラリとして提供している。

- remote.kl1 — スタブとスケルトン
- vario.kl1 — ソケットに対するデコレータ
- vart.kl1 — ソケットの両端に張り付ける変数表

プログラマは dklic を API として利用し、ユーザプログラムと一緒に KLIC でコンパイルして使用することができる。

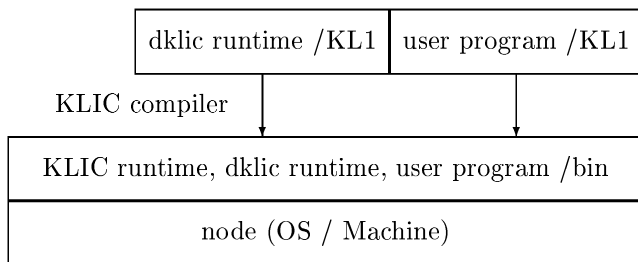

図 2: dklic 処理系の構成

6 遠隔述語呼出しの例

本節では遠隔述語呼出しの例として ping サービスを示す。

クライアント ping.kl1 は、遠隔ノード Host にある pingd:pingd 述語を遠隔呼出ししている。

```
1   :- module main.
2
3   main :- true |
4     klicio:klicio([stdout(RO)]),
5     main(RO).
6
7   main(normal(O)) :- true |
8     unix:argv([Host | _]),
9
10    % pingd:pingd(Ts) @node(Host),
11    remote:call(pingd:pingd(Ts), Host),
12
13    O = [fwrite(Host),
14      fwrite(": pinging.\n") | O1 ],
15    putt(O, 0, Ts, O1).
16
17  putt(O, N, Ts, O) :- wait(N) |
18    O = [putt(N), fwrite(" sent, "),
19      fflush(F) | O1 ],
20    Ts = [N, T | Ts1],
21    putt(F, N, T, Ts1, O1).
22
23  putt(O, N, T, Ts, O) :- wait(T) |
24    O = [putt(T), fwrite(" received.\n"),
25      fflush(F) | O1 ],
26    N1 := N + 1,
27    putt(F, N1, Ts, O1).
```

遠隔述語呼出しは、@node(Host) というプラグマが付いていることを除けば通常の述語呼出しと全く同じ形である。ただし、現在の dklic では remote:call API を用いて遠隔述語のメタ呼出しを行なう。

次に示すサーバ pingd.kl1 は、pingd:pingd の引数ストリームに値が流れてくる度に同じ値を返す。

```
1   :- module main.
2
3   main :- true |
4     pingd:dummy,
5     remote:called.
6
7   :- module pingd.
8
9   dummy :- true | true.
10
11  pingd(Ts) :- Ts = [] |
12    true.
13  pingd(Ts) :- Ts = [T | Ts1] |
14    Ts1 = [T | Ts2],
15    pingd(Ts2).
```

何もしない述語 pingd:dummy を呼び出しているのは、pingd モジュールが削除されることを防ぐためである。静的に述語呼出しされないモジュールは KLIC コンパイラの最適化によって削除されてしまう。

remote:called 述語はサーバソケットを開き、接続されたクライアントからゴールを受け取ってサーバ側で生成する。この場合、dklic ライブラリ内の remote:called が複数のクライアントに応対する真のサーバであり、pingd:pingd はサービスである。

このクライアントとサーバを実行するには、dklic を構成する 3 つのファイル (remote.kl1, vario.kl1, vart.kl1) と一緒に KLIC 処理系でコンパイルすれば良い。

コンパイル方法と実行結果を次に示す。

```
% klic -o pingd pingd.kl1 \
?  remote.kl1 vario.kl1 vart.kl1
%
% klic -o ping ping.kl1 \
?  remote.kl1 vario.kl1 vart.kl1
%
% ./pingd &
%
% ./ping localhost
localhost: pinging.
```

```
0 sent, 0 received.
1 sent, 1 received.
2 sent, 2 received.
        :
```

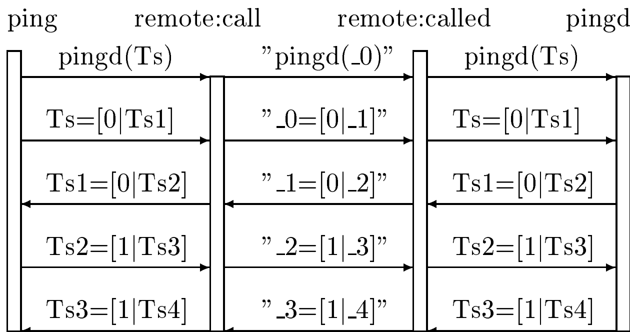

図 3: 遠隔述語呼出しのインタラクション例

クライアント ping が遠隔述語呼出しを実行すると、クライアント側のノードでスタブプロセス remote:call が生成される。remote:call は、サーバ側のノードで事前に起動されていたスケルトンプロセス remote:called に接続し、遠隔述語呼出しゴールを送信する。remote:called は遠隔述語呼出しを受信するとサービス pingd を生成する。(図 3)

その後にやりとりされるゴール群は自ノードで具体化された変数を遠隔ノードに通知する単一化ゴールである。スタブとスケルトンは遠隔ノードで具体化された変数の単一化ゴールを受信すると、自ノードでその単一化ゴールを実行する。

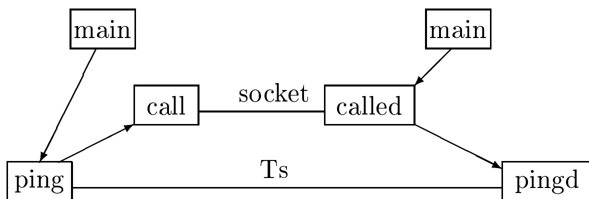

図 4: ping サービスのプロセス生成木

ユーザプログラムからは、ping の要求に直接 pingd が応答しているように見える。実際には、スタブとスケルトンが未定義変数の具体化を感知して単一化ゴールの直列化と復元を行なう。

スタブとスケルトンの間の通信プロトコルは単純で、送信するデータ構造の中の未定義変数以外の部分は KLIC の wrap 機能を用いてそのままバイト列化する。次節では未定義変数の直列化について述べる。

7 分散論理変数の実現

本節では遠隔述語呼出しを可能にする分散論理変数の実装について述べる。

並行論理型言語の変数は単一代入であり、未定義状態を持つ論理変数である。プロセス間通信に利用されるストリームは頭から徐々に具体化されるリストであり、述語呼出しの引数として渡される。

変数の輸出入とその具体化によって、このようなストリームを表現できる。プログラマに意識させずに変数と具体化の輸出入を行なうネットワーク透過な分散論理変数が必要である。

dklic では変数表を導入して、輸出入された変数に ID を付けて管理し、自ノードで具体化が行なわれた時にその具体化を遠隔ノードに伝えることにした。輸出された未定義変数に対して ID を発行することが課題であった。

KLIC ではプログラム内部から未定義変数同士の同一性を検査することができない。一度具体化された変数はその値を比較できるのだが、未定義変数の値を参照するとプロセスが中断し、具体化されるまで待ってしまう。このため、dklic では全ての変数を異なる変数と見なし、同一の変数でも輸出される度にエントリを登録する。

分散変数のエントリを登録するには変数に ID を発行する必要がある。プログラム文面上の変数名は利用することができない。なぜなら、変数名は複数の変数を表している可能性があるからである。dklic では、エントリ毎に新しい整数を発行し、それを ID とした。例えば、ping に対して次のような ID を発行する。

```
< pingd:pingd(_0)
< _0 = [0, _2 | _4]
> _2 = 0
< _4 = [1, _6 | _8]
> _6 = 1
< _8 = [2, _10 | _12]
> _10 = 2
     :
```

_ 付きの整数が、発行された ID を表す。行頭に < が付いているものはスケルトンへの遠隔述語呼出しま

たは具体化通知であり、＞が付いているものはスケルトンからの具体化通知である。

新しい ID を発行したパケットが相手に受信されるまでに、相手が同じ ID を別の変数に発行してしまう可能性があるため、発行者によって ID 空間を分ける必要がある。dklic ではスタブが偶数、スケルトンが奇数を ID として発行することにした。

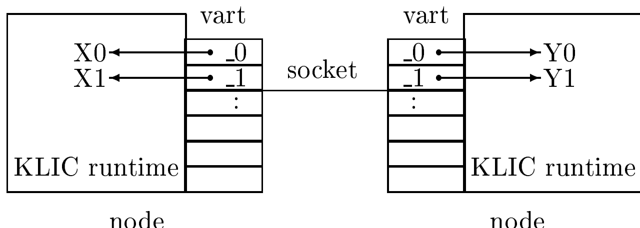

図 5: ノード内外の変数の対応付け

各分散変数は両端ノードの内部変数と両端の変数表エントリによって表現される。変数表の各エントリは ID と内部変数の対である。*ID* は外部に対する変数の識別子である。**内部変数**は識別子 ID が対応付けられた、ノード内部（KLIC）の変数である。（図 5）

変数が具体化されたことは変数の値が未定義でなくなることによって検出できる。変数表に登録された内部変数が具体化されると、変数表はその変数と具体値との単一化ゴールを直列化して通信相手に送信する。

通信相手から直列化された単一化ゴールを受信すると、変数表はその単一化ゴールを復元してノード内部で実行する。具体的には、その ID を持つ内部変数と具体値との単一化を行なう。

以下では分散単一化、分散 GC、変数表の最適化についてこの順に述べる。

7.1 分散単一化

Distributed Oz では各分散変数にマネジャが 1 個付き、各ノードに置かれたプロクシからの単一化要求を管理する。並行に複数の単一化要求が出されると、そのうち 1 つだけが実行され、その単一化が成功した後に他の要求が実行される。

dklic では両端ノードの KLIC 実行系が単一化を行なう。両端ノードで同時に単一化が行なわれると、スタブとスケルトンが自ノードの単一化を通知し合い、通知された遠隔ノードの単一化を自ノードで実行する。

両端ノードで同じ変数に異なる具体値を単一化すると単一化の失敗が起きる（図 6）が、これは逐次プログラムの場合と同じセマンティクスである。片方のノードが何らかの失敗で落ちた時、ソケットのエラーを検出してもう片方のノードが落ちないようにすることができるが、現在の dklic では実装していない。

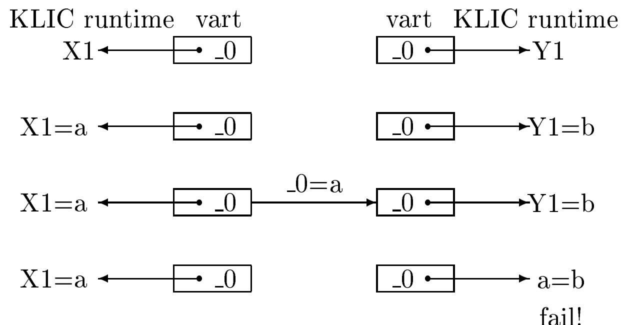

図 6: 分散単一化の失敗

7.2 分散変数管理

KL1 の論理的な計算モデルにおいては、変数の具体化を格納する制約記憶は単調増加する。しかし、KL1 の実装においては GC が行なわれ、不要になったヒープ上の制約記憶領域は再利用される。

dklic の変数表でも参照されなくなった変数を削除する必要がある。削除を行なわないと変数表が無駄な空間を使うばかりでなく、他ノードから参照されなくなったデータの GC が行なえなくなる。

dklic はノード内の実行環境を KLIC に任せているので、内部変数の GC のためには単に変数表から内部変数への参照をなくせば良い。分散変数の占有する他の資源は ID と変数表のエントリである。

分散変数は両端ノードの内部変数にそれぞれ対応付けられているため単一参照（書き込み、読み込みのための参照が 1 つずつ）であり、両端ノードで具体化が感知されれば削除することができる。具体的には変数表のエントリを削除し、ID を発行者に返却すれば良い。エントリが削除されると、そのエントリによって参照されていた内部変数は KLIC 実行系によって適宜 GC される。

ID の再利用にはパケットの損失や追い越しはないという前提（4 節）が必要である。ID を返却するパケッ

ト（相手方からの単一化ゴール）は、その ID を再利用するパケットよりも早く受信されなければならない。

自ノードで ID を発行した分散変数を具体化する場合は、その ID を直ちに再利用する最適化が可能である。次の通信例では、ストリームに自身の ID を局所的再利用している。

```
< pingd:pingd(_0)
< _0 = [0, _2 | _0]
> _2 = 0
< _0 = [1, _2 | _0]
> _2 = 1
< _0 = [2, _2 | _0]
> _2 = 2
      :
```

ID の再利用を行なえば、変数表のワーキングセットはさほど大きくならないと期待できる。なぜなら、プロセス間通信に多用されるストリームは末尾が未定義であるようなリストであり、同時に 1 つの変数しか必要としないからである。また、返事をもらうために未完成メッセージがストリームに流れても、その変数の具体化がさほど遅れることはないと仮定できる。現在は変数表の構造をリストで表現しているが、この仮定を置くことによって配列へ変更することができる。

7.3 分散変数に関する考察と最適化

7.3.1 第三者中継の削除

上記の実装方式に基づく dklic でノードを渡り歩くようなエージェントを記述すると、出発地と現在地との間の通信はエージェントの経由地点の内部変数を中継して行なわれる。これは時間および空間効率を損なうばかりでなく、セキュリティとフォールトトレランスの点からも問題である。同様に、ストリーム併合器のように未定義変数同士を単一化して終了するプロセスも、無駄な中継通信をしつづける可能性がある。

このような第三者による中継を排除するには、中継に使われる内部変数が変数表にのみ出現することを保証もしくは感知することが必要である。KLIC は、個々の論理変数の参照数（出現の数）を調べる言語機能を持たないので、中継排除のためには、参照数体系および参照数解析 (linearity analysis) [5] によって、分散変数を linear な（出現数が 2 の）変数に限定することが第一歩となる。もし分散変数が linear であれば、変数表のエントリを適切なタイミングで調査して、同一の、もしくは異なる変数表の中に同一変数が 2 回出てくることを検知できれば、その変数が中継にしか使われていないことがわかる。したがってその変数の他ノードにおける（二つの）端点同士が直接通信をするように分散変数を再構成すればよい。再構成のためのプロトコルは現在設計中である。

上記の最適化を施すためには、二つの未定義変数が同一であることを感知する必要がある。ところが現在の KLIC では

```
p(X,X) :- true | ...
```

という節は、p の二つの引数が同一の未定義変数であっても起動されないという意味論を採用しており、未定義変数の同一性を感知できるようにするには、KLIC 実行系を改造するか、あるいは generic object [4] 機構を用いて C 言語で本機能を実装する必要がある。

7.3.2 プッシュ配信

現在の dklic の分散変数はプッシュ配信を採用している。これは、分散変数がメッセージ通信に使われること、および（それに伴って）送信したデータは原則として読まれることを前提としているためである。もう一つの理由は、プログラムを要求駆動的に記述すればプル型と同様の挙動も実現できるからである。

一方、KLIC の分散メモリ版ではプル配信を行なっている。これは無駄な通信を起こさない代わりに、広域分散環境においてはプル要求から応答までの通信遅延が無視できなくなる。またプル型配信では分散 GC の必要性が高まる。

dklic は強力なプログラム解析の併用を将来目標としており、必要な通信を静的解析によって翻訳時に決定することも考えられる。

8 遠隔サーバの例

本節では、これまでに紹介した dklic の機能で記述することのできない遠隔サーバの例としてチャットサーバを示す。

遠隔述語呼出しはクライアントが述語を呼び出すため、クライアントによってプロセスが生成されてしまう。すなわち、遠隔述語呼出しの枠組みでは複数の接続をまたいで状態を保持するような真のサーバを書くことができない。書くことができるのはサービスである。

次に示すのは、KLIC のソケットインタフェイスを用いたチャットの例である。現在の dklic で同様のプログラムを書くには、remote:called API に手を入れる必要があり、結局 KLIC のソケットインタフェイスに触れることになる。

クライアント chat.kl1 は標準入出力とサーバへの入出力を開き、標準入力で 1 行読み込む度にサーバへ出力し、それと並列にサーバから 1 行入力される度に標準出力へ書き出す。

```
1   :- module main.
2
3   main :- true |
4     unix:argv([Host | _]),
5     unix:unix([stdin(RI)]) @node(1),
6     unix:unix([stdout(RO),
7       connect2(inet(Host,654321),RSock) ]),
8     main(RI, RO, RSock).
9
10  main(RI, RO, RSock) :-
11      RI=normal(I), RO=normal(O),
12      RSock=normal(Cast,Chat) |
13    lio:lio(LI, I), lio:lio(LChat, Chat),
14    lio:lio(LO, O), lio:lio(LCast, Cast),
15    getl(LI, LChat) @lower_priority,
16    getl(LCast, LO).
17
18  getl(LI, LO) :- true |
19    LI = [getl(L) | LI1],
20    putl(L, LI1, LO).
21
22  putl(L, LI, LO) :- L="." |
23    LO = [putl(L), fflush(F)],
24    wait(F, LI, []).
25  otherwise.
26  putl(L, LI, LO) :- string(L,_,8) |
27    LO = [putl(L), fflush(F) | LO1],
28    wait(F, LI, LI1),
29    getl(LI1, LO1).
30
31  wait(F, LI, LI1) :- F=0 |
32    LI = LI1.
```

サーバ chatd.kl1 はクライアントからのメッセージをそのまま送り返し、他のクライアントからのメッセージもまたブロードキャストする。

```
1   :- module main.
2
3   main :- true |
4     unix:unix([bind(inet(654321),R)]),
5     main(R).
6
7   main(R) :- R=normal(SSock) |
8     generic:new(merge, Lss, Ls),
9     chat(Chat, Lss, Ls),
10    accept(SSock, Chat).
11
12  accept(SSock, Chat) :- true |
13    SSock = [accept2(R) | SSock1],
14    accept(R, SSock1, Chat).
15
16  accept(R, SSock, Chat) :- R=normal(I,O) |
17    lio:lio(LI, I),
18    lio:lio(LO, O),
19    Chat = [chat(LI,LO) | Chat1],
20    accept(SSock, Chat1).
21
22  chat(Chat, Lss, Ls) :-
23      Chat=[chat(LI,LO)|Chat1] |
24    Lss = {Ls1, Lss1},
25    getl(K, LI, Ls1),
26    putl(K, Ls, LO),
27    chat(Chat1, Lss1, Ls).
28  alternatively.
29  chat(Chat, Lss, Ls) :- Ls=[_|Ls1] |
30    chat(Chat, Lss, Ls1).
31
32  getl(K, LI, Ls) :- true |
33    LI = [getl(L) | LI1],
34    getl(K, L, LI1, Ls).
35
36  getl(K, L, LI, Ls) :- L="." |
37    K = kill,
38    Ls = [],
39    LI = [].
40  otherwise.
41  getl(K, L, LI, Ls) :- string(L,_,8) |
42    Ls = [L | Ls1],
43    getl(K, LI, Ls1).
44
```

```
45   putl(K, Ls, LO) :- K=kill |
46     LO = [putl("."), fflush(F) | LO1],
47     wait(F, LO1, []).
48   alternatively.
49   putl(K, Ls, LO) :- Ls=[L|Ls1] |
50     LO = [putl(L), fflush(F) | LO1],
51     wait(F, LO1, LO2),
52     putl(K, Ls1, LO2).
53
54   wait(F, LO, LO1) :- F=0 |
55     LO = LO1.
```

クライアント／サーバ共に、KLIC-3.003 のソケットがストリームを 1 本しか返さないために入出力が同期化されてしまうというバグを修正したもの [9] を利用している。また、クライアント側では、KLIC-3.003 が標準入力を特別扱いして他の入出力をブロックするというバグを回避するために、標準入力に個別の Unix プロセスを割り当てている。

コンパイルと実行の方法を次に示す。

```
% klic -o chatd chatd.kl1 lio.kl1
% ./chatd &

% telnet localhost 654321

% klic -dp -o chat chat.kl1 lio.kl1
% ./chat -p 2 localhost
```

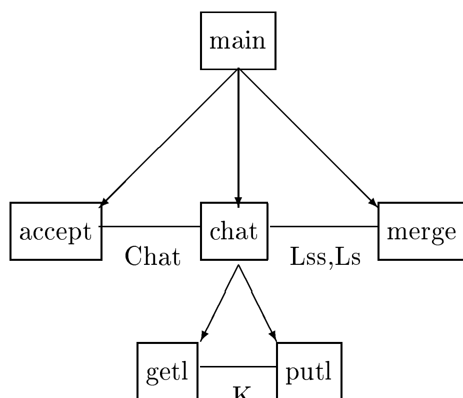

図 7: chatd のプロセス生成木

9 遠隔サーバの実現

本節では、前節のような遠隔サーバへのアクセスを dklic で可能にするための実行系である分散実行系について述べる。

遠隔述語呼出しは、クライアントが直接サーバ（サービス）を述語呼出しする。これは、クライアントからの接続毎にサーバが起動し直されることを意味する。しかし、望みのサーバはクライアントからの接続と無関係に起動し、1 つのクライアントが接続を切っても存続するものである。そうでなければ、複数の接続をまたいで履歴を残すことができない。

KL1 では、述語呼出し木で親戚関係にないプロセス同士が通信を行なうことができない。これは、プロセス間通信に利用されるストリームが述語呼出しの引数として渡され、その他のプロセスから参照できないためである。PIM 上の処理系では OS がサーバストリームを管理しており、プロセスが任意のサーバストリームの登録と取得を要求することができた。

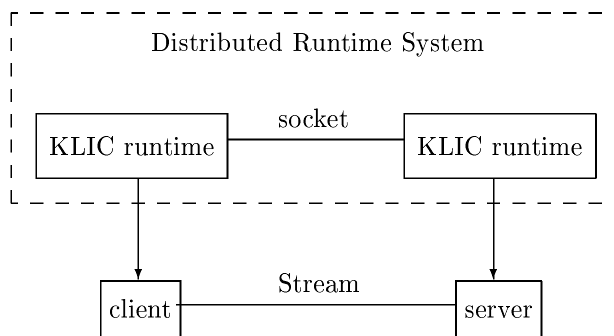

図 8: 分散実行系によるユーザプロセスの生成

dklic では、PIM 上の処理系が単独環境で行なっていたストリーム管理を分散環境で行なうことが目標である。クライアントとサーバが通信を行なうためには、ユーザプログラム同士を兄弟関係とし、分散実行系を共通の親とする必要がある。（図 8）

分散実行系はノード実行系を包含し、ネットワーク透過な分散プログラミング環境を提供するものである。また、プロセスが異常終了しても別のプロセスを道連れにしたり計算資源を占有し続けたりしないように OS のような管理を行なう。

分散実行系を完成させれば、ストリームを用いてプロセスにアクセスできる KL1 のインタフェイスを分散拡張したと言える。分散論理変数の実現も分散実行

系の仕事の一部である。

前節のクライアントはサーバの場所（ホストとポート番号）を知っていたが、KL1/Jのサービス検索 [8,12] はサーバがサービス名とストリームを登録し、クライアントがサービス名によって任意のサーバストリームを要求することができるものである。これを叩き台として分散実行系を構築することができるであろう。

10 まとめと今後の課題

KL1 によって KLIC を拡張することで、遠隔述語呼出しを容易に記述するための実行環境を低いコストで実装することができた。これによって、複数の接続をまたいで履歴を保持しないサービスについては KL1 で記述することができ、逐次の場合とほぼ同じインタフェイスを提供することができた。

遠隔述語呼出しのための実行環境をさらに最適化するには、KL1 による記述では無理があり、generic object 機構を用いた C 言語による KLIC の拡張が必要である。

また、サービスだけでなくサーバの記述を容易にするために、分散実行系の整備を進める必要がある。

分散と関連の深い応用としてプロセス永続化がある。変数の具体化情報の流れを制御することによって、計算途中の状態をファイルに格納したり、通信の中断・再開を行なったりすることが可能になるであろう。

強く型付け・モード付けされた並行論理型プログラムに対しては、通信プロトコルの一貫性を保証することができる。これらの性質は、分散プログラミングを容易にすると期待できる。研究の蓄積された静的解析を分散実装に適用し、最適化を行なうことは今後の重要な課題である。

謝辞

本研究の一部は，科学研究費補助金 (C)(2)11680370 の助成を得て行なった。

また、研究を進めるにあたって、上田研究室言語班における五十嵐宏、加藤紀夫、高山啓、松村量の各氏との議論が参考になった。

参考文献

1. Ueda, K. and Chikayama, T., Design of the Kernel Language for the Parallel Inference Machine. *The Computer Journal*, Vol. 33, No. 6 (1990), pp. 494–500.
2. 瀧和男 編, bit 別冊 第五世代コンピュータの並列処理. 共立出版, 1993.
3. Chikayama, T., Fujise, T. and Sekita, D., A Portable and Efficient Implementation of KL1. In *Proc. 6th Int. Symp. on Programming Language Implementation and Logic Programming (PLILP'94)*, LNCS 844, Springer-Verlag, Berlin, 1994, pp. 25–39.
4. 関田大吾: Inside KLIC Version 1.0. KLIC Task Group / AITEC / JIPDEC, 1998. http://www.klic.org/software/klic/inside/master.html
5. Ueda, K., Linearity Analysis of Concurrent Logic Programs. In *Proc. Int. Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications*, Ito, T. and Yuasa, T. (eds.), World Scientific, 2000, pp. 253–270.
6. 網代育大, 上田和紀, Kima: 並行論理プログラム自動修正系. コンピュータソフトウェア別冊 ソフトウェア発展, Vol. 18, No. 0 (2001), pp. 122–137.
7. 加藤紀夫, 上田和紀, 並行論理プログラムにおける逐次実行部分の抽出方法論. In *Proc. JSSST Workshop on Programming and Programming Languages (PPL2001)*, 2001.
8. 五十嵐宏, 分散 KL1 言語処理系の設計と実装. 早稲田大学理工学部情報学科卒業論文, 1999.
9. 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.
10. 高木祐介, KL1 による分散 KL1 言語処理系の実装. 早稲田大学理工学部情報学科卒業論文, 2000. http://www.ueda.info.waseda.ac.jp/~takagi/
11. 高山啓, 並行論理型言語 KL1 による分散 KL1 言語処理系の実装. 早稲田大学理工学部情報学科卒業論文, 2001.
12. 松村量, 並行論理型言語 KL1 の分散拡張のための遠隔ノード管理の実装. 早稲田大学理工学部情報学科卒業論文, 2001.

13. Van Roy, P., Haridi, S., Brand, P., Smolka, G., Mehl, M. and Scheidhauer, R., Mobile Objects in Distributed Oz. *ACM Transactions on Programming Languages and Systems*, Vol. 19, No. 5, September 1997, pp. 804–851.
14. Haridi, S., Van Roy, P., Brand, P. and Schulte, C., Programming Languages for Distributed Applications. *New Generation Computing*, Vol. 16, No. 3 (1998), pp. 223–261.
15. Haridi, S., Van Roy, P., Brand, P., Mehl, M., Scheidhauer, R. and Smolka, G., Efficient Logic Variables for Distributed Computing. *ACM Transactions on Programming Languages and Systems*, Vol. 21, No. 3, May 1999, pp. 569–626.
16. Diaz, M., Rubio, B. and Troya, J. M., DRL: A Distributed Real-Time Logic Language. *Comput. Lang.*, Vol. 23, No. 2–4 (1997), pp. 87–120.