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

最近、コンピューティングの主流は単独環境から分散環境へと移行しつつある。一度分散環境を経験した者が単独環境を望むことは少ないので、分散コンピューティングの重要性は拡大し続けるだろう。

しかし、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。分散構造、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道具／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。

KL1 [1] は単純な並行論理型言語である。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。また、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。

しかし、これまでKL1の応用は主に並列記号処理を指向していた。KLIC [2] は単純で効率的な並列KL1言語処理系であるが、分散機能の提供はソケットレベルのみである。それ以外に既存のKL1言語処理系は、専用機上の処理系 (PIMOS) とKLICの移植／改良版のみであった。

KLICに依存しない分散KL1言語処理系 [3] も試みられたが、モノリシックで巨大な実装のため未完成である。従って、分散KL1言語処理系の実装を検討することは、分散プログラミングを容易にするためにも並行論理型言語の応用拡大のためにも重要である。

分散KL1 [3] は場所の概念を導入するプラグマ、およびミドルウェアに当たる分散ストリーム管理を提供する。分散プラグマはKLICの並列プラグマに習い、並行計算のネットワーク透過性を保つように設計された。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。

dklic [4] はKL1によって分散APIを記述し、KLICの拡張を行う。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分散KL1言語処理系の核機能を明確に限定することができた。

2 分散KL1プログラムの例

次に示すのは、チャット・クライアントの例である。標準入出力とチャット・サーバへの入出力を開き、標準入力で1行読み込む毎にチャット・サーバへ出力し、それと並列にチャット・サーバから1行入力される毎に標準出力へ書き出す。

遠隔呼び出しは、`@node(HOST)`というプラグマが付いていることを除けば、通常の呼び出しと全く同じ形である。例では省略したが、`HOST`という変数には遠隔ホストのURLを表す文字列が具体化されていなければならない。

```
:- module main.

main :- true |
  unix:unix([stdin(RI), stdout(RO)]),
  server:chat(RChat, RCast) @node(HOST),
  main(RI, RO, RChat, RCast).

main(RI, RO, RChat, RCast) :-
    RI=normal(I), RChat=normal(Chat),
```

```
    RO=normal(O), RCast=normal(Cast) |
  getl(O, I, Chat),
  getl(O, Cast, O).

getl(F, I, O) :- F=0 |
  I = [getl(L) | I1],
  putl(L, I1, O).

putl(L, I, O) :- string(L,_,_) |
  O = [putl(L), fflush(F) | O1],
  getl(F, I, O1).
```

このプログラムの通信例を次に示す。

```
< "This is a pen."
> "This is a pen."
> "That is a desk."
< "I am a student."
> "I am a student."
> "You are a teacher."
```

行頭に < が付いているものはクライアントからのメッセージであり、> が付いているものはサーバからのブロードキャストである。チャット・サーバ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。

3 dklicによるプログラミングの例

現在の dklic は、単純な遠隔呼び出し API を提供している。コンパイラは提供していないため、遠隔呼び出しはプラグマ付きの述語呼び出しではなく、API を利用したメタ呼び出しを行なう必要がある。

次の例は、チャット・クライアントの遠隔呼び出しを dklic の遠隔呼び出し API を利用して書き換えたものである。

```
  % 前略
  % server:chat(RChat, RCast) @node(HOST),
  remote:call(server:chat(RChat,RCast),HOST),
  % 後略
```

行頭に % が付いているものはコメントである。遠隔呼び出し API の利用はプラグマ付きの呼び出しに非常に似ており、変換は容易であることが分かるだろう。

4 分散変数管理

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、遠隔呼び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うネットワーク透過な分散変数管理が必要である。

dklic では変数表を導入して、輸出入された変数に名前をつけて管理し、ローカルに具体化が行われたときにその具体化を遠隔ホストに伝えることにした。輸出入された変数を識別し、命名することが課題であった。

KLIC ではプログラム内部から変数を識別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、未定義状態の変数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入される毎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の命名が必要である。プログラム文面上の静的変数名は利用することができない。なぜなら、静的変数は複数の動的変数を表している可能性があるからだ。dklic では、エントリ毎に新しい整数を生成し、それを変数名とした。先の例に対しては、次のような変数名を生成する。

```
< server:chat(_0, _1)
> _0 = normal(_2)
> _1 = normal(_3)
< _2 = ["This is a pen." | _4]
> _3 = ["This is a pen." | _5]
> _5 = ["That is a desk." | _6]
< _4 = ["I am a student." | _7]
> _6 = ["I am a student." | _8]
> _8 = ["You are a teacher." | _9]
< _7 = []
> _9 = []
```

_ 付きの整数が、生成された変数名を表す。行頭に < が付いているものはクライアントからの遠隔呼び出しまたは具体化通知であり、> が付いているものはサーバからの具体化通知である。

4.1 変数削除

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 の実装においてはガーベジ・コレクションが行われ、不要になった変数記憶は再利用される。

dklic の変数表も不要になった制約を削除する必要があった。削除を行わないと変数表が無駄に大きな空間を使い、計算機のメモリを圧迫してしまう。

4.2 非同期

KLIC にはソケット入出力が同期されてしまうバグがある。dklic の逐次呼び出しではこのバグを回避せず、通信を同期させて遠隔呼び出しを可能にした。

しかし、チャットのように非同期入出力を必要とするアプリケーションは逐次呼び出しでは実現できない。このため、ソケット入出力と分散変数管理を非同期にした並行呼び出し版を実装した。

ソケット入出力を非同期にするには KLIC を修正すれば良いが、分散変数管理の非同期化には並行アルゴリズムが必要であり、逐次呼び出し版で暗黙にあった多くの前提が崩壊した。その結果、並行呼び出し版は逐次呼び出し版とは全く異なる実装となった。

逐次呼び出し版ではホスト間で交互に通信を行い、片方の番が終わってからもう片方の番が始まるので、変数の名前空間が共通であることが保証されていた。しかし、並行呼び出し版では片方で新しく生成した名前が相手に伝わる前に、相手が同じ名前を他の変数に使ってしまう可能性がある。このため、変数の名前空間を分離する必要があった。

4.3 並行

分散変数管理を非同期化するためには、並行な通信プロトコルが必要である。非同期入出力のために少なくとも 2 つの独立なプロセスが必要であり、それらの並行プロセスが非同期に 1 つの変数表を更新できなければならない。変数の輸出入によって変数表を分けることはできない。なぜなら、輸出した変数がどちら側から具体化されるかは分からないからだ。

逐次呼び出し版においては更新操作の逐次性を暗黙の前提として変数表をリストで表現したが、並行呼び出し版においては同じ前提を期待することはできない。このため、変数表をプロセス構造で表現することにした。

4.4 プロセス構造

KL1 では、独立なプロセスは並行である。複数のプロセスは非決定的にスケジューリングされる。しかし、KL1 のデータ型は必ずしも並行でない。ファンクタとベクタでは各要素の処理が同期され、リストは前から順に処理される。

分散変数管理を並行に行なうには、各分散変数を独立なプロセスで表現する必要がある。

逐次呼び出し版では同期で逐次なデータ構造で表現していた変数表を、並行呼び出し版では非同期で並行なプロセス構造で表現した。各分散変数は変数表をサーバとするクライアントであり、変数の輸出によって生成され、具体化の輸出によって削除される。

5 荘園

分散環境には、単独環境にないノイズが多く存在する。また、検出できない失敗も多い。

PIMOS は荘園と言う論理的な計算空間を持ち、プロセスが別の荘園で行なわれた処理の失敗を検出できた。

dklic では、主にサーバの失敗をクライアントが検出できるようにするため、荘園のようなカプセル環境の提供を検討している。

6 分散ストリーム管理

冒頭のチャットの例では、クライアントが直接サーバを呼び出している。これは、クライアントからの呼び出し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呼び出しと無関係に起動し、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。

KLIC では、呼び出し木で親子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呼び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。

PIMOS 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の登録／取得を行なうことができた。dklic では、PIMOS 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課題である。

分散変数管理においてはストリームの時系列を異なる変数として表現できたが、分散ストリーム管理においては同一ストリームの時系列を同一の名前で表す必要がある。なぜなら、分散ストリームは主にサーバ・プロセスのポートであり、複数のクライアントからアクセスされ得るからだ。分散変数の命名法によると、複数のクライアントが同期されクライアント間の順序付けがなされてしまう。実際のサーバへのストリームは、複数のクライアントからのストリームをマージしたものになるだろう。

7 永続プロセス

プロセスを永続化できれば、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。

8 メタ処理系

現在の dklic は単純な実装を目指し、遠隔呼び出し先の述語が相手のホストに存在することを前提としている。

しかし、暗号化サーバのようにローカルホストで行なうべき処理が存在する。だからといって、暗号化サーバを各クライアントが個別に実装するのは無駄である。

このようなサーバは、実行時にローカルホストへ取り込めると良い。述語移送を実現するには、移送された述語を実行するメタ処理系が必要である。

また、分散変数管理において変数の識別について述べたが、メタ変数を扱うことができればこの問題は解決する。

9 オブジェクト指向

KL1 のプロセスはオブジェクトと見なすことができる。ストリームを利用したプロセス間通信は非同期メッセージ渡しに相当する。

しかし、一般に KL1 のプロセスはメタ呼び出しが行えず、first class のオブジェクトとは言えない。KLIC は単独環境においてメタ呼び出しの拡張機能を提供しているが、分散環境においてはメタ処理系を作らなければならない。

分散環境においてプロセスをメタ呼び出しすることができれば、永続プロセスや述語移送の実現も容易になるものと思われる。このような機能を備えた KL1 は、継承のないプロトタイプ・ベースのオブジェクト指向言語であると言える。

分散言語において、あるいは一般にオブジェクト指向言語において継承機能が必要か否かは検討されなければならないが、文面上の継承については delegation へ翻訳すれば良い。現在の dklic は KLIC の利用を前提としているが、コンパイラを 1 から作り上げるなら、もっと効率良く継承を表現することもできる。

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。

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. 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.
3. 五十嵐宏: 早稲田大学理工学部情報学科卒業論文 “分散 KL1 言語処理系の設計と実装”, 1999. http://www.ueda.info.waseda.ac.jp/~igarashi/
4. 高木祐介: 早稲田大学理工学部情報学科卒業論文 “KL1 による分散 KL1 言語処理系の実装”, 2000. http://www.ueda.info.waseda.ac.jp/~takagi/