5N-9

PIE64のデータベースマシンとのインタフェースについての考察

吉田 実, 田中 英彦
東京大学工学部 *

1 はじめに

我々の研究室では、並列推論マシンPIE64の開発をすすめている。[1] PIE64は、データベースマシンをつなぐことによって、大規模なデータを扱うアプリケーションを走らすことができるようになる。ここでは、PIE64とデータベースマシンをつなぐ場合の問題点について考察する。

2 データベース呼び出し

PIE64では、コミッティッドチョイス型言語FLENG[2]を走らすので、FLENGからデータベースにアクセスできるようにする必要がある。通常、データベースマシンではデータベースの整合性を保つトランザクション単位でアクセスするのが常套手段である。よって、次のようにトランザクションを生成する述語を設ける。トランザクションの中間状態は、ユーザプログラムからは見えない。

```
transaction(Result, Transaction)
```

述語transactionは、Transactionで与えられるコマンドを処理し、その処理結果の状態をResultに返す。Transactionは、例えば、以下のようなリストになっている。

```
[replace([name], select((city = "Tokyo"), address),
        Tuples, WriteTuples)]
```

Transactionは、[*command1, command2, ...*]というリストになっていて、各コマンドは、*commandName(arg1, arg2, ...)*という形式である。replaceというコマンドなら、replace(*AttributeListToReplace, Relation, Tuples, WriteTuples*)という形になる。*AttributeListToReplace*には、書き換える必要のあるアトリビュートのリストをかく。アトリビュートには、必要ならリレーションの名前をプリフィックスとしてつける。リレーションには関係代数演算を書くことができる。同じ名前のリレーションが2度以上でてきてあいまいになるときは、alias(*RelationName, AliasedName*)を使って、リレーションの名前を変える。例えば、

```
[alias(family, family1),
read(join((family.father = family1.son),
family, family1), Ts)]
```

*PIE64 - Interface to the database machine, YOSHIDA Minoru, TANAKA Hidehiko, University of Tokyo

```
<Transaction> ::= [<Command> {, <Command>}]

<Command> ::= replace(<AttributeList>, <Relation>,
                  <Tuples>, <Tuples>)
            | delete(<OrgRelation>, <Relation>)
            | insert(<OrgRelation>, <Tuples>)
            | alias(<Relation>, <Relation>)
            | read(<Relation>, <Tuples>)

<Relation> ::= select(<Condition>, <Relation>)
             | join(<Condition>, <Relation>,
                   <Relation>)
             | union(<Relation>, <Relation>)
             | project(<AttributeList>, <Relation>
             | cartesianProduct(<Relation>,
                              <Relation>)
             | setIntersection(<Relation>,
                             <Relation>)
             | thetaJoin(<Condition>, <Relation>,
                       <Relation>)
             | naturalJoin(<Relation>, <Relation>)
             | division(<Relation>, <Relation>)
             | <OrgRelation>
```

表1: Transaction Commands

のように使う。Tuplesには、結果のリレーションの各タプルが返ってくる。*WriteTuples*は、*AttributeListToReplace*でかかれた書き換えるべきリレーションのアトリビュートの値をいれる。そうすると、*Tuples*に対応する部分のタプルが書きかわる。コマンドとしては、その他にread(*RelationName, Tuples*), delete(*RelationName, Relation*), insert(*Relation, Tuples*)などを用意する。トランザクションの内部では、ユーザから見て、逐次性が保たれる。例えば、

```
[read(address, Tuples), delete(address, Tuples)]
```

を処理する場合には、readが終了したあと、一括してdeleteが行なわれるように見える。トランザクションのコマンドの一部を表1にあげる。例での*Tuples, WriteTuples*のような入出力用のストリームの管理は、他のプロセスがストリームの枠組をつくり、transactionが作ることはしない。これは、巨大なリレーションを作ってしまったとき、全てをPIE64のメモリ上に書き出してしまい、メモリがなくなることを防ぐためである。

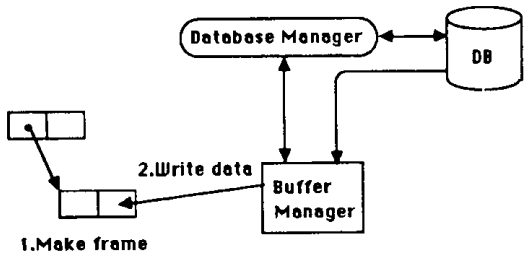

図 1: a Structure of reading data from a database machine

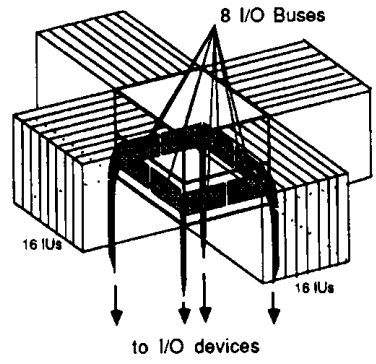

図 2: I/O interface of PIE64

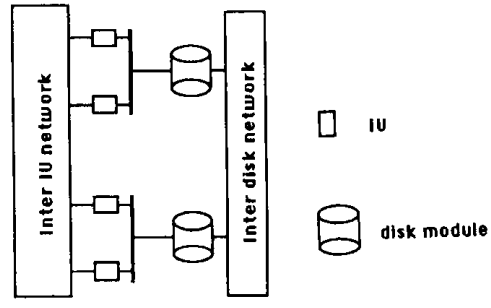

図 3: Network

3 処理系

図1のようK、データベースマネージャはバファマネージャK有限長バッファ通信[3] を行なわせ、例えば、データ読みだし時Kはリストセルが作られたらすぐK値を代入できるようKする。有限長バッファ通信をユーザのストリーム上でやるという方法もあるが、差分リストを管理しなければならないので、ユーザKシステムプログラムほどのことをさせるのはやや負担Kなると思われる。しかし、そのことがユーザK見えなければ構わないので、 FLENG++ のなどで、ユーザから見えないようKする機構がつけば直接差分リストを管理する方法も使える。一方、 PIE64 とデータベース間の通信では、プロトコルを決めて通信することKなるが、ポインタのようなものを渡しておき、サスペンドしたらバインドフックをかけておき、リストセルが作られたら、書き込むというような、 FLENG 処理系の機構はデータベースマシン上Kは持ち込まない。変数の具体化の監視は PIE 上で行ない、インタフェースプロセスが、上限値のあるバッファリングをして出力ストリームの具体化を待っているようKする。データベースとの間では、 PIE のプロセッサ間通信のようKメモリ結合がないので、ポインタを渡すことは二度手間Kなるため、本当K必要なデータのみを値で渡すようKする。上の例なら、*Tuples*を project で必要なデータだけを PIE64 KもってくるようKする。また、データベースユニット内でできることはなるべく内部ですましてしまい、 PIE はコマンドを送るだけで実際のデータはできる限り取りこまないようKする。基本的なデータベース操作はかなりの部分、データベースユニット内で行うことができるはずである。

4 ハードウェアのインタフェース

ハードウェアのインタフェースは、実際Kどのような通信が起きるかを見て決定する必要があるが、 PIE64 の IU 間ネットワークKはハードウェアの制約上、新たKデータベースインタフェース用のネットワークを挿入するのはほとんど困難なので、図3のようK、別Kネットワークを構築する。物理的Kは、図2のようKなる。

5 まとめ

PIE64 とデータベースマシンのインタフェースKついて考察した。ハードウェア的Kは、それほど高速なインタフェースは、普通のアプリケーションK対しては要求されない。ソフトウェア的Kは、トランザクション単位での処理を行うことKよって、複数のプロセスからデータベースを整合性を保ってアクセスすることができる。

参考文献

[1] 小池, 田中,“並列推論エンジン PIE64 の全体構成”, 本大会, (1988)

[2] Nilsson M. and Tanaka H., “FLENG Prolog - Turning supercomputers into Prolog machines”, Proc. Logic Prog. Conf. '86, Tokyo, June, 1986

[3] Takeuchi, A. and Furukawa, K.,“Bounded Buffer Communication in Concurrent Prolog”, New Genaration Computing, Vol.3, No.2, pp.145-155, 1985