データベース・システム　72−14
（1989. 7. 21）

# 演繹データベースの並列処理方式の実現と資源割り当て方式

# Implementation of a parallel processing scheme for deductive databases and resource allocation strategies

波内みさ*　　清木　康**　　劉　澎**
Misa NAMIUCHI　Yasushi KIYOKI　Peng LIU

*筑波大学理工学研究科
**筑波大学電子・情報工学系
University of Tsukuba

あらまし　我々は、演繹データベースに対する問い合わせの並列処理方式を提案し、並列処理システムＳＭＡＳＨ上に、その処理系を実現した。ＳＭＡＳＨは、データベースおよび知識ベースの多様な応用分野を支援するために、柔軟かつ拡張性の高い並列処理環境を実現している。本並列処理システムにおいては、問い合わせの処理効率の向上のために、計算機資源の割り当てが重要である。本稿では、実現した処理系を用いて行った問い合わせ処理実験の結果を示し、計算機資源の割り当て方式について述べる。

Abstract　We have proposed a parallel processing scheme for logical queries, and implemented it on our parallel processing system SMASH. The SMASH system provides flexible and extensible parallel processing environments for supporting various applications of databases and knowledge bases. In our parallel processing system, resource allocation is very important to enhance the processing performance. In this paper, we show some experimental results of logical query processing and present efficient resource allocation strategies.

## 1　はじめに

データベース・システムの普及に伴い、その応用分野が多様化してきている。従来のデータベース・システムは、事務情報の処理を主要な対象として構成されているので、応用分野の多様化に十分に対応することができない。そこで、データベース自身の枠組みの拡張に関する研究が行われている。

我々は、データベースの多様な応用分野および知識ベースを対象とした並列処理システムＳＭＡＳＨを設計し、その実現を行っている（[2]、[4]）。ＳＭＡＳＨは、データベースおよび知識ベースへの問い合わせを実行するための任意の基本演算を関数としてシステム内に統合し、それらを関数型計算の枠組みの中で並列に実行する環境を実現することにより、データベースの多様な応用分野に柔軟に対応することができる。ＳＭＡＳＨでは、関数としてシステムに統合された基本演算を、要求駆動型評価による関数型計算により並列に実行する。このシステムの特徴は、関数型計算の概念をデータベース処理に適用することにより、任意のプログラムに内在する並列性を、関数間の並列性として容易に抽出できることにある。また、関数型計算の評価方式の１つとして知られる要求駆動型評価とストリームの概念をデータベース処理に適用することにより、大量データに対する処理を限られた計算機資源の中で実行するための資源割り当てを、関数型計算の枠組みの中で自然に実現することができる。

現在、次の２種類の環境で実現を行い、すでに問い合わせの並列処理の実験を行っている。

①７台のＳＵＮ３ワークステーションを高速ネットワークにより結合した環境
②共有メモリを持つマルチプロセッサ・マシン

本稿では、共有メモリ型汎用マルチプロセッサ・マシン上に実現した演繹データベースの処理系と、その処理系における資源割り当て方式について述べる。ＳＭＡＳＨにおける演繹データベースの処理系の実現の目的は、ＳＭＡＳＨの拡張性の実証、および、推論操作系の支援である。

２章では、演繹データベースに対する問い合わせの並列処理方式について述べる。この方式は、文献[3]において提案した、演繹データベースのストリーム指向型並列処理方式における再帰的問い合わせのインタープリテーション方式を拡張したものであり、限られたプロセッサ資源を有効に用いて並列性を抽出することを可能にする。

３章では、本処理系のメモリ資源割り当て方式に関する実験とその結果について述べる。４章では、３章の実験を指針として、関数間のデータの受渡し量（グラニュラリティ）を決定する中間バッファへのメモリ資源の割り当て方式、すなわち、関数群へのメモリ資源割り当ての方式を示す。５章では、今後の課題と方針について述べる。

## 2　演繹データベースの並列処理方式

演繹データベースは、関係データベースと一階述語論理の概念の統合により実現されるデータベースである。関係データベースに一階述語論理の推論機能を導入することにより、より高度な問い合わせの支援を可能としている。

本稿で述べる演繹データベースの問い合わせ処理方式は、並列処理システムＳＭＡＳＨ上に実現されている。ＳＭＡＳＨでは、データベースを操作するための任意の基本演算を関数としてシステム内に統合し、それらの基本演算を、関数型計算の枠組みの中で並列に実行する環境を提供している。本問い合わせ処理方式は、後向き推論によるインタープリテーション方式を実現する３種類の基本演算により、演繹データベースの問い合わせ処理を行う。

本方式では、演繹データベースのデータ構造を次のように設定している。

(1)関係データベースの各リレーションをファクト節の集合として扱う。同じ述語名を持つファクト節群は、その述語名をリレーション名とするリレーションとして表現される。
(2)ルール節群は、ホーン節の形式で各ルール毎に独立に表現される。

### ２．１　並列性の抽出

ＳＭＡＳＨでは、要求駆動型評価のもとで、関数計算に内在する次の２種類の並列性が引き出される[2]。

(1) 関数引数の並列評価
(2) 関数の適用側とその本体側、すなわち、ストリームの生産者と消費者の間での並行評価（ストリーム型並列処理）

演繹データベースの問い合わせに内在する並列性は、基本的には、論理型計算におけるホーン節レベルのＯＲ並列性、および、ＡＮＤ並列性に対応する。本処理方式では、処理のための基本演算を、ホーン節の集合としてみなせるリレーションを対象とした集合演算のレベルに設定し、集合を単位としてそれらの各基本演算あるいは基本演算群を並列に処理する。

### ２．２　問い合わせ処理のための基本演算

演繹データベースの問い合わせを実行するために、図１に示す３種類の基本演算（AND、R_unify、F_unify）を設定した[3]。これらの３つの基本演算は、各々関数として定義され、生産者側関数から、ホーン節中の変数と値の束縛環境を表す集合”old binding environments”を引数としてストリームの形式で受け取り、消費者側の関数に対して、新しい束縛環境の集合”new binding environments”をreturn valueとしてストリームの形式で返す。各関数は、論理型プログラムの実行におけるＡＮＤ／ＯＲグラフの各ノードに対応しているが、論理型プログラムとの相違は、各ノードが、ストリームの形式の大量の束縛環境からなる集合を扱う点にある。ここで、束縛環境は、ホーン節中の全ての変数の束縛情報を表すものである。そして、問い合わせのインタープリテーションは、各R_unify、F_unify関数中で、これらの束縛環境の書換えを行うことにより進められる。

要求駆動型評価における１回のデマンドに対して生成されるストリーム・データの量を、以下では、グラニュラリティ（granularity）とよぶ。各関数間には、ストリーム型の並列性の制御を行うために中間バッファを設ける。この中間バッファのサイズが、ストリーム・データのグラニュラリティに相当する。このサイズを操作することによって、関数間で受け渡されるグラニュラリティを制御することができる。

(1)ＡＮＤ

関数ANDは、内包データベースあるいは問い合わせ中のルール節のインタープリテーションを実行する。最初に、ANDは、ホーン節ボディ部と、ストリーム形式の”old binding environments”を受け取る。そして、ホーン節ボディ部の各リテラルに対してR_unifyあるいはF_unifyの関数インスタンスを生成し、それらの関数とAND自身の間で束縛環境をストリームとして受渡しするためのチャネルを生成する。その後、これらチャネルを介して受け取った束縛環境の集合”new binding environments”をreturn valueとしてストリームの形式で返す。

ANDと各F_unify、R_unify間では、ストリーム型並列性の抽出が可能となる。この並列性は、論理型プログラムにおけるＡＮＤ-並列性に対応している。

(2)Ｒ＿ｕｎｉｆｙ

関数R_unifyは、その生成者関数であるANDからホーン節ボディ部中のリテラルを１つ受け取り、それに対するインタープリテーションを行う。引数データとして、AND、F_unify、もしくは他のR_unifyから”old binding environ-

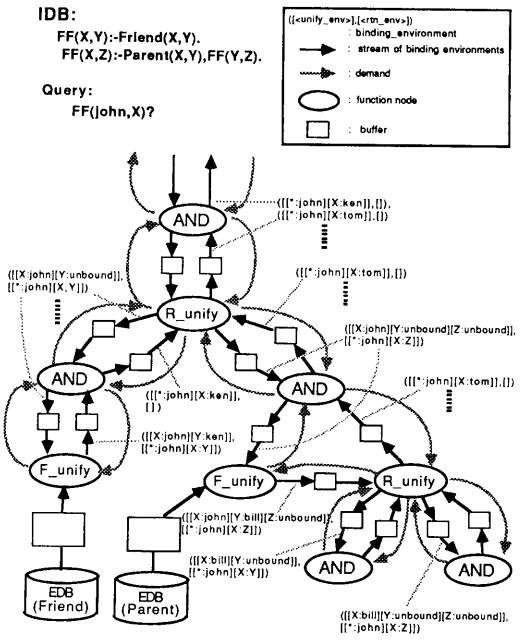


図１：基本演算と問い合わせ処理

ments”をストリームの形式で受け取る。R_unifyは、まず、受け取ったリテラルと同一の述語名をホーン節ヘッド部にもつルールを、内包データベース中から検索する。そして、該当する各ルールに対して、それぞれ、関数ANDのインスタンスを生成し、それらのANDとの間で束縛環境を受け渡すためのチャネルを生成する。次に、ルールの各ボディ部を対応するANDに渡し、さらに、”old binding environments”とルールのホーン節ヘッド部との間で単一化を実行することにより生成される束縛環境の集合を、各ANDにストリームの形式で送出する。

そして、R_unifyが生成した各ANDから”new binding environments”をストリームとして非決定的に受け取り、それらをマージした結果を、AND、R_unify、あるいは他のF_unifyにストリームの形式で出力する。

(3)F__unify

関数F_unifyは、ホーン節ボディ部中の1リテラルをANDから受け取り、そのリテラルに関する変数の束縛環境の集合”old binding environments”と、そのリテラルと同一のリレーション名（述語名）を持つタプル（ファクト群）との間で単一化を実行する関数である。ここでの処理は、tuple-at-a-timeのアプローチとは異なり、”old binding environments”とタプル群との間の単一化を、要求駆動型制御のもとで集合演算として実行する。

F_unifyは、受け取ったリテラルと同一名をもつリレーションからタプル（ファクト）をストリームの形式で受け取り、また、引数として”old binding environments”をAND、R_unify、あるいは他のF_unifyから、ストリームの形式で受け取る。そして、チャネルから読み込まれたタプル群と”old binding environments”との間で、集合演算として単一化を行い、その結果得られた”new binding environments”をAND,R_unify、あるいは他のF_unifyに、return valueとしてストリームの形式で出力する。

## 2．3　再帰的問い合わせの処理方式

本方式では、AND、R_unify、F_unifyの3種類の基本演算が、ルール／ゴール・グラフの各ノードに対応して、関数としてトップダウンに生成され、後向き推論によって問い合わせ処理を行う。これらの関数ノード群は、要求駆動型制御による関数型計算のストリーム指向型並列処理方式の枠組みの中で、並列に実行される。本方式において、末尾再帰的に定義されたルール（tail recursive rules）を参照する問い合わせの並列処理では、ルール／ゴール・グラフを構成する関数ノード群が、プロセッサ資源量を越えて、一時に必要以上多く生成されないように、プロセッサの割り当てを行う。

ここで、1回の再帰呼び出しによって生成される関数ノード群を1グループとし、これらを”世代”と呼ぶことにする。このとき、n台のプロセッサにより、S世代までの再帰呼び出しを伴う問い合わせを処理する場合、関数ノードのプロセッサへの割り当ておよび再帰的処理を次の方法で行う（図2参照）。

(1)1世代目からn世代目までのルール／ゴール・グラフをインタープリテーションによって生成する。このとき、1つの世代の解の処理を行う関数群には、同じプロセッサに割り当てる。これにより、n台のプロセッサによりn世代分の解を並列に生成するルール／ゴール・グラフが用意される。

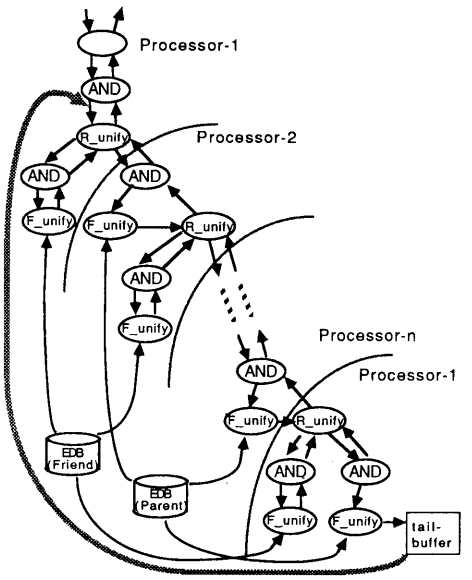

図2：再帰的問い合わせの処理方式

(2)各世代のルール／ゴール・グラフを構成する関数群に、3章で示す資源割り当ての計算方式を用いて、中間バッファへのメモリ資源の割り当てを行う。このとき、第n*i+1世代目（S=n*t, i=1,…,t）を求めるグラフの中間データ（束縛環境）を生成する関数の出力バッファ（以下、tail_bufferとよぶ）には、次世代を計算するために必要となる全中間データを格納するために必要な領域を割り当てる。

(3)n*(i-1)+1世代からn*i世代までのn世代の処理を並列に行う。

(4)(1)において生成したルール／ゴール・グラフと同じものを、(1)で使ったn台のプロセッサによって再度生成し、(2)のtail_bufferに格納された束縛環境を入力ストリームとして、n*i+1世代からn*(i+1)世代までの解を並列に求める。そして、それらの解を完全に生成し終わると、n*(i+1)+1世代以降の解を求めるために、(2),(3),(4)を繰り返す。

この方法により、任意の世代数（S世代）の再帰的処理を、S世代の各世代の計算のための関数インスタンスを一度に生成することなく、n世代分の関数インスタンスだけを繰り返し用いることにより、解を並列に求めていくことができる。また、n世代毎に問い合わせ処理を一度中断することにより、中間データのサイズを正確に把握することが可能になる。したがって、4章において示すメモリ資源の割り当て計算をn世代毎に適用することにより、正確に中間バッファのサイズの設定を行うことができる。

## 3　共有メモリを持つ並列処理マシン上での実現と実験

### 3．1　実現環境

共有メモリを持つバス結合型汎用並列処理マシン（Balance 8000[1], OS:UNIX4.2BSD）上に、ＳＭＡＳＨの基本プリミティブ[2]を実現し、それらの基本プリミティブを用いて問い合わせ処理系を実現した。この問い合わせ処理系は、２章に示した３種類の基本演算により、問い合わせを並列に処理する。ここで用いた並列処理マシンは、４台のプロセッサからなり、８Ｍバイトの共有メモリを持つ。各プロセッサは、８Ｋバイトのライトスルー・キャッシュを備えている。

### 3．2　問い合わせ処理実験の環境設定

本処理方式では、１回のデマンドにより関数間で受け渡されるデータのサイズ（グラニュラリティ）が、ストリーム型並列性に影響を与えるので、その設定が重要である[7]。ＳＭＡＳＨでは、グラニュラリティ（中間バッファ・サイズ）は、基本プリミティブchannelのパラメータとして、柔軟に設定可能である。

中間バッファ・サイズの設定に関して、同じ問い合わせについて、様々なグラニュラリティを設定し、問い合わせ処理の実験を行った。関数間に配置される中間バッファは、次の２種類に分類される。

（１）関数の入力引数に対応するストリーム・データを格納し、それを呼び出し側の関数から受け取るためのバッファ（以下では、入力バッファとよぶ。）
（２）呼び出し側の関数へ、関数の計算結果を返すためのバッファ（以下では、出力バッファとよぶ。）

実験では、再帰的ルールを参照する次の問い合わせを用いた。

```
FF("value" ,X)?          …(a)
```

この問い合わせが参照するルールは、次のように定義されている。

```
FF(X,Y) :- Friend(X,Y).
FF(X,Z) :- Parent(X,Y),FF(Y,Z).
```

これらのルールが参照するファクト群は、リレーションParentとリレーションFriendに格納されており、それぞれ、1024, 2048タプルからなる。単一化（unification）の成功率（選択率；sf(selectivity factor)）は、それぞれ、1/512、1/1024に設定する。

この実験では、３台のプロセッサを用いて、３世代までの再帰呼び出しを行い、処理時間を計測した。

### 3．3　実験結果

問い合わせ実行時において生成されるルール／ゴール・グラフを、図３に示す。グラニュラリティの設定、すなわち、入力バッファ・サイズおよび出力バッファ・サイズを、次のように設定することによって実験を行った。以下では、各バッファ・サイズを、関数間で受け渡される束縛環境あるいはタプルを単位として表す。

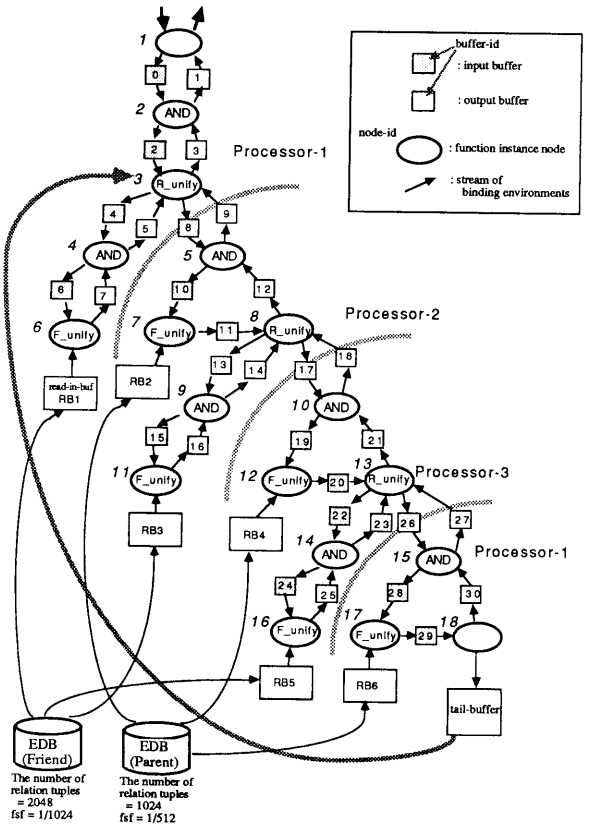


図３：問い合わせ処理において生成されるルール／ゴール・グラフ

［実験１］各関数インスタンスの出力バッファを同じサイズに設定する。そのサイズを、問い合わせの全ての解の個数の１／ｘとし、ｘを1,2,3,…と変化させた場合の各処理時間を比較する。各関数インスタンスの入力バッファは、その関数インスタンスの全入力ストリームを格納するサイズに設定する。したがって、入力データの伝達時のストリーム型並列性は、抽出されない。

［実験２］各関数インスタンスの入力バッファ・サイズを、そのバッファを介して受け渡される全データの個数の１／ｘとし、ｘを1,2,3,…と変化させた場合の各処理時間を比較する。各関数インスタンスの出力バッファは、［実験１］と同様に各々同じサイズであり、さらに、［実験１］において処理時間が最短であったときのサイズに設定する。

実験１の結果を、図4-1（(a)のタイプの問い合わせが１０個入力された場合）、図4-2（８０個入力された場合）に示す。これらより、関数インスタンスの出力バッファ群のサイズを小さくしていくにしたがい、ストリーム型並列性が抽出され処理時間は短縮されていくことがわかる。しかし、あるサイズ（Ｂｏ）より小さくなると、処理時間が長

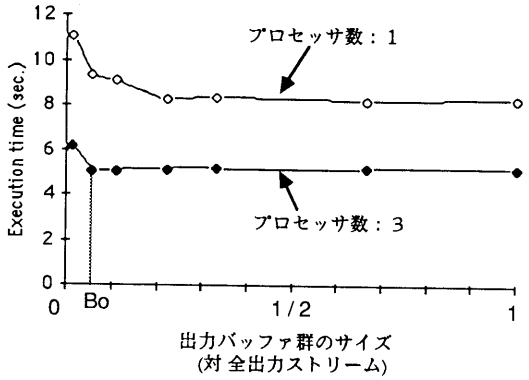

図4-1：［実験１］の結果（入力１０個の場合）

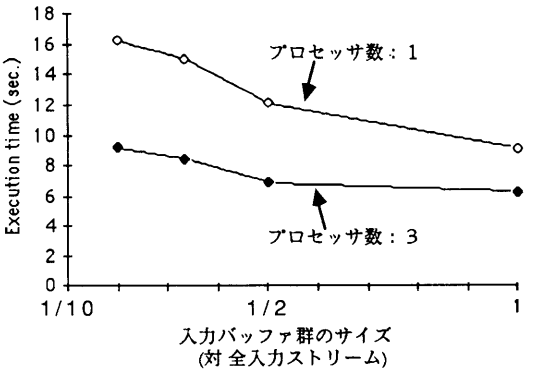

図5-1：［実験２］の結果（入力１０個の場合）

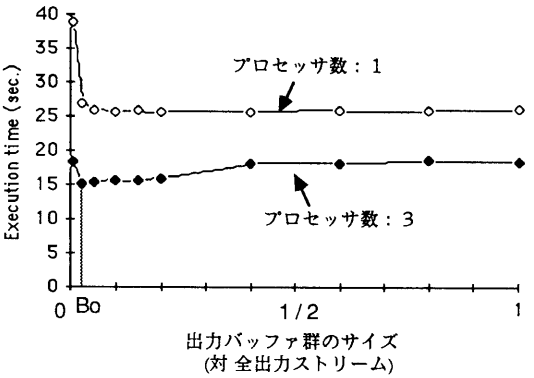

図4-2：［実験１］の結果（入力８０個の場合）

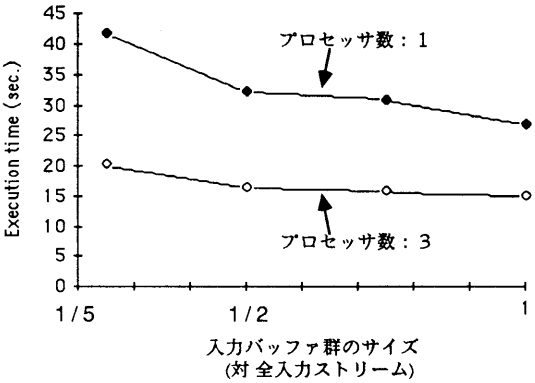

図5-2：［実験２］の結果（入力８０個の場合）

くなっていく。これは、バッファのサイズがＢｏよりも小さくなると、関数を起動するためのコンテクスト・スイッチの回数が多くなり、そのオーバーヘッドが顕著になることによる。

実験２の結果を、図5-1（(a)のタイプの問い合わせが１０個入力された場合）、図5-2（８０個入力された場合）に示す。これらより、入力バッファが入力データをすべて格納できるサイズをもつ場合に最も効率がよく、そのサイズが小さくなるにつれて、実行時間が長くなることがわかる。これは、入力バッファ群のサイズが小さくなると、関数F_unifyでのリレーション・アクセスの回数が増えるからである。

## 4　演繹データベースにおけるメモリ資源割り当て方式

我々は、関係演算において、処理のグラニュラリティと処理時間の関係を式で表し、関係データベースにおけるメモリ資源の最適なアロケーションを与えるアルゴリズムを提案している（[5],[6]）。このアルゴリズムを、提案している演繹データベースの並列処理方式におけるメモリ資源の割り当てに適用する。

本処理方式において、実際にデータベースをアクセスする関数は、F_unifyである。[5]、[6]で提案しているメモリ資源の割り当て方式が適用できるのは、このF_unifyへ束縛環境を渡す中間バッファと、F_unifyが外延データベースからリレーションを読み込むためのバッファ（以下、これを読み込みバッファと呼ぶ）である。したがって、本章では、関数間の中間バッファを、F_unifyでの処理に影響を与えるものと、その他の中間バッファの２種類に分類し、各々について資源割り当ての方式を考察する。そして、それらを統合して、演繹データベースの問い合わせ処理におけるメモリ資源割り当てを行うための指針を示す。

図６は、2.3で示した再帰的問い合わせ処理方式によるＦＦの問い合わせ処理の、ｋ世代目のルール／ゴール・グラフを示している。この図において、F_unifyの処理時間に直接関係するものは、$BS_{k2}$、$BS_{k3}$、$BS_{k6}$、$BS_{k7}$である。

### ４．１　F_unifyに関係する中間バッファへのメモリ資源割り当て方式

関数AND、R_unifyに対して入力ストリーム（実引数）を伝達する中間バッファのサイズは、これらの関数の入力ストリームの生成を要求するデマンドの発行回数、および、コンテクスト・スイッチの回数に影響を与える。しかし、関数F_unifyに関して、入力ストリームを受け取るための中間バッファのサイズは、上記のデマンドおよびコンテクス

ト・スイッチの回数の他に、F_unifyでのリレーション・アクセスおよびデータのソーティングの回数に影響を与える。全体の問い合わせ処理の中で、このF_unifyの処理の割合が非常に大きいので、F_unifyの処理時間を短縮することは重要である。

この関数F_unifyへのバッファの割り当て量を決定するために、次のようなパラメータを導入する。

n：使用可能なプロセッサの数
q：一度に処理を行うことのできる再帰呼び出しの世代（n）を1グループとしたときのグループ番号
k：再帰呼び出しの世代
w：（q－1）グループまでの処理によって生成された束縛環境数；これがqグループへの入力ストリームのサイズとなる。
$F_P$（$BF_P$）：parentリレーションのファクト数（ブロック数）
$F_F$（$BF_F$）：friendリレーションのファクト数（ブロック数）
$T_u$：単一化に必要な時間
$T_s$：クイック・ソートの係数
$T_{bs}$：サーチの係数
$T_{io}$：1ブロックのファクト群をディスクから入出力する時間

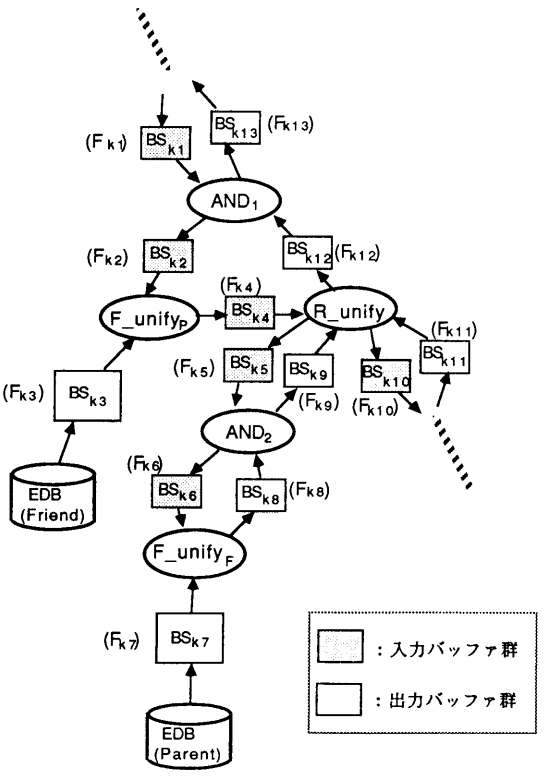


図6：FFの問い合わせにおけるk世代目のルール／ゴール・グラフ

$fsf_P$：parentリレーションの選択率
$fsf_F$：friendリレーションの選択率
$BS_{ki}$：中間バッファのサイズ（単位：タプルあるいは束縛環境数）
$BBS_{ki}$：中間バッファのサイズ（単位：ブロック数）

図6において、各バッファを流れる中間ストリームのサイズは次のようになる。

$$
\begin{aligned}
F_{k1} &= F_{k2} = \omega * (fsf_P * F_P)^{k-1} \\
F_{k3} &= F_P \\
F_{k4} &= F_{k5} = F_{k6} = F_{k10} \\
&= F_{k2} * F_{k3} * fsf_P = \omega * (fsf_P * F_P)^k \\
F_{k7} &= F_F \\
F_{k8} &= F_{k9} = F_{k6} * F_{k7} * fsf_F \\
&= \omega * (fsf_P * F_P)^k * F_F * fsf_F \\
F_{k11} &= \omega * \textstyle\sum_{i=k+1}^{n} (fsf_P * F_P)^i * F_F * fsf_F \\
F_{k12} &= F_{f13} = F_{k9} + F_{k11} \\
&= \omega * \textstyle\sum_{i=k}^{n} (fsf_P * F_P)^i * F_F * fsf_F
\end{aligned}
$$

これらを使って関数$F_unify_P$、$F_unify_F$の処理時間を表すと、以下のようになる。

$AND_1$：
$$T_{AND_1} = T_u * (F_1 + F_{k12})$$

$F\_unify_P$：
$$
\begin{aligned}
T_{F\_unify_P} = {} & \lceil F_{k2}/BS_{k2} \rceil * (T_s * BS_{k2} \log_2 BS_{k2} + \\
& (T_{bs} * \log_2 BS_{k2} + T_u * fsf_P * BS_{k2}) * F_{k3}) + \\
& T_{io} * \lceil F_{k2}/BS_{k2} \rceil * \lceil F_{k3}/BS_{k3} \rceil * BBS_{k3}
\end{aligned}
$$

$R\_unify$：
$$T_{R\_unify} = T_u * (F_{k4} + F_{k9} + F_{k11})$$

$AND_2$：
$$T_{AND_2} = T_u * (F_{k5} + F_{k8})$$

$F\_unify_F$：
$$
\begin{aligned}
T_{F\_unify_F} = {} & \lceil F_{k6}/BS_{k6} \rceil * (T_s * BS_{k6} \log_2 BS_{k6} + \\
& (T_{bs} * \log_2 BS_{k6} + T_u * fsf_F * BS_{k6}) * F_{k7}) + \\
& T_{io} * \lceil F_{k6}/BS_{k6} \rceil * \lceil F_{k7}/BS_{k7} \rceil * BBS_{k7}
\end{aligned}
$$

簡単のために、ここでは、各世代における$F_unify$の実行時間のみについて考える。qグループ内の1～n世代までの処理時間は、次のようになる。

Generation 1:
$$
\begin{aligned}
g_1 = {} & \lceil F_{12}/BS_{12} \rceil * [T_s * BS_{12} \log_2 BS_{12} + \\
& (T_{bs} * \log_2 BS_{12} + T_u * fsf_P * BS_{12}) * F_{13} + \\
& T_{io} * \lceil F_{13}/BS_{13} \rceil * BBS_{13}] + \\
& \lceil F_{16}/BS_{16} \rceil * [T_s * BS_{16} \log_2 BS_{16} + \\
& (T_{bs} * \log_2 BS_{16} + T_u * fsf_F * BS_{16}) * F_{17} + \\
& T_{io} * \lceil F_{17}/BS_{17} \rceil * BBS_{17}]
\end{aligned}
$$

Generation 2:
$$
\begin{aligned}
g_2 = {} & \lceil F_{22}/BS_{22} \rceil * [T_s * BS_{22} \log_2 BS_{22} + \\
& (T_{bs} * \log_2 BS_{22} + T_u * fsf_P * BS_{22}) * F_{23} + \\
& T_{io} * \lceil F_{23}/BS_{23} \rceil * BBS_{23}] + \\
& \lceil F_{26}/BS_{26} \rceil * [T_s * BS_{26} \log_2 BS_{26} + \\
& (T_{bs} * \log_2 BS_{26} + T_u * fsf_F * BS_{26}) * F_{27} + \\
& T_{io} * \lceil F_{27}/BS_{27} \rceil * BBS_{27}]
\end{aligned}
$$

Generation k:

$$
\begin{aligned}
g_k \;=\; & \lceil F_{k2}/BS_{k2} \rceil * [T_s * BS_{k2} \log_2 BS_{k2} + \\
& (T_{bs} * \log_2 BS_{k2} + T_u * fsf_P * BS_{k2}) * F_{k3} + \\
& T_{io} * \lceil F_{k3}/BS_{k3} \rceil * BBS_{k3}] + \\
& \lceil F_{k6}/BS_{k6} \rceil * [T_s * BS_{k6} \log_2 BS_{k6} + \\
& (T_{bs} * \log_2 BS_{k6} + T_u * fsf_F * BS_{k6}) * F_{k7} + \\
& T_{io} * \lceil F_{k7}/BS_{k7} \rceil * BBS_{k7}]
\end{aligned}
$$

$$\vdots$$

Generation n:

$$
\begin{aligned}
g_n \;=\; & \lceil F_{n2}/BS_{n2} \rceil * [T_s * BS_{n2} \log_2 BS_{n2} + \\
& T_{bs} * (\log_2 BS_{n2} + T_u * fsf_P * BS_{n2}) * F_{n3} + \\
& T_{io} * \lceil F_{n3}/BS_{n3} \rceil * BBS_{n3}] + \\
& \lceil F_{n6}/BS_{n6} \rceil * [T_s * BS_{n6} \log_2 BS_{n6} + \\
& (T_{bs} * \log_2 BS_{n6} + T_u * fsf_F * BS_{n6}) * F_{n7} + \\
& T_{io} * \lceil F_{n7}/BS_{n7} \rceil * BBS_{n7}]
\end{aligned}
$$

したがって、qグループの全処理時間は、

$$TG = \sum_{i=1}^{n} g_i.$$

で表すことができる。

資源割り当て決定のアルゴリズムは、以下のとおりである。

各世代間には共通の変数がないので、それぞれ独立に計算を行うことができる。ここで、k世代目からn世代目までに割り当てられるメモリの量をtとし、k世代目からn世代目までの最小の実行時間を$G_k(t)$とする。このとき、全体の最小の実行時間$G_1(t)$を求めるために、以下の手順で計算を行う。

*step 1:* $G_n(t) = g_n(t) \quad (t = 1, 2, \ldots, BS)$

*step 2:* $G_{n-1}(t) = \min_{1 \le r < t} \{g_{n-1}(r) + G_n(t-r)\} \quad (t = 1, 2, \ldots, BS)$

$$\vdots$$

*step k:* $G_k(t) = \min_{1 \le r < t} \{g_k(r) + G_{k+1}(t-r)\} \quad (t = 1, 2, \ldots, BS)$

$$\vdots$$

*step n-1:* $G_2(t) = \min_{1 \le r < t} \{g_2(r) + G_3(t-r)\} \quad (t = 1, 2, \ldots, BS)$

*step n:* $G_1(BS) = \min_{1 \le r < BS} \{g_1(r) + G_2(BS-r)\}$

このアルゴリズムにより、第qグループのすべてのF_unifyに関係する中間バッファについて、処理時間を最小にするためのメモリ資源の割り当てを求めることができる。

## 4．2　F_unifyに関係しない中間バッファへのメモリ資源割り当て方式

（1）入力バッファ群

入力バッファ群は、各関数の処理対象である入力ストリームを伝達するバッファ群である。本処理系では、入力ストリームを速やかに各関数に伝達することが重要である。図6の$BS_{k1}$と$BS_{k2}$の関係について考える。$BS_{k2}$のサイズは、4.1で示したアルゴリズムによって決定されるので、ここでは$BS_{k1}$を対象とする。$BS_{k1}$のサイズが$BS_{k2}$に比べて大きいと、$BS_{k1}$のデータを伝達するために「$BS_{k1}/BS_{k2}$」回のコンテクスト・スイッチが起こる。これに対して、$BS_{k1}$のサイズが$BS_{k2}$に比べて小さいと、$AND_1$バッファ$BS_{k2}$を満たすために「$BS_{k2}/BS_{k1}$」回のデータの伝達が必要である。これによって、$F_unify_P$での処理が遅延してしまう。したがって、$BS_{k1}$と$BS_{k2}$のサイズが同じ場合には、不必要なコンテクスト・スイッチが起こらないので、最も効率よく入力ストリームを伝達できると考えられる。

また、他の入力バッファのサイズは、次のように設定することにより、コンテクスト・スイッチの回数を最小にすることができる。

$$BS_{k1} = BS_{k2}$$
$$BS_{k4} = BS_{k5} = BS_{k6}$$

したがって、最も処理効率を良くする入力バッファ群のサイズは、4.1のアルゴリズムによりF_unifyの入力バッファのサイズを決定した後、そのサイズに合わせて決定すればよい。

（2）出力バッファ群

各関数のreturn valueをストリームとして伝達するのが出力バッファ群である。

(1)で示した入力バッファ・サイズの決定方法と同じ理由から、$BS_{k8}$、$BS_{k9}$および$BS_{k12}$、$BS_{k13}$について次の関係を設定できる。

$$BS_{k8} = BS_{k9}$$
$$BS_{k12} = BS_{k13}$$

$BS_{k8}$のサイズの条件は、$F_unify_F$で生成されたデータが滞りなく出力されることである。つまり、$F_unify_F$の処理が出力待ちの状態で中断されることのないように、サイズを設定する必要がある。

R_unifyでは、複数個のバッファから出力ストリームを受け取り、それらをマージして出力するが、その結果を受け取るバッファのサイズの設定は重要である。図6では、$BS_{k12}$がこのバッファに対応し、R_unifyによってマージされた$BS_{k9}$と$BS_{k11}$のデータを格納する。

$BS_{k12}$のサイズには、次の6通りが考えられる。

① $BS_{k12} = F_{k12}$
② $BS_{k9} + BS_{k11} < BS_{k12} < F_{k12}$
③ $BS_{k12} = BS_{k9} + BS_{k11}$
④ $BS_{k12} = BS_{k9}$
⑤ $BS_{k12} = BS_{k11}$
⑥ $BS_{k12} < BS_{k9}$、$BS_{k12} < BS_{k11}$

このうち、①のサイズの場合には、全出力結果が$BS_{k12}$に伝達され終わるまで出力ストリームを送ることができないので、ストリームの流れが中断される。また⑥の場合には、無駄なコンテクスト・スイッチが起こる。

残る4通りの場合に関しては、各々が有効となる処理の特徴が次のようにまとめられる。

②出力データが$BS_{k9}$、$BS_{k11}$に到着する間隔が非常に短く、$BS_{k12}$のサイズをそれらの和よりも大きく設定した方が、コンテクスト・スイッチのオーバーヘッドが軽くなる場合

③$BS_{k9}$、$BS_{k11}$にデータが到着する時刻がほぼ同じで、②と同じ理由でコンテクスト・スイッチのオーバーヘッドを軽くすることができる場合

④$BS_{k9}$から受け取る出力データの量が$BS_{k11}$のそれより十分大きい場合

⑤$BS_{k11}$から受け取る出力データの量が$BS_{k9}$のそれより十分大きい場合

したがって、処理の性質によってこれらのいずれかを選択的に適用することによって、効果的に並列性を引き出すことが可能になる。

しかし、3章での実験結果より、関数におけるコンテクスト・スイッチは、その回数が非常に多くならない限り処理時間に影響を与えないほど軽い。したがって、メモリ資源の制約が強い処理系では、必ずしも上記の場合分けに従わなくとも、処理の効率はそれほど劣化しない。

### 4．3　演繹データベースにおけるメモリ資源割り当て方式の指針

4.1、4.2において、中間バッファ・サイズの設定方法を提案し、議論を行った。ここでは、これらを統合したメモリ資源の割り当て方式について考察する。

3章における実験により、入力バッファ群のサイズは、入力ストリーム全体を格納できるサイズよりも小さくなると、処理効率が劣化する。これに対して、出力バッファ群は、そのサイズが大きくなるにつれて並列性の減少による処理時間の延長が若干見られるが、コンテクスト・スイッチの影響が現れなくなるサイズ$B_o$よりも大きければ、全体として、処理時間にほとんど影響を与えない。出力バッファ群の大きさは、この実験結果より設定の指針が得られる。

本処理方式では、入力バッファ群と出力バッファ群との間には関係がなく、各々のサイズを独立に設定することができる。したがって、入力バッファ群と出力バッファ群に割り当てるメモリのサイズは、4.1、4.2で述べた方法により独立に決定できる。入力バッファ群に割り当てるサイズとして、入力ストリームをすべて格納できるサイズを確保することができ、かつ、出力バッファ群のサイズとして$B_o$が確保できる場合には、効率のよい処理を行うことができる。しかし、使用できるメモリ量がこのサイズを下回る場合には、入力バッファ群と出力バッファ群とが処理時間に与える影響の度合を考慮して、メモリ資源の割り当てを決定する必要がある。この場合の資源割り当て方法について、現在、検討を行っている。

## 5　おわりに

本稿では、演繹データベースの問い合わせ処理を、関数型計算の枠組みの中で並列に実行する方式について述べた。提案方式では、演繹データベースの処理系の基本演算として、AND、R_unify、F_unifyの3種類の演算を設定した。また、本方式では、末尾再帰的問い合わせの処理に関して、限られたプロセッサ資源のもとで、任意回の再帰的な処理を行う方式を示した。

また、共有メモリ型マルチプロセッサ・マシン上に実現したＳＭＡＳＨにおいて提案方式を実現し、資源割り当てに関する実験を行い、その結果を示した。そして、この実験結果を指針として、並列処理および実行時間に影響を与えるメモリ資源の割り当て方式について示した。

4.1で提案した資源割り当て計算のアルゴリズムは、すでに実現されている。今後、4章で提案した資源割り当て方式について、より詳細に検討を行う予定である。

参考文献：

[1] Balance 8000 Parallel Programming, Sequent computer Systems, Inc. 1985.

[2] Y. Kiyoki, K. Kato and T. Masuda, "A relational database machine based on functional programming concepts," Proc. 1986 ACM-IEEE Computer Society Fall Joint Computer Conf., pp. 969-978, 1986.

[3] Y. Kiyoki, K. Kato, N. Yamaguchi and T. Masuda, "A stream-oriented approach to parallel processing for deductive databases," in Proc. 5th Int. Workshop on Database Machines, pp. 102-115, 1987.

[4] Y. Kiyoki, P. Liu, K. Kato, T. Masuda, "SMASH: an extensible parallel processing system for databases and knowledge bases," The Joint Scandinavian-Japanese Seminar on Information Modelling and Knowledge Bases, 1988.

[5]劉、清木、益田、”ストリーム指向型関係演算処理におけるバッファ資源割り当ての計算方式、”情報処理学会論文誌、Vol. 29, No.8, pp. 770-781, 1988.

[6] P. Liu, Y. Kiyoki and T. Masuda, "Efficient algorithms for resource allocation in parallel and distributed query processing environments," Proceeding of the IEEE International Conference on Distributed Computing Systems, 1989.

[7]波内、清木、山口、劉、益田、”並列処理システムＳＭＡＳＨにおける演繹データベースの処理系と資源割り当て方式、”電子情報通信学会データ工学研究会報告、DE88-7、pp. 49-56, 1988.