計算機アーキテクチャ　62－2
（１９８６．７．２１）

# 並行プログラムの ストリーム型並列処理方式

加藤　和彦*　清木　康**　益田　隆司**

* 筑波大学　工学研究科　** 筑波大学　電子・情報工学系
茨城県新治郡桜村天王台1-1-1

本稿では，関数性を守って記述されたプログラムを，並列に実行する方式について述べる．この方式では，関数計算の理論的背景に基づいて計算機資源が管理される．関数計算の駆動方式として要求駆動型制御を用いることにより，不用な計算を避けることができる．さらに，ストリームの概念に基づいて関数間のデータの受渡しを行うことにより，限られた計算機資源のなかで並列処理を行うことが可能となる．本稿では，要求駆動型評価機構を実現する基本プリミティブを示す．基本プリミティブを組み合わせて関数計算を行うことにより，計算機資源に応じた並列性を引き出すことが可能となる．基本プリミティブを用いて実行される関数の例として，関係データベース演算のプログラムを示す．本方式の特徴を明らかにするために，提案方式に基づいた関係データベース処理系を用いて行った並列処理環境のシミュレーションの実行結果を示す．

# A Stream-oriented Parallel Processing Scheme of Concurrent Programs

Kazuhiko KATO*　Yasushi KIYOKI**　Takashi MASUDA**

*Doctorial Program of Engineering, University of Tsukuba
**Institute of Information Sciences and Electronics, University of Tsukuba
University of Tsukuba, Sakura-Mura, Ibaraki 305, Japan

We present a novel scheme for executing concurrent programs within the framework of functional computation. In this scheme, computer resources are managed on the basis of theoretical neatness of functional computation. By using the demand-driven evaluation as a driving method, parallelism can be exploited within restricted resources avoiding unnecessary computations. We define basic primitives which are used to implement demand-driven evaluation and function application. By using the basic primitves in function computation, parallelism can be exploited fitting to available computer resources. To make features of our scheme clear, some experimental results are shown in this paper.

## 1．まえがき

関数型プログラムは，関数の計算結果が入力引数にのみ依存し，計算の履歴に依存しないという性質を持っている．関数間の依存関係は，引数によるデータの受渡しだけであり，関数の計算中に他の関数の実行の影響を受けることがない．この性質により，プログラムに内在する並列性を引き出すことが容易となる[4]．

関数型プログラミングの主要な概念は次のようにまとめられる。

（1）関数式の値は入力引数にのみ依存し，計算履歴に依存しない[2]．この概念は参照透明性(referential transparency)と呼ばれる．

（2）関数型プログラムの実行は副作用を引き起こさない．従って，引数の受渡し機構である値呼び(call-by-need)と名前呼び(call-by-name)は同じセマンティクスをもつ．

（3）関数型プログラムには，並列計算の可能性があり，その並列性を引き出すことが容易である．

本稿では，関数型プログラミングの理論的背景に基づいて，限られた資源の中で並列処理を実現する方式を提示する．ある一つの系を構成する各プロセスの間で関数性が守られている場合，本方式により，それらのプロセスは要求駆動型制御のもとで並列に実行される．本方式では，関数計算における要求駆動型の評価機構に基づいた多様な並列処理環境を実現するために，数種類の基本プリミティブを提供している．本稿では，関数計算の駆動方式，並列性，引数の評価機構について論じる．本方式を実現するための基本プリミティブを示し，また，それらの基本プリミティブを用いて実行される基本プリミティブの例として，関係データベース演算[3] のプログラムを示す．我々は，要求駆動型制御に基づいた並列処理環境において，関係データベース演算のプログラムが実行される場合のシミュレーションを行った．本稿では，そのシミュレーション結果を示し，本方式の特徴を明らかにする．

## 2．設計思想

### 2．1駆動方式

関数型プログラムにおける関数計算の駆動方式は次のように分類できる[14],[15]．

（1）データ駆動型評価(data-driven evaluation)

（2）要求駆動型評価(demand-driven evaluation)

（3）逐次型評価(sequential evaluation)

本方式では，限られた資源（プロセッサ資源およびメモリ資源）のなかで大量のデータを扱うために，関数計算の駆動方式として要求駆動型評価を採用し，要求駆動型評価を実現するための基本プリミティブを用意する．要求駆動型評価は，次の利点を持つ．

（1）関数引数の評価は，関数の実行中にその引数への参照が起こった時点で行われる．従って，その関数が結果データを生成するのに必要な計算のみが行われ，不用な計算は行われない．

（2）計算機資源が限られた環境のもとで，関数間の巨大な中間データのやり取りを行うことが可能である．

要求駆動型評価では，関数が引数データを参照するときに，引数データの生産者側の関数にデマンドを伝達する必要がある．1回のデマンドにより関数間で受渡しされるデータの量が小さいときは，デマンド伝達のオーバヘッドが問題となるが，1回のデマンドによって引き渡されるデータの量を比較的大きく設定すれば，デマンド伝達のオーバヘッドは，データの転送に対して問題とならない．むしろ，要求駆動方式で引数を評価する事により，関数間のデータの受渡しを限られた資源のなかで行えることによる効果が大きい．

### 2．2 並列性

我々のアプローチでは，関数型プログラムに内在する次のような並列性を引き出す機構を提供する．

（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，データの生産者(producer)とデータの消費者(consumer)の間での並行評価（ストリーム型並列処理）[5]

（1）の並列性は，複数の引数データの消費を行う関数が，その引数データを生産する関数に対するデマンドの発行を同時に行うことによって引き出される．（1）により，データの受渡しによる関係がない，互いに独立な関数の計算が並列に行われる．

（2）の並列性を引き出すためには，データの消費者側の関数は，全データの生成を待つことなく，一部のデータが生成された時点で計算を開始できる性質を持っている必要がある．データ駆動型評価のもとで（2）の並列性を引き出すためには，生産者側の関数が引数データ全体を作り終わるのを待つことなく，消費者側の関数が計算を開始する機構が必要となる[1][8]．要求駆動型評価のもとで（2）の並列性を引き出すためには，消費者側の関数からのデマンドに先行して生産者側の関数が計算を開始する必要がある．本方式では，消費者側の関数が計算を始める前にデマンドを先出しする機構を実現することにより，この並列性を引き出す．この場合，生産者側の関数は，1回のデマンドに対して，一定量のデータを生成したのち，計算を中断して次のデマンドを待つ．これにより，データの生産者とデータの消費者の間での並列処理が可能となる．1回のデマンドに対して生成されるデータ量を以下，グラニュラリティと呼ぶ．要求駆動型評価と（2）の並列性を組み合わせる事により，限られた資源の中でのストリーム型の並列処理が実現される．

グラニュラリティは，メモリ使用量と関数間の並列性に影響を与える．グラニュラリティが小さい場合は，メモリ使用量は少ないが，デマンドの発行回数が増えるためにプロセッサ間の通信によるオーバヘッドが増大し，処理効率を悪化させる．一方，グラニュラリティが大きい場合は，デマンドの発行回数と通信によるオーバヘッドは減少する．しかし，使用するメモリ量が増大し，また，（2）の並列性が十分に引き出されない．（グラニュラリティが実引数のデータサイズに設定される場合には，（2）の並列性は引き出されない．）最適なグラニュラリティの設定は，データの内容，利用可能なメモリ量，プロセッサの処理能力，プロセッサ間のデータ通信速度に応じて決定される．

## 2．3　引数評価機構

要求駆動型評価を実現するためには，関数の引数評価においてcall-by-nameあるいはcall-by-needの引数受渡し機構が必要である．すなわち，関数本体の中でその引数に対する参照が起こった時点で引数が評価される．　call-by-nameでは，関数内で引数に対する参照が複数回あった場合（再参照）に，その引数は参照のたびに評価し直される．それに対しcall-by-needでは，引数は最初に参照されたときにだけ評価が行われ，それ以後の参照に対しては最初の評価結果の値が使用される．

call-by-nameでは，引数の評価結果が一度参照されるとその値を削除できる．従って，関数間で巨大な中間結果の受渡しが行われる場合にも，グラニュラリティを適切に設定することにより，限られたメモリ資源の中でそれらの関数を処理することが可能となる．しかしながら，同じ引数が複数回参照される場合には，その実引数を生成する関数を，参照の度に計算し直さなければならない．以後，この引数評価法を再計算方式と呼ぶ．一方，call-by-needでは，引数の評価結果が一度生成されると，すべての参照が終了するまでその評価結果を保持しておかなければならない．巨大な中間結果が生成される場合には，メモリ・オーバーフローを引き起こす可能性が高くなる．しかし引数の再評価は必要ない．以後，この引数評価法をcaching 方式と呼ぶ[6]．関数間で巨大な中間結果の受渡しを行う処理では，中間結果を保持するか否かは，処理時間とメモリ使用量に大きな影響を与えるので，この選択は重要である．我々は，両方の評価法を実現するための基本プリミティブを提供する．それぞれの引数について，再計算方式，あるいは caching 方式を指定することが可能である．すべての引数評価をcaching 方式で行うと，メモリ使用量が大変大きくなる可能性がある．また，すべての引数評価を再計算方式で行うと，処理時間を極端に増大させる可能性がある．本方式では，引数ごとに評価法を指定することにより，処理の内容と処理環境に柔軟に適応できる．

## 3．基本プリミティブとその使用法

本章では，関数計算における要求駆動型制御，並列処理，および2種類の引数評価法を実現する基本プリミティブを示す．また，例として，それらの基本プリミティブを用いて記述した関係データベース演算のプログラムを示す．

### 3．1　基本プリミティブ

channel(type, granularity, parameter_passing_method)

channelは，基本プリミティブnewによって生成される関数インスタンスの入力ストリームまたは出力ストリームを受け渡すためのチャネルを設定し，その識別子を return valueとして返す．チャネルは，ストリームデータの要素を格納するためのバッファに対応する．チャネルの属性として，type，granularity，parameter_passing_methodが指定される．typeは，このチャネルを介してやり取りをされるストリームの各要素のデータタイプを指定する．granularityは，1回のデマンドにより送られるデータ量を指定する．（granularity は，チャネルのバッファ容量に対応する．）parameter_passing_methodは，ストリームが再参照されたときに再計算を行う(call-by-name)か，cachingを行う（call-by-need)かを指定する．

new(f, pid, cid, parmeters)

fで示された関数のインスタンスを生成する． pid は，その関数インスタンスが配置されるプロセッサの識別子である．関数fが生成したストリームデータが出力されるチャネルはcid により指定される．関数fの入力引数（入力ストリームおよびその他の引数）はparmetersで指定される．生成された関数インスタンスは，その関数インスタンスが生成するストリームデータを消費する関数インスタンスからデマンドが送られてくるまで，関数計算を開始しない．

異なるプロセッサに配置された関数インスタンスは，要求駆動型制御に基づき，並列に処理される．同一プロセッサに複数の関数インスタンスが配置された場合は，その中から実行可能な関数インスタンスが選ばれて実行される．この場合，計算を開始した関数インスタンスは，あらかじめ指定されたグラニュラリティのデータを生成し終えたとき，または，到着している入力データを処理し終えたときに実行を中断する．このように，同一プロセッサに割り当てられた関数は，コルーチンのように実行が行われる．

pre-demand(cid)

cid で示された入力チャネルにストリームを出力する関数へデマンドを伝達する．pre-demandは，関数計算の駆動方式として要求駆動型評価を実現するための基本プリミティブの一つであり，ストリームの消費者側の関数が，生産者側の関数に対する最初のデマンドの先出しを行うために用いられる．デマンドの先出しを行うことにより，生産者側の関数は計算を先行的に行うことが可能となり，ストリームの消費者側の関数と生産者側の関数との間の並列性を引き出すことができる．

get1(cid)

cid で示された入力チャネルのバッファからストリームの1要素を取り出す．バッファが空のときは，そのチャネルに出力を行う関数へデマンドを伝達し，そのチャネルのバッファがデータで満たされるまで待つ．ストリームの各要素は，一度get1によりバッファから取り出されると，その時点で排除される．

get2(cid)

この基本プリミティブが用いられる場合は，cid で示された入力チャネルのバッファには，ダブルバッファリング機構が実現されることが前提となる．すなわち，入力バッファには二つの領域が存在し，一方の領域にストリームデータの書き込みが行われている間に，他方の領域にあるストリームデータを読み出すことができる．cid で示された入力チャネルの二つのバッファ領域の一方が空のときは，そこへストリームデータを格納するためのデマンドを発行する．それから，他方のバッファ領域を埋めているストリームデータを取り出しを行う．データを取り出している方のバッファが空となったときには，デマンドを発行したのち，他方のバッファからストリームデータの取り出しを開

始する．バッファにストリームデータが到着していない場合には，その到着を待つ．ストリームの各要素は，一度バッファから取り出されると排除される．

put1(d)
d で示されたストリームデータの１要素（その関数インスタンスのreturn valueの一部）を出力チャネルのバッファに関数のreturn valueとして書き出す．１回のデマンドに対してあらかじめ指定されたグラニュラリティのデータを生成し終えると，次のデマンドを待つ．

put2(d)
この基本プリミティブは，cid で示された出力チャネルのバッファにダブルバッファリング機構が実現されている場合に用いられる．チャネルのバッファには同時アクセス可能な２つの領域が存在し，一方の領域にput2によりd で示されるストリームの要素を出力している間に，消費者側の関数は他方の領域のストリームの要素をget2により取り出すことができる．

get1およびput1は，ストリームの消費者側の関数と生産者側の関数が同じプロセッサに配置された場合に用いられる．get2およびput2は，それらの２関数が異なるプロセッサに配置された場合に用いられ，それらの２関数間で，ストリーム型の並列処理が実現される．

関数本体内で他の関数インスタンスを生成し，関数適用を行う場合には，生成される関数インスタンスとの間に入出力のためのチャネルが指定される．これは，基本プリミティブchannelとnewを用いて行われる．この場合，生成された関数インスタンスへのストリームデータの送出は，基本プリミティブsend1(cid，d)あるいはsend2(cid，d)によって行われる．send1およびsend2は，各々put1およびput2と同様の動作をするが，書き込み用チャネルがcidによって明示的に指定される点が異なる．生成された関数の出力ストリームは，基本プリミティブreceive1(cid) あるいは receive2(cid)によって受け取る．receive1およびreceive2は，get1およびget2と同様の動作をする．

mark_end_of_stream()
ストリームデータの終端にストリームの生成の終了を示す識別子として"EOS"を書き込む．

check_end_of_stream(cid)
cidで示されたストリームの終端("EOS")を検出した場合にTRUE，そうでない場合にFALSEをreturn valueとして返す．

## ３．２　関数計算の実現

ある系を構成する各関数は，その関数が配置されるプロセッサの能力を最大限に引き出すように変換される．例えば，ノイマン型アーキテクチャのプロセッサ上に配置されて実行される関数は，基本プリミティブの発行を含むシーケンシャルなオブジェクトコードにコンパイルされる．また，例えば，データフロー型アーキテクチャのプロセッサ上に配置されて実行される関数は，単一代入則を守ったオブジェクトコードにコンパイルされる．以下では，関数が配置されるプロセッサはノイマン型アーキテクチャのプロセッサであり，関数は，シーケンシャルなオブジェクトコードにコンパイルされるものとする．以下で示すプログラムは，このオブジェクトコードをＣ言語[7]の記法を用いて抽象的に示したものである．

ここでは，例として，関係演算（関係データベース演算）の関数を基本プリミティブを用いて実行するプログラムを示す．関係演算を我々のアプローチに基づいて実現するには，選択演算や結合演算などの関係演算を関数として記述する．演算間でやり取りされる中間リレーションを，タプルを要素とするストリームデータとみなし，ストリームデータを扱う関数として関係演算を記述する．リレーションをストリームデータと見なした場合，関数間で受け渡されるデータの単位（グラニュラリティ）は，つぎの３種類が考えられる．

（１）リレーションレベル・グラニュラリティ
（２）ページレベル・グラニュラリティ
（３）タプルレベル・グラニュラリティ

グラニュラリティは，基本プリミティブchannelによりチャネルの属性として指定される．また，各グラニュラリティにおいて，ストリームの生成側の関数とその消費者側の関数の間で並行処理を実現する場合としない場合とが考えられる．また，同じストリームに対する複数回の参照がある場合は，ストリームの再計算(call-by-name)，あるいはストリームのcaching(call-by-need)の一方が選択される．この選択は，基本プリミティブchannelによりチャネルの属性として指定される．

以下では，基本プリミティブを用いて，要求駆動型評価に基づく関係演算処理を行うためのプログラムを示す．

（１）ページレベル・グラニュラリティ，ストリーム型並列性なし

この組み合わせでは，必要のない計算を排除できるだけでなく，限られた大きさのバッファ（ページサイズと等しい）を介して関数間のストリームの受渡しが可能となる．プログラム１に示される関数selection は，この関数の生成データを消費する関数からデマンドを受け取ると，基本プリミティブget1を用いて，この関数の入力データを生成する関数にデマンドを発行し，入力バッファにデータが格納されるのを待つ．データがバッファに格納されると，それらに対して選択演算が実行され，その結果が基本プリミティブput1により出力ストリームの要素として出力チャネルのバッファに格納される．この場合には，デマンドはバッファのサイズに応じて複数回発行されることになる．

プログラム１

```
define_function selection(relation, a)
stream relation;
item a;
{
   tuple tu;
   while (!check_end_of_stream(relation)) {
      tu = get1(relation);
      if (selection_test(tu, a))
         put1(tu);
```

```
    }
    mark_end_of_stream();
}
c1 = channel(tuples, INPUT_BUFFER_SIZE,
          RECOMPUTATION or CACHING);
/* specification of channel for output stream */
c2 = channel(tuples, OUTPUT_BUFFER_SIZE,
          RECOMPUTATION OR CACHING);
/* specification of channel for input stream */
new(selection, pid, c2, c1, a);
/* creation of function instance */
```

（2）ページレベル・グラニュラリティ，ストリーム型並列処理あり

要求駆動型評価では，基本的に引数の評価が必要となった時点でデータ生成側の関数へデマンドが発行される．要求駆動型評価のなかで，データの生成側の関数とデータの消費側の関数の間でストリーム型の並列処理を実現するためには，データの生成側の関数がデータの消費側の関数に対して，デマンドを先出しする必要がある．このようなストリーム型並列処理を実現するためには，チャネルに二つのバッファ領域が用意され，ダブルバッファリング機構が実現されなければならない．プログラム２に示される関数selection では，関数計算を開始する前に，ストリームの生産者側の関数にデマンドが基本プリミティブpre-demandにより先出しされる．生産者側の関数は，デマンドを受け取ると，関数計算を開始し，１ページ分の出力データを出力チャネルのバッファの一方の領域に格納する．この関数計算の間に，消費者側の関数はバッファの他方の領域内のストリームデータに対し演算を実行し，再び基本プリミティブget2によりデマンドを発行する．その後，先出しされたデマンドによって生成されたページの消費をget2によって開始する．このように，このプログラムの中で発行されるデマンドはそのget2が消費するページの生成要求ではなく，次のget2が消費するページの生成要求である．

プログラム２

```
define_function selection(relation, a)
stream relation;
item a;
{
   tuple tu;
   pre-demand(relation); /* pre-issuing a demand */
   while (!check_end_of_stream(relation)) {
      tu = get2(relation);
      if (selection_test(tu, a))
         put2(tu);
   }
   mark_end_of_stream();
}
c1 = channel(tuple, INPUT_BUFFER_SIZE,
          RECOMPUTATION or CACHING);
c2 = channel(tuple, OUTPUT_BUFFER_SIZE,
          RECOMPUTATION or CACHING);
new(selection, pid, c2, c1, a);
```

（3）ページレベル・グラニュラリティ，ストリーム型並列処理あり，再参照あり

上述の関数selectionは，入力ストリームの各データ要素の参照は１度だけであった．しかし，一つの入力ストリームを複数回参照する関数では，その入力ストリームを生成する関数を再計算するか，あるいは，一度生成した入力ストリームをcachingしておく必要がある．例えば関係演算では，結合演算などの２項関係演算がこれに対応する．２項関係演算では，一方の入力ストリームであるアウタリレーションの各ページに対して，他方の入力ストリームであるインナリレーションの全ページをつき合わせなければならない．この場合に，再計算を行うか，cachingを行うかは，チャネルの属性により指定される。再計算を行う場合には，限られたメモリ資源の中で関数計算を進めることができるが，cachingが行われる場合には入力ストリーム全体を保持する必要があるのでメモリ・オーバフローが引き起こされる可能性がある．結合演算の記述例をプログラム３に示す．

プログラム３

```
define_function join(relation_1, relation_2,pagesize
)
stream relation_1; /* stream of outer-relation */
stream relation_2; /* stream of inner-relation */
int pagesize;      /* size of outer-relation page */
{
   tuple in1[pagesize], in2;
   int i;
   pre-demand(relation_1);
   pre-demand(relation_2);
   while (!check_end_of_stream(relation_1)) {
      for (i = 0; (i < pagesize) &&
           !check_end_of_stream(relation_1); i++)
         in1[i] = get2(relation_1);
      sort(in1); /* sorting of outer-relation page*/
      while (!check_end_of_stream(relation_2)) {
         in2 = get2(relation_2);
         if (!check_end_of_stream(relation_1) &&
             check_end_of_stream(relation_2))
             /* pre-issuing a demand to request
                re-reference to the stream of
                inner-relation */
             pre-demand(relation_2);
         if (binary_search(in1, in2))
            put2(concatenate(in1, in2));
            /* comparing an inner-relation tuple
               (in2) with outer-relation tuples
               (in1) by binary search algorithm */
            /* concatenating tuples if joining
               condition is satisfied */
      }
   }
   mark_end_of_stream();
}
```

```
c1 = channel(tuple, BUFFER_SIZE_1,
          RECOMPUTATION or CACHING);
/* channel for input stream of outer-relation */
c2 = channel(tuple, BUFFER_SIZE_2,
          RECOMPUTATION or CACHING);
/* channel for input stream of inner-relation */
c3 = channel(tuple, OUTPUT_BUFFER_SIZE,
          RECOMPUTATION or CACHING);
/* channnel for output stream */
new(join, pid, c3, c1, c2, BUFFER_SIZE_1);
```

（4）リレーションレベル・グラニュラリティ

1回のデマンドに対して，生産者側の関数の完全なリダクションを行わせる場合，すなわち，生産者側の関数にすべての出力データの生成を完了させる場合には，グラニュラリティとしてCOMPLETE_REDUCTIONが設定される．この場合には，1回のデマンドに対して生成されるデータ量が予想できないので，限られたメモリ資源のなかで関数間のデータの転送は行えない．また，ストリームの生産者側の関数と消費者側の関数の間では，ストリーム型の並列性を引き出せない．プログラム2の関数 selectionにおいて，リレーションレベル・グラニュラリティを実現するために，基本プリミティブchannelによってグラニュラリティとしてCOMPLETE_REDUCTIONが次のように指定される．

```
c1 = channel(tuple, COMPLETE_REDUCTION,RECOMPUTATION
 or CACHING);
c2 = channel(tuple, COMPLETE_REDUCTION,RECOMPUTATION
 or CACHING);
new(selection, pid, c2, c1, a);
```

## 4．実験

3．1で示した基本プリミティブを1台のSUN-2 ワークステーション[13]上に実現し，基本プリミティブを用いて記述した関係演算のプログラムを実際に動作させた．並列プロセッサ環境は，1台のプロセッサで実行した場合の実行時データに基づいて評価される．1関係演算は3．2に示したように1関数として記述され，各関係演算は要求駆動型評価により計算される．ここでは，各種の要求駆動型評価を実現するプログラムを実行させた結果を示し，本方式の有効性について考察を行う．

### 4．1　実験環境

図1に示す3個の結合演算と4個の選択演算から成る問い合わせを対象とした．演算対象リレーションのタプル数，タプル長，選択演算の選択率および結合演算の結合率は表1のように設定した．

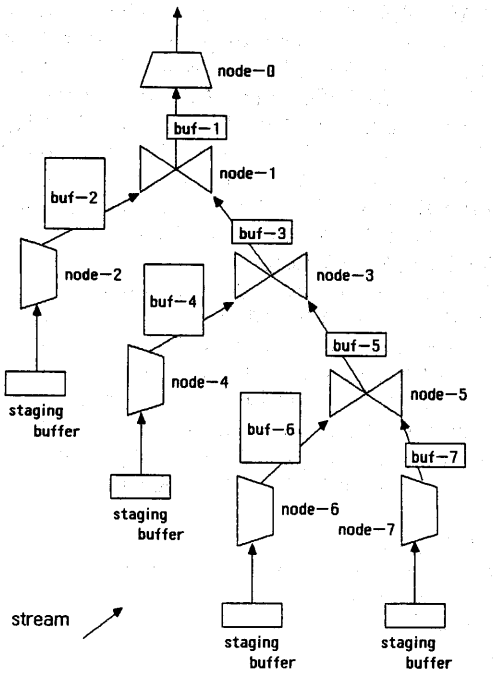
結合演算の場合には，アウタリレーションの各ページとインナリレーションの全ページのつき合わせを行う必要がある．従って，アウタリレーションのストリームを関数間でページレベル・グラニュラリティにより受渡しする場合には，インナリレーションを生成する関数の再計算，あるいは，そのインナリレーションの cachingが必要となる．その場合には，再計算により問い合わせ処理を行った．

ここでは，各関係演算を図1の中に示したように5台のプロセッサに配置することによって並列処理が行われる環境が，シミュレーションにより評価される．並列処理環境は，次のように仮定されている．

（1）プロセッサ間のストリーム・データおよびデマンドの転送は，排他的に行われるものとする．すなわち，2台のプロセッサ間で通信が行われている間は，他のプロセッサ間の通信は行えない．プロセッサ間のデータ転送速度は，16.6(msec/2KBytes)とする．

（2）1プロセッサに複数の関係演算が割り当てられた場合には，それらの実行順序はそのプロセッサ内のスケジューラによって制御される．

図1に示す問い合わせを，表2に示す4種類の環境(Environment-1, ..., -4)のもとで実行させた．各環境はは，グラニュラリティをチャネルの属性として設定し，基本プリミティブを3．2で述べたように用いることにより実現される．




```
(relation operation nodes)
 node-0: projection node without duplicate elimination
 node-1, node-3, node-5: join node
 node-2, node-4, node-6, node-7: selection node
(processor allocation)
 processor-0: node-0
 processor-1: node-1, node-2
 processor-2: node-3, node-4
 processor-3: node-5, node-6
 processor-4: node-7
(buffers)
 buf-1: buffer for storing operand pages of projection
 buf-2, buf-4, buf-6: buffers for storing outer-relation
                      pages of join operations
 buf-3, buf-5, buf-7: buffers for storing inner-relation
                      pages of join operations
```

図1　問い合わせ

表1　パラメータの設定

| | |
|---|---|
| tuple size | 64 bytes |
| operand attribute (integer value) | 4 bytes |
| cardinality (number of tuples) of each source relation | 10000 tuples |
| cardinality of each intermediate relation | 1000 tuples |
| selection selectivity factor(ssf) | 0.1 |
| join selectivity factor(jsf) | 0.001 |

(cardinality of intermediate result relation of the selection)
= ssf*(cardinality of source relation);
(cardinality of intermediate result relation of the join)
= jsf*(cardinality of outer relation)
*(cardinality of inner relation).

表2　問い合わせ処理環境

granularity (buffer size) for input stream: (number of tuples)

| Experimental Environment | selection operations | | | | join operations | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | node-1 | | node-3 | | node-5 | |
| | node-2 (staging buffer) | node-4 (staging buffer) | node-6 (staging buffer) | node-7 (staging buffer) | outer-relation (buf-2) | inner-relation (buf-3) | outer-relation (buf-4) | inner-relation (buf-5) | outer-relation (buf-6) | inner-relation (buf-7) |
| Environment-1 | page-level (1000) | page-level (1000) | page-level (1000) | page-level (1000) | relation-level (1000) | page-level without stream parallelism (100) | relation-level (1000) | page-level without stream parallelism (100) | relation-level (1000) | page-level without stream parallelism (100) |
| Environment-2 | page-level (1000) | page-level (1000) | page-level (1000) | page-level (1000) | relation-level (1000) | page-level with stream parallelism (100) | relation-level (1000) | page-level with stream parallelism (100) | relation-level (1000) | page-level with stream parallelism (100) |
| Environment-3 | page-level (1000) | page-level (1000) | page-level (1000) | page-level (1000) | relation-level (1000) | page-level with stream parallelism (100) | relation-level (1000) | page-level with stream parallelism (100) | page-level with stream parallelism (500) | page-level with stream parallelism (recomputation) (100) |
| Environment-4 | page-level (1000) | page-level (1000) | page-level (1000) | page-level (1000) | relation-level (1000) | relation-level (1000) | relarion-level (1000) | relation-level (1000) | relation-level (1000) | relation-level (1000) |

## 4．2　実験結果

図2～図5にEnvironment-1～Environment-4の各々の場合の各関係演算の動作状況をタイムチャートに示す．Environment-1とEnvironment-2を比較すると，Environment-2の場合に行われるデマンドの先出しにより，関係演算間の並列性が引き出され，処理効率が改善されることがわかる．また，Environment-2とEnvironment-4の比較において，Environment-2のように，ページレベルにグラニュラリティを設定することにより，高い並列性が引き出されることがわかる．Environment-3では，関係演算（図1におけるnode-7の選択演算）が再計算される．この様子がタイムチャートに表れている．この場合には，再計算によるオーバヘッドが引き起こされるが，メモリ資源が限られた環境，すなわち，アウタリレーションがバッファ内に入り切らない環境では，この計算機構は有効である．

アウタリレーションの受渡しをリレーションレベル・グラニュラリティに設定し，インナリレーションを受渡すグラニュラリティを変化させた場合の応答時間の変化を図6に示す．グラニュラリティが小さい場合には，デマンドが送出される回数，および，プロセッサ間でのストリーム・データの交信回数が増大するので，応答時間が長くなる．また，グラニュラリティが大きい場合には，関数間でのストリーム型並列処理の効果が引き出されないために，応答時間が長くなる．また，この場合には，メモリ使用量が増大するので，より大きなメモリ領域が用意されている必要がある．適切なグラニュラリティの設定は，プロセッサ間の通信速度およびストリームデータを格納するために利用できるメモリ量に依存する．

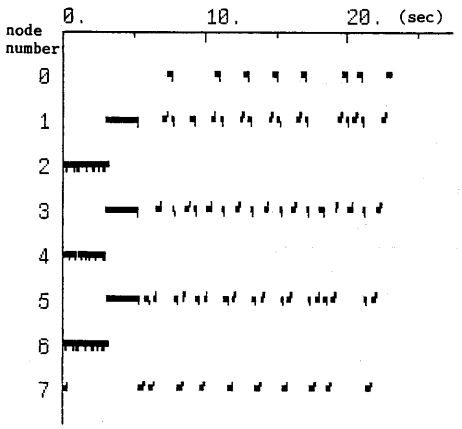



図2　タイムチャート(Environment-1)

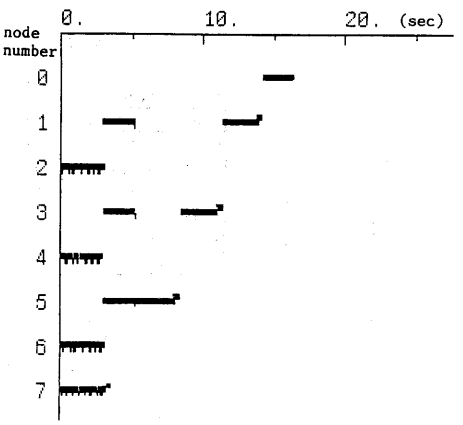



図5　タイムチャート(Environment-4)

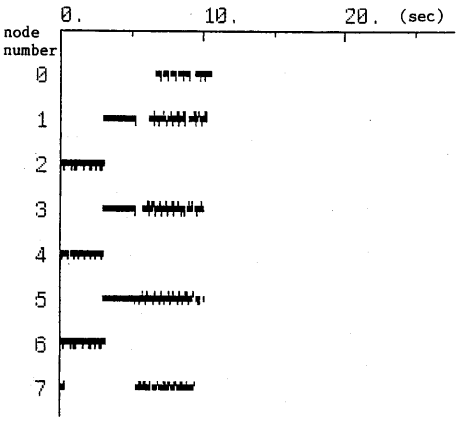
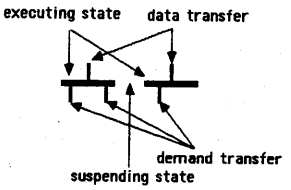



図3　タイムチャート(Environment-2)



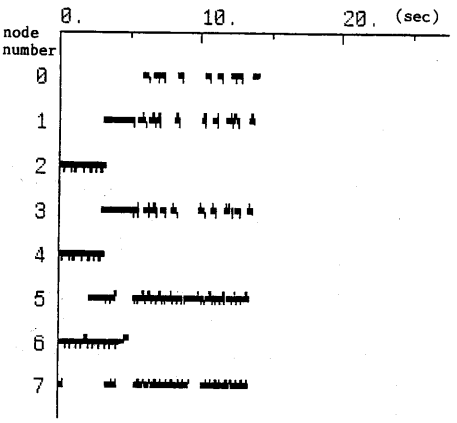




図4　タイムチャート(Environment-3)

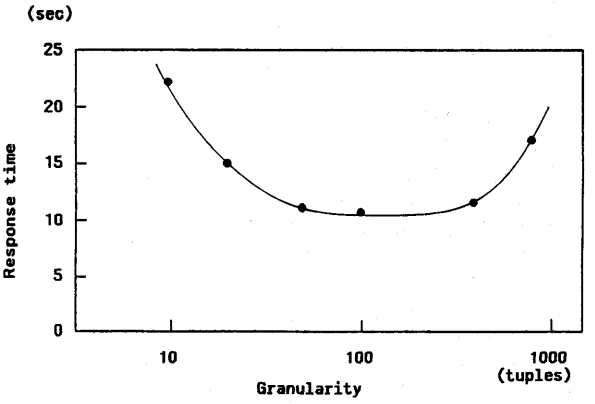



図6　グラニュラリティと応答時間

## 5．おわりに

関数性を守って記述したプログラムを、限られた計算機資源のなかで，関数間の並列性を引き出して処理する方式について述べた。本方式では，関数計算の駆動方式として要求駆動型制御を用いることにより、不用な計算を避け、限られた計算機資源のなかで関数間のデータの受渡しを行いながら並列処理を行うことが可能である。要求駆動型評価を実現する基本プリミティブを示し、その基本プリミティブを用いて並列処理を行なったときの実験結果を示した。

近年のＶＬＳＩ技術と，ローカル・エリア・ネットワーク（ＬＡＮ）技術の発展により，複数台の高性能マイクロプロセッサを，高速データ転送が可能なＬＡＮで結合した環境を容易に実現できるようになった．提案方式は，このような環境における実現に適している[11]．我々は，この方式を関係データベース処理や知識処理の実現に適用している[10]，[12]．

## 参考文献


[1] M. Amamiya and R. Hasegawa, "Dataflow computing and eager and lazy evaluations," New Generation Computing, vol. 2, no. 2, pp. 105-129, 1984.

[2] J. Backus, "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs," Comm. ACM, vol. 21, no. 8, pp. 613-641,Aug. 1978.

[3] E. F. Codd, "A relational model of data for large shared data banks," Comm. ACM, vol. 13, no. 6, pp.377-387, Jun. 1970.

[4] D. P. Friedman and D. S. Wise, "Aspects of applicative programming for parallel processing, "IEEE Trans. Comput., vol. C-27, pp. 289-296, Apr. 1978.

[5] G. Kahn and D. MacQueen, "Coroutines and networks of parallel processes," in Information Processing77: Proceedings of IFIP Congress 77, pp. 993-998, Aug. 1977.

[6] R. M. Keller and M. R. Sleep, "Applicative caching," in Proc. ACM Conf. Functional Programming Lang. Comput. Arch., pp. 131-140, 1981.

[7] B. W. Kernighan, D. M. Ritchie, "The C Programming Language", Prentice-Hall, Inc., 1978

[8] 清木，長谷川，雨宮，"先行・遅延評価機構を用いた関係演算処理方式," 情報処理学会論文誌，vol. 26, no. 4, pp. 685-695, 1985

[9] Y. Kiyoki, R. Hasegawa and M. Amamiya,"A stream-oriented parallel processing scheme for relational database operations," to appear in Proc. 1986 Int. Conf. Parallel Processing, 1986.

[10] Y. Kiyoki, K. Kato, T. Masuda, "A relational database machine based on functional programming concepts," to appear in Proc. ACM-IEEE Computer Society Fall Joint Computer Conf., 1986.

[11] Y. Kiyoki, K. Kato, T. Masuda, "A stream-oriented approach to distributed query processing in a local area network," Research Report, Institute of Information Sciences and Electronics, Univ. of Tsukuba, Mar. 1986.

[12] 清木，加藤，益田，"要求駆動型制御による関係演算パイプライン処理方式の実現法," 情報処理学会アドバンストデータベースシンポジウム予稿集，pp. 51-58, 1985

[13] "Programmers Reference Manual for the Sun Workstation", Sun Micro Systems, Inc., 1982

[14] P. C. Treleaven, D. R. Brownbridge and R. P. Hopkins, "Data-driven and demand-driven computer architecture," ACM Computing Surveys, vol.14, no. 1, Mar. 1982.

[15] S. R. Vegdahl, "A survey of proposed architectures for the execution of functional languages," IEEE Trans. Comput., vol. C-33, no. 12, pp. 1050-1071,Dec. 1984.