データベースシステム 89−19
（１９９２．７．２４）

関数型計算に基づくデータベースシステムの並列処理実行系の実現方式

佐藤 聡　　清木 康
筑波大学電子情報工学系

我々は、多様な応用分野のデータベースに対応する並列処理システムを提案している。このシステムはデータベース処理に関数型計算の概念を適用し、データベース処理におけるデータの流れをストリームとして扱うことにより、問い合わせに内在する並列性を抽出する。

本稿では、このシステムの並列実行系の設計について述べ、その実行系の実現方式を提示し、実行系を用いて行った実験の結果を示す。

The Implementation Method of Parallel Processing in a Database System based on Functional Computations

Akira SATO and Yasushi KIYOKI
Institute of Information Sciences and Electronics
University of Tsukuba, Tsukuba, Ibaraki 305, JAPAN
Phone:0298 53 5163 Fax:0298 53 5206
Email: akira@softlab.is.tsukuba.ac.jp , kiyoki@is.tsukuba.ac.jp

We have presented a parallel processing system for supporting a wide variety of database applications. The main feature of this system is the functional programming concept applied to parallel processing for databases. This system exploits parallelism inherent in a query by performing functional computations incorporating streams.

In this paper, we present the design and the implementation method of our parallel processing scheme, and several experimental results of query processing.

1 はじめに

近年、データベースは事務処理分野だけでなく、科学技術分野でも応用されるようになってきている。このようなデータベースの応用分野の多様化に対応するための方法として、それぞれの応用分野に適したデータベース演算やデータ構造の定義を行なうシステムを提供することが考えられる。

また、データベースの普及とともに、データベースに格納されるデータの量は大規模化しており、データベース処理の高速化が不可欠な課題となっている。近年、汎用の並列・分散処理環境が広く普及し、それらの環境が容易に利用できるようになっており、データベースの処理の高速化の手法として、それらの汎用の並列・分散環境を利用することが有効であると考えられる。

このような背景に基づき、我々は、多様な応用分野のデータベースに対応する並列処理システムを提案している[1][2]。このシステムの特徴は、データベースの並列処理を関数型計算の枠組の中で実現する点にある。このシステムでは、データ構造、および、データベース演算が関数型計算の枠組のなかで宣言的に記述され、問い合わせの最適化が行なわれ、さらに、関数型計算により並列性が抽出される。本システムでは、データベース設計者はデータベース演算を関数として定義する。データベース使用者はこの定義された関数を組み合わせることによって問い合わせを発行する。本システムは、データベース処理におけるデータの流れをストリームとして扱うことにより、データベース演算間に内在する並列性を抽出することができる。

本稿では、本システムの並列処理方式として提案しているストリーム指向並列処理方式について概観し、さらに、本システムにおける並列処理実行系の設計について述べる。また、その並列処理実行系の実現方法を提示し、その実行系を用いて行なった実験について述べる。

2 ストリーム指向型並列処理方式

我々は、データベース処理において、任意のデータベース演算間に内在する並列性を抽出する方式として、関数型計算に基づくストリーム指向並列処理方式を提案している[1][2]。この方式は、データベース処理におけるデータベース、および、問い合わせの中間結果をストリームとして扱い、関数型計算の枠組みの中で定義された任意のデータベース演算を、要求駆動型評価によって並列に処理する方式である。

この方式を実現する並列処理システムにおいて、データベース設計者はデータベース演算を関数として定義する。また、問い合わせ処理は、定義されたデータベース演算を組み合わせることにより構成される式を評価することによって行なわれる。

要求駆動型評価でストリーム型の引数を評価する場合、1回のデマンドあたりの関数間のデータの転送量（以下、この量をグラニュラリティとよぶ）を定めるができるので消費者関数のストリームの計算の進行に応じて、生成者関数の計算の進行を制御できるという利点がある。したがって、限られたメモリ資源の中で、データベースのような大量のデータを並列に処理する場合にこの評価方式は有効である。

この評価では、デマンド伝達のオーバーヘッドが処理効率の劣化を引き起こす場合がある。グラニュラリティを小さく設定した場合、メモリの使用量は少なくて済むが、デマンド伝達の回数が増えるため、オーバーヘッドが増大し、処理速度に影響を与える。しかし、データベース処理においては、グラニュラリティは比較的大きく設定できるので、デマンド伝達のオーバーヘッドはデータの転送と比較して問題とならない。グラニュラリティの設定による問合せの最適化の方法については文献[3] [4]に示している。

この方式により抽出される並列性には、つぎの2種類がある。

1. 関数の引数を並列に評価することによる並列性
2. ストリーム型並列性

3 並列実行処理系の環境

我々は、ストリーム指向型並列処理方式に基づいた計算を行う抽象マシンの基本命令セットとして、プリミティブ・セットを設定した[2]。プリミティブ・セットとは、データベースを対象とする演算を関数型計算の枠組みの中で並列に処理するための、ソフトウェア・アーキテクチャの命令セットである。本システムの並列処理実行系は、そのプリミティブ・セットを解釈・実行する。すなわち、本実行系は、プリミティブ・セットを基本命令セットとして持つ抽象マシンであり、本実行系を実現することは、この抽象マシンの解釈実行系を実現することに対応する。

3.1 プリミティブ・セット

プリミティブ・セットは、その操作内容によって、次の4種類に分類される。

1. 関数インスタンス、および、チャネルの生成

   `new(function,site,parameter-list)`

   関数インスタンスの生成。

   ```
   channel(element-size,granularity,
           stream-parallelism,cashing-site,
           producer-site,consumer-site-list)
   ```

   チャネルの生成。

2. ストリーム要素、および、デマンドの送受信、ストリームの再参照

   `pre_demand(channel)`

   先行的なデマンドの発行。

   `get(channel),receive(channel)`

   ストリーム要素の受信。

`put(channel,element),send(channel,element)`

ストリーム要素の送信。

`mark_eos(channel)`

ストリームの生成終了。

`rewind(channel)`

ストリームの再参照。

3. チャネルの非決定的選択

`select(channel-list)`

非決定的なストリーム要素やデマンドを送受するチャネルの選択。

`enable(channel)`

引数で指定されたチャネルを、select プリミティブにおいて選択可能な状態にする。

`disable(channel)`

引数で指定されたチャネルを、select プリミティブにおいて選択不可能な状態にする。

4. 複雑なデータの定義、格納、参照

本システムでは、複雑なデータ構造を表現するために識別子 (identifier) 型（以下、id 型とよぶ）を導入している [2]。

`define-type(name,type-description)`

データ型の定義。

`store(value,table)`

実データの格納。

`fetch(identifier,table)`

id 型による実データの参照。

3.2 実現を行なうハードウエア環境

本実行系を実現するハードウェア環境として、以下のものを設定している。

1. 汎用のシングル・プロセッサをデータグラム・サービスを提供する汎用の高速のローカル・エリア・ネットワークによって結合した環境（以下、ネットワーク型並列処理環境という）。
2. 汎用の共有メモリ型マルチ・プロセッサを用いた環境（以下、共有メモリ型並列処理環境という）。
3. 1 と 2 を統合した環境（以下、統合環境という）。

4 並列処理実行系の設計

4.1 並列処理実行系の構成

本システムの並列処理実行系は次の 3 機能単位から構成される。

4.1.1 関数インスタンス

本システムでは、データベースの基本演算をすべて関数として記述し、関数を単位として並列性を抽出する。実行時に起動された関数を関数インスタンスとよび、並列処理実行時には、関数インスタンスを単位として並列に実行される。つまり、関数インスタンス内部は逐次的に実行されるが、関数インスタンス間において並列性が抽出される。

4.1.2 チャネル

関数インスタンス間において、デマンドおよびストリーム要素を受け渡す機構をチャネルと呼ぶ。チャネルの主な機能は、サイト間およびサイト内通信である。ここで、サイトとは、ネットワークに結合されているシングル・プロセッサ、もしくは、共有メモリ型マルチ・プロセッサである。本実行系では、問い合わせを並列に実行する際に、各関数インスタンスを複数のサイトに割り付ける。よって、デマンドおよびストリーム要素の送受を行う関数インスタンス群は、同じサイトに配置される場合と、異なるサイトに配置される場合がある。従って、関数インスタンス間のデマンドやストリーム要素の送受には、サイト間通信（ネットワーク通信）の機能、および、サイト内通信の機能が必要となる。このようなサイト間通信機能、および、サイト内通信機能は、チャネルを介して提供される。すべての関数インスタンスは、デマンドやストリーム要素の送受をチャネルを介して行うことにより、サイト間通信かサイト内通信かを意識する必要がなくなる。すなわち、ストリーム要素の送受の手続きが、関数インスタンスに対して透明となる。

4.1.3 カーネル

カーネルは、次のような機能を実現する。

1. プリミティブの解釈・実行

すべてのプリミティブの解釈実行は、カーネルによって行われる。プリミティブ・セットの項で述べたように、プリミティブは、4 種類に分類され、それぞれの解釈実行はモジュール化されている。

2. ネットワーク通信

チャネルの項で述べたように、チャネルを介してストリーム要素やデマンドの送受を行なう関数インスタンス群が異なるサイトに配置される場合、デマンドやストリーム要素の転送にネットワーク通信が必要である。この場合、カーネルは、関数インスタンスからチャネルに向けて出された通信要求をネットワークに転送する。また、カーネルは、他のサイトからネットワークを介して送られてきた通信要求をチャネルを介して関数インスタンスに転送する。

また、関数インスタンスやチャネルの生成を他のサイトに対して要求する場合、あるいは、他のサイトからの関数インスタンスやチャネルの生成を要求され

る場合に応じるために、ネットワーク通信の機能が必要である。本システムでは、各サイトが協調的に計算を進めるため、関数インスタンスやチャネルの生成要求は非同期に行なわれる。よって、データベース演算における実行の並列性を失わないために、ネットワーク通信の受信機能を専門におこなうプロセスがカーネルによって生成される。本稿では、このプロセスを通信エージェントとよぶ。この通信エージェントは、ネットワーク通信を介して送られてくる関数インスタンスやチャネルの生成要求の受信だけではなく、ネットワーク通信を介して送られてくるデマンドやストリーム要素の受信を行なう。

3. プロセスのスケジューリング

   関数インスタンスの項で述べたとおり、並列処理実行系では、問い合わせを関数インスタンスを単位として並列に実行する。カーネルは、その関数インスタンスをプロセスとして扱う。また、ネットワーク通信の項で述べたように、カーネルは通信エージェントとよぶプロセスを生成する。カーネルは、これらプロセスのスケジューリングを行う。

4.2 関数インスタンスの実行スケジューリング

要求駆動型評価の原則に従い、1グラニュラリティ分のストリーム要素を生成した生成者関数インスタンスは、次回のデマンドを待って実行を中断する。同様に、1グラニュラリティ分のストリーム要素を消費し尽くした消費者関数インスタンスは、次のストリーム要素群が到着するまで実行を中断する。関数インスタンスには、次の5状態がある。

1. 実行可能状態

   この状態は、関数インスタンスが現在実行可能となっていることを表す。

2. 実行状態

   この状態は、関数インスタンスが現在プロセッサ上で実行中であることを表す。

3. デマンド待機状態

   この状態は、関数インスタンスがデマンドの到着を待っていることを表す。

4. データ待機状態

   この状態は、関数インスタンスがストリーム要素の到着を待っていることを表す。

5. 終了状態

   この状態は、関数インスタンスが計算を終了したことを表す。

4.3 チャネルの設計

チャネルは、以下のような機能を実現するものでなければならない。

- チャネルの入力用インタフェースに接続された生成者関数インスタンスから、チャネルの出力用インタフェースに接続された消費者関数インスタンスへ、順序を維持しながら、各ストリーム要素を確実に転送する。
- チャネルの出力用インタフェースは、入力用インタフェースへ確実なデマンドの転送を行なう。

本システムでは、関数インスタンスはプリミティブnewにより、指定されたのサイト上に割り付けられる。それらの複数サイトに配置される関数インスタンス間のストリーム要素やデマンドの転送は、チャネルを介して行われる。チャネルは次のように分類される。

1. サイト内通信を行なうチャネル。

   通信を行なう関数インスタンス群が、同一のサイト上に配置されている場合のチャネル。この場合、チャネルは主記憶上のメモリ・コピーにより実現できるので、デマンド、および、ストリーム要素の転送の信頼性がある。

2. サイト間通信を行なうチャネル。

   通信を行なう関数インスタンス群が、それぞれ異なるサイト上に配置されている場合のチャネル。この場合、チャネルはサイト間を結合しているネットワーク通信のメッセージ転送により実現される。並列処理実行系において、デマンド、および、ストリーム要素の転送の信頼性を保証する必要がある。

本稿で提案するチャネル設計は、層化の手法を適用した。その利点は次にあげるとおりである。

- デマンド、および、ストリーム要素の転送の信頼性を保証する層と、保証された転送を利用する層とに分けることにより、各々の設計や実現が行いやすく、また、理解しやすくなる。
- 保証された転送を利用する層は、2つに分類したチャネルで共用することができるため、それらの設計を一括して行なうことができる。
- 層化することにより、本処理系内の他のモジュールと共有することができる。たとえば、信頼性を保証する部分は、関数インスタンスおよびチャネルの生成の部分と共有できる。

チャネルの層化の基準を以下のように設定した。

- 層5

  この層では、関数型計算に基づいて、関数インスタンスがプリミティブの発行を行なう。

- 層4

  この層は、2つに分類したチャネルの共通部分である。

- 層3

  この層では、データ通信の信頼性を保証する。

- 層2

  この層では、上向き多重化(upward multiplexing)を行なう。上向き多重化とは、複数の上位層の接続点を1つの下位層の接続点に対応づけることである。

- 層1

  この層は、想定してあるデータグラムサービスを提供する。

また、層間のインタフェースは次の様に設計した。

- 層5－4間のインタフェース

  このインタフェースは、チャネルを介してストリーム要素、あるいは、デマンドの転送を行なうプリミティブに対応する。それらのプリミティブは以下の通りである。

  `pre_demand()`

  先行的なデマンドの発行

  `get(elm),recieve(elm)`

  ストリーム要素の受信

  `put(elm),send(elm)`

  ストリーム要素の送信

  `mark_eos()`

  ストリームの生成終了

  `rewind()`

  ストリームの再参照

- 層4－3間のインタフェース

  このインタフェースは、次の3機能である。

  `send_page(page)`

  1ページの転送を行なう。

  `request_page(page)`

  1ページ分の生成要求(デマンド)を行なう。

  `reinit()`

  ストリーム再参照のための再初期化を行なう。

- 層2－3間のインタフェース

  このインタフェースは、次の2機能である。

  `reply(frag,p_num,f_num)`

  第`p_num`番目のページの第`f_num`番目のフラグメント`frag`の転送を行なう。

  `request(n,w,list)`

  第`n`番目のページから`w`ページ分の生成要求を行なう。

  ただし、`list`は消費者側において、第`n`番目のページの中で未受信のフラグメント番号のリストを表す。

- 層1－2間のインタフェース

  ネットワーク通信として、次の2種類のインタフェースが提供されていることを仮定している。

  `sendto(site,message)`

  ネットワークへメッセージを送信する。

  `recvfrom(site,message)`

  ネットワークからメッセージを受信する。

本稿で提案する層間のインタフェースの特徴は、次のようにまとめられる。

- 層4－3間のインタフェースにおいて、消費者側と生成者側とで対となるインタフェースを同じ仕様に設計した。サイト内通信を行なうチャネルは、その対となるインタフェースを結び付けることにより実現する。
- 層3－2間のインタフェースは、べき等(idempotent)である。べき等とは、同じ操作を複数回実行しても、その操作を1回だけ実行した場合と全く同じ結果を与え、操作に影響する様な副作用を起こさないような性質をいう。

次節以降において、今回設計を行なった層4、3、および、2における動作アルゴリズムを示す。アルゴリズムの表記のために、以下のような手続きをもっているPASCAL風の言語を用いる。

dequeue (que)	待ち行列queから1つ要素を取り出し、その要素を値として返す。
enqueue (elm,que)	待ち行列queに要素elmを加える。
mod (x,y)	xをyで割った余りを返す。
sleep (process)	プロセスprocessをデータ待機状態またはデマンド待機状態に遷移させる。
wakeup (process)	プロセスprocessを実行可能状態に遷移させる。
page_to_frag (page,n)	ページpageをフラグメントに分け、その第n番目のフラグメントを値として返す。

4.3.1 層4

この層は、2つに分類されたチャネルの共通部分を実現する。この層は通信単位を変換するサービスを提供する。

最上位層（層5）では、関数インスタンスどうしがチャネルを介してストリーム要素を単位として、通信を行っている。一方、ストリーム指向型並列処理方式では、要求駆動評価に基づいて、1グラニュラリティを単位として、ストリーム要素群の転送や、デマンドの発行が行なわれる。ここでは、このグラニュラリティ分のストリーム要素のことをページとよび、ページを格納する領域をバッファとよぶ。この層では、上位の層で送受されるストリーム要素と、下位の層で送受されるページとの通信単位の変換を行なう。また、通信単位の変換の際に、要求駆動型評価に基づいて次のページの生成のデマンドの発行も行なう。

この層は主に次のような状態情報を持っている。

`current` ：バッファをさす変数

現在参照しているバッファをさす。変数の値が NULL ならば、現在参照しているバッファが存在しないことを表す。

`full` ：バッファを要素とする待ち行列

この待ち行列の各要素はストリーム要素で満たされているバッファである。

`free` ：バッファを要素とする待ち行列

この待ち行列の各要素はストリーム要素がないバッファである。

`stat` ：チャネルの状態を表す変数

最終のストリーム要素が転送された場合はこの変数が終了状態となる。

`owner` ：プロセスを指す変数

このチャネルを介してストリーム要素を送出、または、受信する1関数インスタンスを指す。

4.3.1.1 上位層とのインタフェース 上位層とのインタフェースのうち、put,get を例にとって、この層にアクセスが行なわれた場合の動作アルゴリズムを示す。

get(elm)
仕様
ストリーム要素をバッファから受けとる。
出力
elm ストリーム要素が格納される変数
アルゴリズム

```
if current = NULL then
  if stat = 終了状態 then
    return ( EOS )
  endif
  request_page( dequeue ( free ) ) ;
  while full が空である do
    sleep (owner) /*sleep itself*/
  endwhile
  current := dequeue ( full );
  if current がさすバッファが最終ページである then
    stat := 終了状態 ;
  endif
endif
current がさすバッファから elm へ
ストリーム要素を取り出す;
if current がさすバッファが空である then
  enqueue ( current , free );
  current := NULL ;
endif
```

put(elm)
仕様
ストリーム要素をバッファに書き込む。
入力
elm ストリーム要素が格納されている変数
アルゴリズム

```
if current ≠ NULL then
  if current がさすバッファが満たされている then
    send_page( current );
    current := NULL;
  endif
endif
if current = NULL then
  while free が空である do
    sleep (owner) /*sleep itself*/
  endwhile
  current := dequeue ( free );
endif
elm から current がさすバッファに
ストリーム要素を書き込む ;
```

4.3.1.2 下位層とのインタフェース 下位層とのインタフェースのうち、send_page, requ_page を例にとって、この層にアクセスが行なわれた場合の動作アルゴリズムについて示す。

send_page(page)
仕様
ページ転送を行なう
入力
page ストリーム要素で満たされたページ
アルゴリズム

```
enqueue ( page , full );
wakeup ( owner ) ;
```

request_page(page)
仕様
ページ転送の要求を出す。

入力

page 空ページ

アルゴリズム

```
if stat = 終了状態 then
   無視する;
else
   enqueue ( page , free ) ;
   wakeup ( owner ) ;
endif
```

4.3.2 層3

この層は、上位層に対して信頼性を保証するサービスを提供する。

層2で提供されるサービスは、上向き多重化されたデータグラムサービスであるため、そのサービスを使ってメッセージ交換を行うと、メッセージの紛失や重複が起きる可能性がある。そのため、この層では、メッセージの紛失や重複に対する誤りを訂正する。

また、層2で提供されるサービスには、メッセージの大きさに上限がある。したがって、この層では、上位の層で送受されるページを、層2のサービスが利用できるように、小さなフラグメントに分ける。

この層は主に次のような状態情報を持っている。はじめに、消費者側と生成者側に共通な状態情報を示す。

free ：バッファを要素とする待ち行列

この待ち行列の各要素は、空のバッファである。

stat ：状態変数

このデータ構造を使用するチャネルの状態を表す。

npage ：定数

チャネルがもっているバッファの数を表し、チャネルが生成される時に設定される。ダブルバッファリング機構が設定された場合は2、そうでない場合は1である。

pages[n] バッファをさす配列。

受信待ち、または、送達確認待ちのバッファをさす。配列の大きさは、npage である。

lists[n] ：順序番号を要素とする待ち行列の配列

現在、受信を待っているフラグメントの順序番号を要素とする待ち行列の配列である。すなわち、pages[i] のページのフラグメントを受信した場合、そのフラグメントの順序番号が lists[i] から取り除かれる。

nfrag ：定数

1バッファあたりのフラグメントの数を表し、チャネルが生成される時に設定される。

次に、消費者側と生成者側とで異なる状態情報を示す。

- 消費者側で使用する変数

  nxt 次に受信すべきページの順序番号を表す。

  wnd 受信可能なページ数を表す。

- 生成者側で使用する変数

  una 送達確認待ちをしているページの最小順序番号を表す。

  pro 生成し終えているページの最大順序番号を表す。

  req 生成要求をしているページの最大順序番号を表す。

4.3.2.1 上位層とのインタフェース 上位層からインタフェース request_page, send_page を用いて、この層にアクセスが行なわれた場合の動作アルゴリズムを説明する。

request_page(page)

仕様

ページ転送の要求を出す。

入力

page 空ページ

アルゴリズム

```
pages[ mod (( nxt + wnd ),npage )] = page ;
i = 0 ;
while i < nfrag do
   enqueue ( i, lists[mod ((nxt + wnd ),npage )]) ;
endwhile
wnd を 1 増やす;
request( nxt ,wnd ,lists[mod (nxt ,npage )] );
```

send_page(page)

仕様

ページ転送を行なう。

入力

page ストリーム要素で満たされたページ

アルゴリズム

```
if page が最終ページである then
   stat を終了状態にする;
endif
pro を 1 増やす;
pages[mod (pro ,npage )] = page;
i = 0 ;
while i < nfrag do
   reply( page_to_frag (page,i) , pro ,i) ;
   i を 1 増やす;
endwhile
```

4.3.2.2 下位層とのインタフェース 下位層とのインタフェース request, reply を用いて、この層にアクセスされた場合の動作アルゴリズムについて説明する。

reply(frag,p_num,f_num)

仕様

p_num 番目のページの f_num 番目のフラグメント転送。

入力

frag 受信したフラグメント

p_num 受信したフラグメントのページ順序番号

f_num 受信したフラグメントのフラグメント順序番号

アルゴリズム

```
if 既に受信済みのデータである then
  無視する;
else
  page  = pages[mod (p_num,npage )];
  フラグメントを page の適切な位置へコピーする;
  rmqueue ( f_num , lists[mod (p_num,npage )] );
  if p_num = nxt then
    while page のすべてのフラグメントを受信した do
      if page が最終ページである then
        stat を終了状態にする;
      endif
      nxt を 1 増やし、 wnd を 1 減らす;
      send_page( page  ) ;
      page  = pages[mod (nxt ,npage )] ;
    endwhile
  endif
endif
```

request(n,w,list)

仕様

第 n 番目のページから w ページ分の転送 (生成) 要求。

入力

n 転送要求されているページの最小順序番号

w 受信可能なページ数

list 消費者側における n 番目のページの未受信フラグメント番号のリスト

アルゴリズム

```
while n <  una do
  enqueue (pages[mod (una ,npage )],free );
  una を 1 増やす;
endwhile while n+w-1 >  req do
  requ_page( dequeue ( free ) ) ;
  req を 1 増やす;
endwhile
if pages[mod (n,npage )] がある then
  while list が空でない do
    num = dequeue ( list );
    reply(page_to_frag (pages[mod (n,npage )],i),n,num);
  endwhile
endif
```

これら下位層との 2 インタフェースはべき等なサービスを提供している。これにより、メッセージの重複に対する回復処理を行なっている。

4.3.2.3 エラー回復処理 消費者側に以下のタイムアウト処理を加えることにより、メッセージの紛失に対する回復処理が可能となる。

Timeout()

仕様

ネットワーク通信において、データが紛失した場合のエラー回復処理。

アルゴリズム

```
request( nxt ,wnd ,lists[mod (nxt ,npage )]);
```

4.3.3 層 2

この層では、上向きの多重化を行なう。一般に、サイト間通信を行なうチャネルは複数存在する。1 つのサイトには、物理的に 1 つのネットワークのみが接続されていると仮定すると、サイト間通信を行なうチャネルとネットワークの間で上向き多重化を行なう必要がある。また、この層のサービスは、チャネルを介した関数インスタンス間のデマンド、および、ストリーム要素の転送以外にも、チャネルや関数インスタンスの生成要求の転送にも使用される。

送信時には、受けとったメッセージの種別 (ページ、デマンド、チャネル生成、関数インスタンス生成) 、宛先アドレス (どのサイト上のどのチャネル) といったヘッダをつけ、下位層とのインタフェース sendto を用いてメッセージを送り出す。

前述したとおり、受信は、通信エージェントとよばれるプロセスが代行する。通信エージェントは、1 つのサイトに 1 つ生成される。そして、そのサイト内で他のサイトからの受信処理をすべて行う。通信エージェントは、下位層とのインタフェース recvfrom を用いて、メッセージを受け取り、そのメッセージの種別、宛先アドレスを、ヘッダ部分から解釈し、適切な層 3 との接続点を判断し、その接続点に対して、データ本体を送る。

5 実現

並列処理実行系の設計に基づき、ネットワーク型並列処理環境の上で実現を行なった。実現を行なったネットワーク型並列処理環境は、5 台の SUN 社製 SparcStation1 (Sun OS Release 4.1.1) を Ethernet によって結合したハードウエア環境である。

本実行系は、関数インスタンスや通信エージェントを並

列処理の単位としてのプロセスとして扱う。この場合のプロセスとは、CPU 割り当ての単位であり、CPU 以外の資源割り当てや保護の単位ではない。本実行系のプロセスは、軽量プロセス (lightweight process) を用いて実現した。すなわち、本実行系のプロセスは、UNIX に代表されるオペレーティング・システム (以下、OS という) において用いられる概念であるプロセスを用いて実現したものではない。

従来、OS において、プロセスは資源割り当て、および、保護の単位として扱われている。本実行系のプロセスは、次のような理由により、軽量プロセスとして実現した。

- 軽量プロセスの生成、および、消滅については、OS のカーネル・コールを用いずに、本実行系の中で独立に行なえる。
- 各軽量プロセスのスケジューリングは、OS が行なうのではなく、本実行系の中で独立に行なうことができる。すなわち、本システムのデータベース処理に適したスケジューリングを行なうことができる。
- 軽量プロセスのコンテキスト切替えは、OS のカーネル・コールを介さずに行なえるので、そのオーバヘッドが小さい。これは、本実行系のように、コンテキスト切替えが頻繁に起こるシステムにおいて有効である。
- 軽量プロセスのメモリ空間は、OS のカーネル・コールを用いることなく共有することができる。これにより、プロセス間の同期・通信のオーバヘッドが小さくなる。

今回は、文献 [5] において提案した軽量プロセスを用いた。

層 1 は、データグラムサービスを提供するために、UDP (Internet User Datagram Protocol) を用いた。

6 実験

上述した方法によって、本システムの並列処理実行系の実現を行ない、並列性の抽出について、その実行系の性能を評価するための実験を行なった。

近年、ネットワークの通信速度に対して、CPU の処理速度の方が相対的に速くなってきている。ネットワーク型並列処理環境では、プロセッサ数を増やすことにより、ネットワーク上の通信量が増えることになる。よって、ネットワークの転送時間が全体の処理時間に大きく影響することが考えられる。

今回の評価実験では、次の 2 点に着目した。

1. プロセッサ数と全体の処理時間
2. ネットワークの転送時間に対するプロセッサ内の処理時間の割合と全体の処理時間の変化

6.1 実験環境

実験に用いたデータベース演算は次の通りである。

read 二次記憶上のデータベース $InputDB$ からデータを読み込んで、ストリーム要素として出力するデータベース演算である。データベース $InputDB$ は、1 タプル中に複数の整数値を持っており、1 タプルのデータ長が 128 バイト、データベース・サイズは 10000 タプルとした。このデータベースの各タプルは乱数を用いて自動生成した。

$func_n$ これは、入力チャネルの各ストリーム要素を読み込み、そのストリーム要素中にある id 型データと、二次記憶上のデータベース $InterDB_n$ 中の実データを取り出すデータベース演算である。出力チャネルに書き込む各ストリーム要素のデータ長、および、ストリーム要素数は入力チャネルのそれと同じとする。データベース $InterDB_n$ の実データは、データベース $InputDB$ と同様に、1 タプル中に複数の整数値を持っており、1 タプルのデータ長が 128 バイト、データベース全体は 10000 タプルからなっている。このデータベースも、乱数を用いて自動生成した。

この演算は、他に引数 num を持つ。この演算では、入力チャネルの各ストリーム要素について、データベース $InterDB_n$ に num 回アクセスする。2 回目以降のデータベースのアクセスには、直前にデータベース $InterDB_n$ から読み込んだデータ中の id 型データをもとにデータベース $InterDB_n$ に再帰的にアクセスする。これによって、1 タプルあたりの計算量 (計算深度) を変えることができる。本実験では、この計算深度を変化させることによって、ネットワークの転送時間に対するプロセッサでの処理時間の割合を変化させる。

write これは、入力チャネルの各ストリーム要素を読み込み、二次記憶上の $OutputFile$ へ書き込む演算である。

次に、実験に用いた問い合わせについて説明する。
問い合わせは以下のような式になる。

$$write(func_1(func_2(func_3(read()))))$$

演算 *write*,*read* にはそれぞれ 1 つずつのサイトを割り当て、残り 3 つの演算に対して、1 台から 3 台のサイトを割り当てる。

1 台の場合は、3 つの演算が 1 つのサイトに配置される。2 台の場合は、$func_1$ と、$func_2$,$func_3$ が各々別のサイトに配置される。3 台の場合は、各演算がそれぞれ異なるサイトに配置される。

また、$func_n$ に与える引数 num は、次のように設定した。まず、num を 1 にした場合の $func_n$ のみの実行時間を測定した。そして、$func_n$ のみの実行時間が、num を 1 とした場合の時間の 2,3,4,5 倍になるように、それぞれ

num の値を定め、実験を行なった。演算間でデータの転送量は同一なので、プロセッサ内の処理時間とネットワークの転送時間の割合が変わることになる。

また、各チャネルは、生成時にダブル・バッファリング機構を設定した。また、チャネルのバッファの大きさはすべて同じになるように設定し、その値を、2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096 タプルが格納できる大きさとした。

6.2 実験結果、および、考察

前節のようにして、実験を行なった。サイトの総数 (*write*, *read* に割り当てたサイトも含む) と実行時間との関係 (バッファの大きさは 64 タプル) を図 1 に示し、バッファの大きさと実行時との関係 (サイトの総数は 4) との関係を図 2 に示す。

図 1 より、計算深度が浅い (sindo-1) 場合は、通信の負荷が CUP の負荷より大きいため、プロセッサの数を増やすと、ネットワークを使う通信量が増えることになり、実行時間が長くなる。逆に計算深度が深い (sindo-5) 場合は、通信の負荷が相対的に小さくなるため、プロセッサの数を増やす分だけ、より高い並列性を抽出できることがわかる。

7 おわりに

本稿で述べた並列処理実行系の設計により、関数定義において、チャネルを介して通信する関数間の物理的位置を意識する必要がなくなった。また、共有メモリ型並列処理環境とネットワーク型並列処理環境の統合が可能となった。

今後は、共有メモリ型並列処理環境における問い合わせ処理実験、および、データ識別子による参照時におけるデータ転送の支援について検討する。

参考文献

[1] Kiyoki,Y.,Kato,T. and Masuda,T.: A Relational Database Machine based on Functional Programming Concepts, *Proc. 1986 ACM-IEEE Computer Society Fall Joint Computer Conf.*, pp.969-978,Nov. 1986.

[2] Kiyoki,Y., Kurosawa,T., Kato,K. and Masuda,T.: The Software Architecture of a Parallel Processing System for Advanced Database Applications, *Proc. the 7th IEEE international conference on Data Engineering*, pp.220-229,April 1991.

[3] 大西元, 清木康: 関数型並列データベース・システム SMASH における資源割り当て方式の拡張. アドバンストデータベースシステムシンポジウム論文集, pp.43-51. 情報処理学会,Dec.1990

[4] Liu,P., Kiyoki,Y. and Masuda,T.: Efficient Algorithms for Resource Allocation in Distributed and Parallel Query Processing Environments, *Proc. the 9th International Conference on Distributed Computing Systems*, pp.316-323,June 1989.

[5] 新城靖, 清木康: 並列プログラムを対象とした軽量プロセスの実現方式. 情報処理学会論文誌, Vol.33,No.1, pp.62-73,1992.

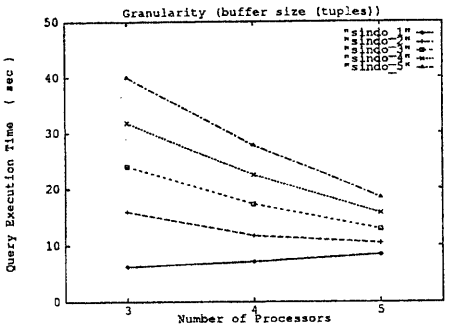

図 1 サイト数と実行時間

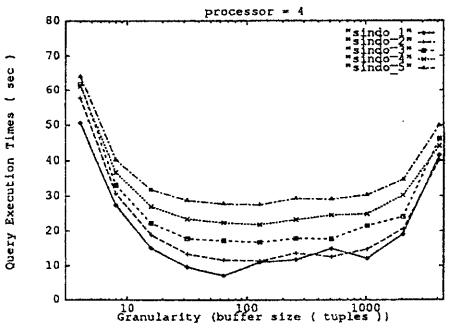

図 2 グラニュラリティーと実行時間