計算機アーキテクチャ　68－12
(1987・11・13)

データベースおよび知識ベースを対象とした並列処理システム：ＳＭＡＳＨ

SMASH : A Parallel Processing System for Manipulating Databases and Knowledge Bases

加藤　和彦*　　清木　康**　　益田　隆司**
Kazuhiko Kato, Yasushi Kiyoki and Takashi Masuda
*筑波大学　工学研究科
**筑波大学　電子・情報工学系
*Doctoral Program of Engineering, University of Tsukuba
**Institute of Information Sciences and Electronics, University of Tsukuba

あらまし　本稿では，データベースの多様な応用分野および知識ベースに柔軟に適応可能な並列処理システムＳＭＡＳＨの関数計算機構について述べる．本並列処理システムは，大量データを対象とする任意の基本演算を並列処理環境の中に容易に組み込む機能を備えている．新しい基本演算は関数型プログラミングの枠組みの中で関数として記述され，それらの基本演算は関数型計算の枠組みの中で並列に実行される．本並列処理システムは，大量データを扱う基本演算群を要求駆動型評価による関数型計算の枠組みの中で並列に実行するもので，関数型計算を大量データの処理に適用するという新しいアプローチに基づくシステムとして位置付けることができる．

ABSTRACT　In this paper we present a functional computation mechanism of a parallel processing system named SMASH. The SMASH is a parallel processing system for manipulating large amounts of data within a framework of functional computation. In order to cope with a great variety of applications of databases and knowledge bases, new basic operations are defined as functions and flexibly integrated into the parallel processing environment. This paper also presents an implementation scheme of the parallel processing system.

はじめに

近年，データベース管理システムの普及とともにその応用分野は急速に広がり，また，データベースの知識ベースへの拡張に関する研究がさかんに行われている．

関係データベースの基本演算である関係データベース演算の処理の高速化を目指して，現在までに並列処理を指向した多くのデータベースマシン・アーキテクチャが提案されてきた．しかし，データベースの多様な応用分野および知識ベースに柔軟に適応するためには，関係データベース演算の処理機能だけでなく，新しい任意の基本演算の追加を行い，それらを並列処理環境の中に容易に組み込む機能が必要となる．

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（たとえば，演繹型データベース）に柔軟に適応できる並列処理システムＳＭＡＳＨの実現を行っている[5][7]．本並列処理システムは，大量データを対象とする任意の基本演算を並列処理環境の中に容易に組み込む機能を備えている．新しい基本演算は関数型プログラミングの枠組みの中で関数として記述され，それらの基本演算は関数型計算の枠組みの中で並列に実行される．関数型プログラムの重要な性質として，プログラムに内在する並列性の抽出が容易であることが知られている．本並列処理システムは，大量データを扱う基本演算群を関数型計算の枠組みの中で並列に実行するもので，関数型計算を大量データの処理に適用するという新しいアプローチによるシステムとして位置付けることができる．

我々は，ＳＭＡＳＨ上で関係データベースへの問い合わせを並列に処理するための方式およびその実現方式を提案し[5][8]，また，演繹データベースへの再帰的呼び出しを含む問い合わせを並列に処理する方式およびその実現方式を提案している[7]．ＳＭＡＳＨ上での演繹データベースの並列処理方式として，関数型計算の枠組みの中でルール／ゴール・グラフ（rule/goal graph)を実行時にトップダウンに構築するインタープリテーション・モデルを示した．その方式では，ルール／ゴール・グラフの各ノードは１関数に対応し，各アークは関数の引数あるいは関数のreturn valueに対応する．このインタープリテーション・モデルを実現する３基本演算（AND, R_unify, F_unify)を提示し，各々を関数として関数計算の枠組みの中で並列に実行する方式を示した．

本稿では，ＳＭＡＳＨが実現する関数計算機構について述べる．

本システムでは，基本演算の対象となる大量データはストリームとして扱われ，基本演算は要求駆動型制御のもとで関数計算として駆動される．ストリームに対する関数計算を要求駆動型制御により実行することにより，限られた計算機資源の中で大量データに対する並列処理が実現される．

このような並列処理環境を実現するために，基本プリミティブを設定した．基本プリミティブは，要求駆動型制御

による関数計算の駆動，ストリームの制御等の関数計算のための基本機能をハードウェアに依存しない水準で記述できるように設定されている．また，基本プリミティブは，処理対象とする個々の基本演算に依存しない水準で，それらの基本機能を実現できるように設定されている．

本システムの対象となる各基本演算は，関数型プログラミングの枠組みの中で，ストリームを入出力とする関数プログラムとして記述される．基本演算を単位として記述された各関数は，関数計算を実行するための基本プリミティブを含む逐次的なオブジェクト・コードに変換される．それらの基本演算は，関数を単位としてプロセッサへ割り当てられ，要求駆動型制御による関数計算の枠組みの中で並列に実行される．その結果，関数を粒度とした並列性が引き出される．

2．システム構成

ＳＭＡＳＨは，図１に示すように複数台のプロセッシング・サイトをネットワークにより結合したハードウェアにより構成される．現在の実験用プロトタイプは，ローカルエリア・ネットワークにより結合された７台のSunワークステーションからなる．各プロセッシング・サイトは，図１に示すような階層により構成されている．

(1) 関数プログラミング・レベル（Functional programming level）

このレベルでは，任意の基本演算を関数プログラムとして記述する環境が実現される．本システムでは，関数を単位としてプロセッシング・サイトへの割り当てを行うので，このレベルにおける関数の単位を粒度とした並列性が引き出される．例えば，関係データベース演算の処理系を実現する場合，選択演算，射影演算，結合演算などの各関係データベース演算を基本演算として，すなわち，１関数として記述した場合には，関係データベース演算間の並列性が抽出されるが，個々の関係データベース演算をさらに細分化し，複数の関数により定義した場合には，演算内の並列性も抽出されることになる．

(2) プログラム変換レベル（Program transformation level）

関数プログラミング・レベルにおいて記述された各関数は，このレベルで逐次的に実行されるオブジェクト・コードへ関数単位に変換される．このオブジェクト・コードは，要求駆動型制御による関数計算の駆動，並列性の抽出，ストリームの制御を実現する基本プリミティブを含む．基本プリミティブについては，各プロセッシング・サイトにおいて共通の解釈が行われる．

(3) 実行・通信制御レベル（Execution & communication control level）

オブジェクト・コードに変換された関数が実際にこのレベルにおいて駆動される．このレベルでは，基本プリミティブの解釈が行われ，また，プロセッシング・サイト間あるいはプロセッシング・サイト内の関数間でストリーム・データおよびデマンドの通信が制御される．

(4) アーキテクチャ・レベル（Architecture level）

各プロセッシング・サイトが基本プリミティブに関して共通の解釈を行うことにより，関数間での並列処理が実現される．各プロセッシング・サイトは，実行・通信制御レベルにおいて基本プリミティブを解釈し，アーキテクチャ・レベルにおいてその実行を行う．アーキテクチャ・レベルは各プロセッシング・サイトで異なっていてもよく，アーキテクチャ・レベルで実行可能なコードへの基本プリミティブの解釈は実行・通信制御レベルにおいて行われる．

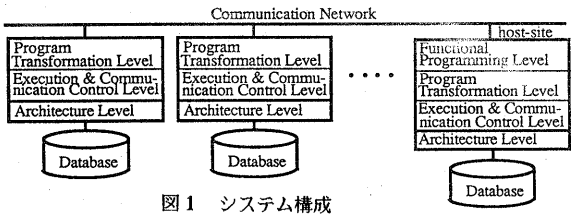

図１　システム構成

3．大量データを対象とした関数計算の並列処理方式

引数の評価と関数の引数への適用(apply)を行うことを関数計算という．関数計算の最も重要な性質は，関数の値がその関数の引数値だけから決定されることである．この性質は参照透明性(referential transparency)と呼ばれ，同じ関数が同じ文脈で複数回出現している場合，それらはすべて同じ値を返す．この性質により，関数の評価の過程で副作用を生じないので，関数プログラムに内在する並列性を抽出することが容易となる．提案方式は，このような特徴を持つ関数計算を大量データを扱う処理に適用した新しい方式として位置づけることができる．

本方式では，関数計算の枠組みの中で大量データを扱うために，大量データをストリームと呼ばれるデータ構造で表現する．ストリームは次のような性質をもつものをいう．

(1)　値の列(sequence)であって，その値の数は概念的に無限個であってもよい．

(2)　ストリームは，必ず，先頭から後方へ向かって参照される．

(3)　ストリームを構成する個々の要素の値は，参照要求があったときに確定されれば良い．

ストリームを入力または出力とする関数，あるいはストリームを入出力とする関数をストリーム関数と呼ぶ．また，実行時に起動されたストリーム関数を関数インスタンスと呼ぶ．本方式では，ストリーム関数のストリーム型引数に対する関数適用は，そのストリーム関数を関数インスタンスとして生成・起動し，実引数を生成する関数インスタンス（生産者）と起動された関数インスタンス（消費者）を１方向にデータが流れる通信チャネルで結合することにより行う．ストリーム型引数をストリーム関数へ適用する場合の引数受渡し法として３．２に示す５通りの方法を通信チャネルが実現する．

関数プログラムで記述された再帰呼び出し毎に同一関数インスタンスを起動すると，各関数インスタンスを異なるプロセッサに配置することにより引き出される並列性よりも，関数インスタンスの起動に関わるオーバヘッドが大きくなり，効率を低下させる場合がある．この問題に対処するための手法として再帰除去(recursion removal)がある[2][3]．この手法を用い，再帰呼び出しを繰り返し(iteration)に変換することにより，複数個の関数インスタンスを生成して実行すべき計算を，１個の関数インスタンスにより行うことができる．

3．1　駆動方式

関数計算において，評価可能な関数式が複数存在するとき，評価を行う順序は，計算の駆動方式によって定められる．関数計算の駆動方式は次のように分類できる[1][9][10]．

（1）逐次型評価(sequential evaluation)
（2）データ駆動型評価(data-driven evaluation)
（3）要求駆動型評価(demand-driven evaluation)

データベース処理や知識ベース処理のような大量データを扱う処理においては，処理対象データのメモリ・オーバーフローが処理の続行を不可能にしたり，入出力操作による処理効率の低下を引き起こす．本方式では，限られた計算機資源の中で大量のデータを並列処理するために，関数計算の駆動方式として要求駆動型評価を用いる．

要求駆動型評価は，関数の入力引数の値が必要になった時点で，その引数値の計算を行う関数に対してデータの生成を要求するデマンドを伝達し，デマンドを受け取った関数は計算を開始する．データ駆動型評価とは対照的にトップダウンで計算が進められる．要求駆動型評価は次のような特徴をもつ．

（a）計算の実行制御がデマンドよって制御されるため，並列性の制御を行い易い．計算がトップダウンで行われるため，消費者関数の計算の進行に応じて生産者関数の計算を行うことができる．

（b）引数値が必要となった時点でその値を求める計算が開始されるため，ノンストリクト関数を計算する場合は不必要な計算は行われない．

（c）データが必要になったときに必要なデータ量だけをデータの生成者に生成させるように，デマンドにより制御することができる．すなわち，データの一部が生成された時点で，そのデータに対する計算を開始できる関数を評価する場合には，1回のデマンドで生成するデータ量をあらかじめ定めておき，データを計算する側の関数が必要に応じてデマンドを発行することにより，そのデータを生成する側の関数のデータ生成を制御することができる．この特徴により，限られたメモリ資源下で大量データに対する並列処理を実現することが可能となる．

（d）計算の実行制御がデータの受け渡しの他に，引数の値を要求するデマンドの受渡しを行う必要がある．

要求駆動型評価の場合，データの伝達以外にデマンドの伝達が必要であり，デマンドの伝達によるオーバーヘッドが処理効率の悪化を引き起こす可能性がある．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ー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ーヘッドは問題とならない．むしろ，大量データを処理する場合は，要求駆動により引数を評価することにより，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．

我々は既に文献[6]において，大量データを扱う関数計算の駆動方式をデータ駆動型評価と要求駆動型評価とで行った場合の比較を行い，大量データを扱う関数計算では要求駆動型評価が有効であることを示している．

本方式では，要求駆動型評価のもとで，関数型計算に内在する次のような並列性が引き出される．

（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，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）

（1）の並列性は，複数の引数データの消費を行う関数が，それらの引数データを生成する関数に対してデマンドの発行を同時に行うことにより引き出される．これにより，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を並列に行うことができる．

（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，消費者側の関数によって発行された1回のデマンドに対して，予め定められた一定量のストリーム要素を生成した後，計算を中断して次のデマンドを待つ．これにより，ストリームの生産者と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ストリーム・データの量を以下，グラニュラリティと呼ぶ．

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ーヘッドが増大し，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ーヘッドは減少する．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ストリーム全体に設定される場合には，（2）の並列性は引き出されない．）既に，文献[4][5]において，グラニュラリティのストリーム型並列性に与える影響に関する実験結果を示し，ストリーム型の並列性を最大限に引き出すためのグラニュラリティの設定に関して議論している．

3．2　引数受渡し法

関数計算の枠組みの中で大量データに対する処理を行う場合，関数間のデータの受渡し法が重要となる．例えば，実際の計算機の使用可能なメモリ量は限られているので，適切なデータの受渡し法を用いない場合，受け渡されるデータ量が使用可能なメモリ量を越え，多大なオーバーヘッドを引き起こすか，あるいは，処理の続行が不可能となる．本処理方式は様々な計算機資源（プロセッサ数およびメモリ量）に柔軟に適応するために，表1および図2に示す5種類の引数受渡し法を提供する．

ここでいう共有ありとは，要求駆動型評価におけるcall-by-needによる引数受渡しに対応し，また，共有なしとは，call-by-nameによる引数受け渡しに対応する．第1レベルの共有とは，複数の関数インスタンスによる同一ストリームの同時参照による共有である．第2レベルの共有とは次の通りである．一般に，関数が再帰呼び出しを含む場合，同一の複数の関数インスタンスが生成される．このとき，それらの複数インスタンス各々が同一ストリームを参照する場合がある．このような再帰呼び出しを含む関数が，再帰除去によって，複数の関数インスタンスを生成しない1個の関数インスタンスに置き換えられると，そのストリームはループにより毎回参照されることになる．この参照のことを，以下では，再参照と呼ぶ．このとき，最初の参照時に生成したストリーム全体を保持しておき，2回目以降

表１　引数受渡し法

第１レベル共有なし	第２レベル共有なし	再計算	①
	第２レベル共有あり	生産者サイト・キャッシング	②
		消費者サイト・キャッシング	③
第１レベル共有あり	第２レベル共有なし		
	第２レベル共有あり	生産者サイト・キャッシング	④
		消費者サイト・キャッシング	⑤

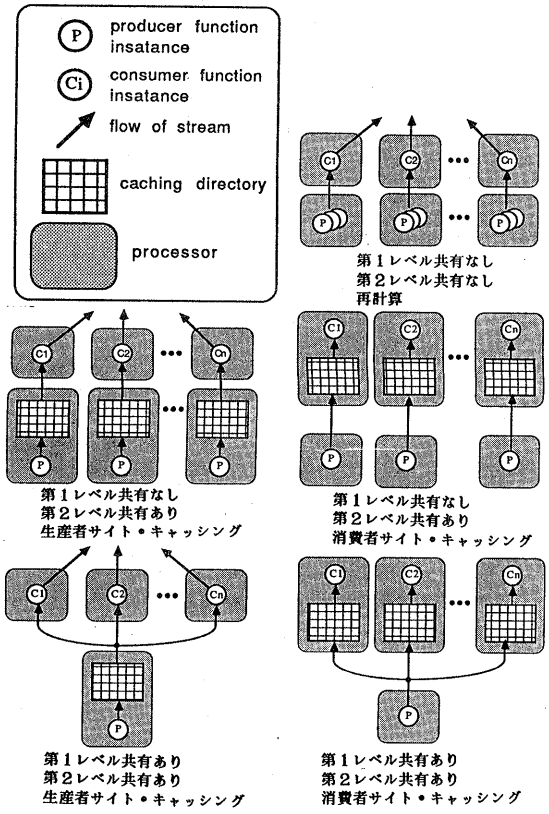

図２　引数受渡し法

の参照では，そのストリームを読み出すことによる共有を第２レベルの共有と呼ぶ．再参照時に毎回そのストリームの再生成（再計算）を行う場合は第２レベルの共有なしと呼ぶ．第１レベルおよび第２レベルの共有をするためにストリーム全体を格納しておくことをキャッシングと呼び，格納領域をキャッシング・ディレクトリと呼ぶ．

表１の①～③は，第１レベルの共有をしない受渡し法であり，④，⑤は第１レベルの共有を行う受渡し法である．ｎ個の消費者関数インスタンスが同一のストリームを入力する場合を考える．①～③の受渡し法ではｎ個の生産者関数インスタンスが必要であるのに対し，④～⑤の受渡し法ではただ１個の生産者関数インスタンスが生成されるだけである．

第１レベルの共有なしで，かつ，第２レベルの共有なしの方式が①であり，第２レベルの共有をする方式が②，③である．②と③の各々は，第２レベルの共有をする場合にキャッシング・ディレクトリを生産者側のサイトにとる方式，および，消費者側のサイトにとる方式に対応する．

第１レベルの共有を行う場合には，一般に個々の消費者関数インスタンス毎に消費速度と再計算の回数が異なるので，生産者関数インスタンスと消費者関数インスタンス群の間にストリーム全体を格納するキャッシング・ディレクトリが必要となる．このように，「第１レベル共有あり」を実現するためには，キャシッング・ディレクトリが必要であり，「第２レベル共有なし」で再計算を行うことは意味がない．従って，第１レベルの共有を行う場合には，第２レベルの共有をしない方式は必要ない．第２レベルの共有をする方式に関しては，④，⑤のみを実現する．④は，生産者側のサイトにキャッシング・ディレクトリを用意する方式であり，⑤は消費者側のサイトに用意する方式である．

３．３　非決定性

本方式において，(1) 関数間のより高い並列性を引き出すため，および，(2) 関数インスタンス内で生成される子関数インスタンスとの間でストリーム要素の受渡しを行うためには，非決定性(nondeterminism)の導入が不可欠である．このことを示すために次のように定義される２変数関数f-g-unionを考える．

(f-g-union x y) = (union (f x) (g y))

変数x，yはストリーム型の変数，関数f-g-union，f，g，unionはストリームを入力とし，ストリームを出力とする関数とする．関数unionは，２個の入力ストリームのいずれか一方が到着すれば内部の計算を開始できる関数とする．この関数式から，union，f，gのそれぞれに対応する関数インスタンスが生成され，並列処理が行われる．

(1) に関して，本方式では，関数内部は逐次コードに変換されて実行されるが，もし非決定性を記述できないとすると，関数インスタンスfまたはgのどちらから先にストリームの要素を受け取るかを逐次コード生成時に決定しなければならない．その場合，２個の入力のいずれか一方が到着すれば内部の計算を開始できるというunion関数の性質による並列性を引き出せない．この並列性を引き出すためには，union関数において，先に到着したストリーム・データを非決定的に選択して処理を開始するという機能が必要である．このように，デマンドおよびストリーム・データの到着を非決定的に選択し，デマンドおよびストリーム・データの到着に合わせて非決定的操作を行う機能が必要となる．関数計算は，計算の出力が入力によってのみ定まるという決定的な性質を持つため，非決定性の導入による関数計算の拡張には注意を要する．ストリームを要素間の順序に意味がある列（シーケンス）として扱わなければならないような処理においては，このような非決定性の導入は困難である．しかし，データベース処理や知識ベース処理のような応用分野では，基本演算間で渡されるデータを要素間の順序関係に意味がない集合として扱える場合が多い．

本方式では，ストリームを要素間の順序に意味がないデータの集合とみなせる場合は，ストリームを集合として扱う．

(2) に関して，本方式では，関数f-g-unionのインスタンス内で，xおよびyをストリームとして各々 fおよび gに渡し，return valueとして unionからストリームを受け取る．このとき，xおよびyの引渡しとreturn valueの受取りを非決定的に行わないとデッドロックが発生する．デッドロックを回避するためには，非決定性を導入し，関数f，gからデマンドが発行された場合はそれぞれストリームx，yの要素を渡し，関数unionのreturn valueの一部としてストリームの要素が生成された場合はその要素を受け取ることが必要となる．

4．基本プリミティブ

3．で述べた要求駆動型制御に基づく関数計算の提案方式を実現するのに必要な基本機能を明らかにし，それらを基本プリミティブとして設定した．基本プリミティブは，アーキテクチャに依存しないレベルで並列処理プログラムの記述を行うことを可能にする．本処理方式では，各関数は関数型プログラミングの枠組みの中で記述され，基本プリミティブの呼び出しを含むプログラムに変換され，実行される．基本プリミティブを表2に示す．

関数インスタンスの生成とチャネルによる結合

関数インスタンス間で受け渡されるデマンドとデータの受渡しの仲介となり，3．2に示した引数受渡し法を実現する機能をチャネルと呼ぶ．指定したプロセッサの間にデマンドとストリーム・データを通信するための通信路を設定する基本プリミティブをchannel，指定したプロセッサ上に関数インスタンスを生成する基本プリミティブをnewと呼ぶ．基本プリミティブchannelは，設定した通信路の識別子（システム中で一意）を返す．この識別子をチャネル識別子と呼ぶ．チャネル識別子は基本プリミティブnewにより，生成される関数インスタンスに渡される．生成された関数インスタンスは，この識別子を用いて他の関数インスタンスとデマンドおよびストリーム・データの受渡しを行う．

ストリームの共有

3．2において議論した第1レベルの共有あり・なしは，生産者関数インスタンスの生成個数およびチャネルの結合によって決められる．第2レベルの共有あり・なしは，キャッシング・ディレクトリの配置を行うか否かをプリミティブchannelのcaching_site引数で指定される．また，キャッシング・ディレクトリの配置を行う場合には，それを生産者側のサイトに配置するか，消費者側のサイトに配置するかに関するしてもcaching_site引数で指定される．

ストリーム要素の受渡し

関数インスタンス間で受け渡されるストリームは次の2種類がある．

（1）関数インスタンスの出力が他の関数インスタンスの入力となる場合．

例：(f .. (g ..) .. (h ..))

（2）関数インスタンスが本体の計算の中で他の関数インスタンスを呼び出す場合．

例：(f ..) = (.. (g s) ..)，sは関数fから関数gへ渡されるストリーム．

（1）では関数インスタンス群の間でのストリームの受渡し関係がトリー状になる．この場合，関数の入力引数として受け渡されるストリームに対しては，プリミティブgetによりストリームの要素をチャネルから取り出す．関数のreturn valueの一部としてストリームの1要素をチャネルに書き出すときはプリミティブputを用いる．

（2）では，関数インスタンス間でのストリームの受渡し関係はグラフ状になり，ストリームの受渡し関係にサイクリック構造が現れる．そのようなサイクリック構造が生じるとデッドロックが発生する可能性がある．これに対処するために，ストリームの受渡しに関して，（1）と異なるプリミティブを用いてデッドロックを回避する．（2）のように子関数インスタンスを生成し，その子関数インスタンスにストリームの1要素を渡す場合には，プリミティブsendを使用し，子関数インスタンスがreturn valueとして生成したストリームの1要素を受け取る場合はプリミティブreceiveを使用する．プリミティブget，receiveはいず

表2　基本プリミティブ

new(f, pid, parameters, cid)	
機能	fで指定された関数のインスタンスをpidで指定されたプロセッサ上に生成する．
channel(type, granularity, stream_oriented_parallelism, produer_pid, consumer_pid_list, caching_site)	
機能	生産者関数インスタンスと消費者関数インスタンスの間でストリームデータを受け渡すためのチャネルを設定し，設定したチャネルの識別子を返す．
channel_expand(cid, pid)	
機能	cidで指定されたチャネルを，pidで指定されたプロセッサに配置される関数インスタンスを消費者とするよう拡張する．
predemand(cid)	
機能	ストリーム・データの参照に先行して消費者関数インスタンスから生産者関数インスタンスへデマンドを先出しする．
get(cid)	
機能	cidで指定されたチャネル内の消費者バッファから，生産者のストリーム関数のreturn valueとして生成されたストリームの1要素を取り出す．取り出すべき要素がない場合は，生産者関数インスタンスにデマンドを発行し，呼び出した消費者関数インスタンスをsuspendし，カーネル内のスケジューラを起動する．
put(cid,el)	
機能	cidで指定されたチャネル内の生産者バッファへ，自関数インスタンスのreturn valueとしてストリームの1要素を書き出す．チャネル内のバッファが一杯になったときは，生産者関数インスタンスにデマンドを発行し，呼び出した生産者関数インスタンスをsuspendし，カーネル内のスケジューラを起動する．
receive(cid)	
機能	cidで指定されたチャネル内の消費者バッファから，自分が生成した子関数インスタンスがreturn valueとして生成したストリームの1要素を取り出す．取り出すべき要素がない場合でも，このプリミティブを呼び出した消費者関数インスタンスをsuspendすることはない．がないチャネルにこの基本プリミティブが発行された場合の返却値は未定義となる．
send(cid, el)	
機能	cidで指定されたチャネル内の消費者バッファへ，ストリームの1要素を書き出す．チャネル内のバッファが一杯になったときでも，呼び出した生産者関数インスタンスをsuspendすることはない．
rewind(cid)	
機能	cidで指定されたチャネルに対し，チャネルを経由して渡されるストリームの再参照を要求する．
select(cid1, cid2, .., cidn)	
機能	cid1, cid2, .., cidnで指定されたチャネルの中から，下に示す条件に合致する1チャネルを非決定的に選択する． ①まだ消費されていないストリーム・データがチャネル内の消費者バッファに残っている入力チャネル． ②データを要求するデマンドが到着しており，発行されたデマンドに対する1グラニュラリティ分のストリーム・データを生成し終えてない出力チャネル． 引数として指定された入力チャネルの中で，チャネル内の消費者バッファが空であって，デマンドが発行されていない入力チャネルにはデマンドを発行する．
enable(cid)	
機能	cidで指定されたチャネルを選択可能とする．
disable(cid)	
機能	cidで指定されたチャネルを選択不可能とする．

れもチャネル内の消費者バッファからストリームの１要素を取り出すプリミティブであるが，バッファ内の要素が尽きた場合に動作が異なる．getは，要素が尽きた場合，関数インスタンスの実行を中断し，生産者関数インスタンスにデマンドを発行するが，receiveは，要素が尽きた場合でも実行を中断しない（消費者バッファが空のチャネルにreceiveを発行した場合，receiveが返す値は不定となる）．これに対応してプリミティブputとsendも動作が異なる．putは，チャネル内のバッファが満たされた場合，次のデマンドが到着するまで関数インスタンスの実行を中断するが，sendは中断しない（sendを発行してもストリーム要素のバッファへの書き込みは行われない）．プリミティブsend，receiveは，プリミティブselectと組み合わせて使用される．プリミティブsend，receiveの中ではデータの到着およびデマンドの到着を待ち合わせず，プリミティブselectの中でデータあるいはデマンドの到着を待つ．プリミティブselectの非決定性を実現する機能を用いて，子関数からのデータの到着とデマンドの到着を非決定的に待つことにより，子関数インスタンスとのストリームの受渡しに関するデッドロックを回避することができる．

並列性の抽出

３．１で述べた２種類の並列性は，プリミティブselectとpredemandを用いて抽出することができる．まず第１の並列性は，複数の生産者関数インスタンスに同時にデマンドを発行することによって行うが，これはプリミティブselectにより実現される．

第２の並列性(stream parallelism)は，消費者関数インスタンスがストリームの要素の参照に先行してデマンドを発行することにより抽出される．これはプリミティブpredemandを用いて行う．

ストリームの再参照

３．２で述べたように，再帰呼び出しを含む関数に対して再帰除去を行うと，ストリームの再参照が必要となる場合がある．この機能を提供するプリミティブをrewindと呼ぶ．rewind(cid)により，チャネル識別子cidで指定されたチャネルを経由して得られるストリームを再参照することができる．

非決定的選択

プリミティブselectは，１個または複数のチャネルに対してデマンドの到着またはデータの到着を非決定的に選択する．この機能は次のような場合に使用される．

（１）　同時に複数の関数インスタンスタンスに対してデマンドを発行する場合．

（２）　子関数インスタンスとストリームの受渡しを行う際，デッドロック状態を回避するために，子関数インスタンスからのデマンドの到着とストリーム・データの到着を多重待ちする場合．

（３）　複数のストリームを受け渡す場合に，データあるいはデマンドが到着したチャネルを非決定的に選択することにより，より高い並列性を引き出す場合．

（１），（２）については，それぞれ並列性の抽出，およびストリーム要素の受渡しの項で述べた．（３）の場合について，プリミティブselectを用いて記述した和演算のプログラムを用いて議論する．ここで示すプログラムは，実行可能なオブジェクト・コードをＣ言語の記述法を用いて抽象的に示したものである．

```
function union(str1, str2) return result_str
  stream_of_tuple str1, str2;
{
  tuple t;

  predemand(str1);
  predemand(str2);
  while (!str1_exhausted && !str2_exhausted) {
    selected = select(str1, str2);
    switch (selected) {
    str1:
      t = get(str1);
      if (t == EOS)
        str1_exhausted = TRUE;
      else {
        if (check_duplicate(pool, t)) {
          put(result_str, t);
          append(pool, t);
        }
      }
      break;
    str2:
      t = get(str2);
      if (t == EOS)
        str2_exhausted = TRUE;
      else {
        if (check_duplicate(pool, t)) {
          put(result_str, t);
          append(pool, t);
        }
      }
      break;
    }
  }
  put(result_str, EOS); /* EOS: End Of Stream */
}
```

このように，str1，str2のどちらに先にデータが到着するか，プログラムの記述時には決定できない場合，select(str1，str2)により実行時に選択を行うことが可能となる．

プリミティブselectは，複数のチャネルへのデマンドおよびストリーム・データの到着を多重待ちし，非決定的に選択を行う．多重待ちするチャネルに制限を加える機能を提供するプリミティブがdisable，enableである．disable(cid)はチャネル識別子cidで指定されたチャネルをプリミティブselectの選択対象から外し，enable(cid)は選択対象に加える．

チャネル識別子cid1，cid2，cid3で指定されるチャネルからストリームを入力として受け取り，３個のストリームの各要素に対して処理Pを施すプログラムを次に示す．

```
f(cid1, cid2, cid3)
{
  stream_element el[3];
  int i;
        ...
  enable(cid1); enable(cid2); enable(cid3);
  for (i = 0; i > 3; i++) {
    cid = select(cid1, cid2, cid3);
    el[i] = get(cid);
    disable(cid);
  }
  ... = P(el[0], el[1], el[2]);
  ...
}
```

このプログラム中のenable，disableの機能により，select(cid1，cid2)，select(cid2，cid3)，select(cid3，cid1)，select(cid1)，select(cid2)，select(cid3)の６個のselectとそれに付随するプログラムを省くことができる．

チャネルの動的拡張

プリミティブchannelを発行する場合には，そのチャネルを介してストリームを受け取る消費者関数インスタンスの個数があらかじめ決定されていなければならない．しかし，次のような場合には，チャネル生成時に消費者関数インスタンスの個数を決定できない．

（1）子関数インスタンスの計算を進行させないと他の子関数インスタンスを生成できない場合．

（2）ストリームを共有するすべての子関数インスタンスを生成する前に，一部の子関数インスタンスの実行を開始させたい場合．

3．2で述べた第1レベルの共有を行う場合には，すでに生成されているチャネルに対して消費者関数インスタンスを動的に追加する機能が必要である．この機能を提供するプリミティブがchannel_expand(cid, pid)である．このプリミティブは，すでにプリミティブchannelで生成されたチャネル（チャネル識別子cidで指定される）に対し，プロセッサ識別子pidで指定されたプロセッサ上に生成される関数インスタンスを消費者関数インスタンスとして追加する．ただし，この場合，チャネル識別子cidで指定されるチャネルの属性としてPRODUCER_SITE（生産者関数インスタンスが配置されるサイト側にキャッシング・ディレクトリをもつ）が指定されていなければならない．CONSUMER_SITE（消費者関数インスタンスが配置されるサイト側にキャッシング・ディレクトリをもつ）が指定されている場合には，後から追加された消費者関数インスタンスにストリーム・データを渡すことができない．

5．基本プリミティブの実現

我々は，4．で示した基本プリミティブを複数台の汎用プロセッサが高速ネットワークで結合されたハードウェア環境で実現する並列処理システムSMASHの製作を進めている．本章では，SMASHにおける基本プリミティブの実現法について述べる．基本プリミティブは，SMASHの実行・制御レベルにおいて解釈される．このレベルは，図3に示すように実現されている．

このレベルを構成するモジュールにはカーネル，チャネル，関数インスタンスがある．カーネルは各プロセッサに1個存在し，

・関数インスタンスの生成と消去
・チャネルの生成と消去
・関数インスタンスの実行のスケジュール
・プロセッサ間の通信

を行う．ストリームの受渡しは，生産者関数インスタンスと消費者関数インスタンスが異なるプロセッサに配置される場合と，同一プロセッサに配置される場合がある．プロセッサ間通信とプロセッサ内通信を関数インスタンスに対して透明にするために，チャネルを次のように構成する．1個のチャネルについて，生産者関数インスタンスとの間でデマンドおよびストリームの受渡しを行う部分をproducer channel portとして，生産者関数インスタンスと同一のプロセッサ上に配置する．1個のチャネルに接続する消費者関数インスタンスは1個または複数個存在するが，各消費者関数インスタンスとの間でデマンドおよびストリームの受渡しを行う部分をconsumer channel portとして，各消費者関数インスタンスと同一のプロセッサ上に配置する．同一プロセッサ内のデマンドおよびストリームの受渡しはproducer channel portとconsumer channel portの間の共有メモリを用いた通信により行い，異なるプロセッサ間の受渡しはネットワークを経由したメッセージ・パッシング方式で行う．

システムを構成するモジュール間の通信階層を図4に示す．基本プリミティブは，関数インスタンスとカーネルおよびproducer/consumer channel portとの間のインターフェースと位置づけられる．関数インスタンスの生成(new)，チャネルの生成と拡張(channel, channel_expand)，チャネルの選択(select, enable, disable)はカーネルの機能として実現する．また，関数インスタンス間のストリーム要素の受渡し(get, put, receive, send)，デマンドの先出し(predemand)，およびストリームの再参照(rewind)に関するプリミティブは，producer/consumer channel portの機能として実現する．

基本プリミティブ以外のモジュール間のインターフェースとして以下のものを用いる．

(1) Channel Kernel Interface (CKI)

Consumer channel portおよびproducer channel portがカーネルにプロセッサ間の通信および関数インスタンスの起動を依頼するためのインターフェース．

(2) Network Communication Interface (NCI)

カーネルがネットワークを介してプロセッサ間通信を行う場合のカーネルとネットワーク間のインターフェース．

また，同一階層間のプロトコルとして以下のものを用いる．

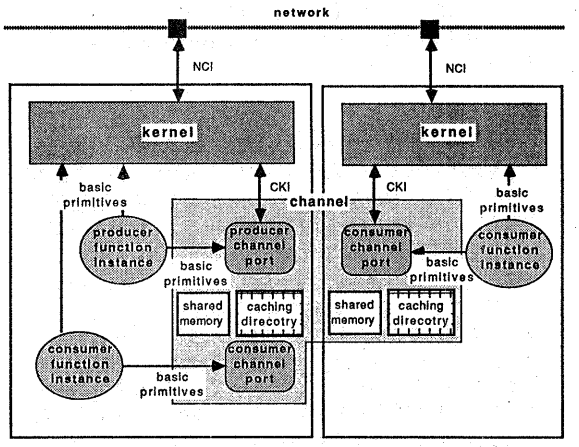

図3　実行・通信制御レベルの実現

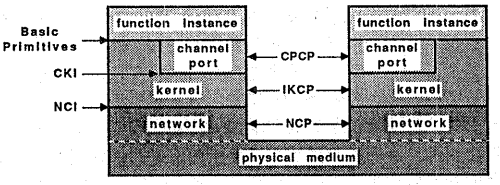

図4　モジュール間の通信階層

(1) Channel Port Communication Protocol(CPCP)
consumer channel portとproducer channel portがデマンドおよびストリームの転送を行うためのプロトコル.
(2) Inter-Kernel Communication Protocol (IKCP)
カーネル間で基本プリミティブの解釈を行うために使われるプロトコル.
(3) Network Communication Protocol (NCP)
プロセッサ間を結合するネットワークにより通信を行うためのプロトコル.使用するネットワークに依存する.

これらのプロトコルを用いて,例えば,図3に示す1個の生産者関数インスタンスが生成するストリームを2個の消費者関数インスタンスが共有する場合(第1レベルの共有),ストリームの転送は次のようにして行われる.

(1)生産者関数インスタンスがプリミティブputまたはsendによりストリームの1グラニュラリティ分のデータの生成を終える.生成したストリーム・データは,共有メモリ領域に置かれる.

(2)-1 生産者関数インスタンスと同一のプロセッサに配置されている消費者関数インスタンスに対しては,その消費者関数インスタンスに接続しているconsumer channel portにCPCPにより1グラニュラリティ分のストリーム・データを共有メモリ中に生成し終えたことを伝える.

(2)-2 生産者関数インスタンスと異なるプロセッサに配置されている消費者関数インスタンスに対しては,共有メモリ領域中に生成された1グラニュラリティ分のストリーム・データを相手プロセッサのconsumer channel portへ転送することを,自プロセッサ内のカーネルへCKIにより依頼する.カーネルはIKCPを用いて共有メモリ中のストリーム・データを相手のプロセッサのカーネルに転送する.転送はNCPにより行われる.相手のカーネルは,CKIを用い,指定されたconsumer channel portにストリーム・データを渡す.

7. おわりに

本稿では,大量データを扱う様々な応用分野に適応できる並列処理方式とその実現法について述べた.様々な計算機資源(メモリ資源およびプロセッサ資源)に柔軟に対応するための多様なデータ受渡し機構と,要求駆動型制御の枠組みで計算を行うための基本プリミティブを示した.

現在,我々は7台のSunワークステーションをローカル・エリア・ネットワークにより結合した環境で,提案並列処理方式の実現を進めている.また,大量データに対する処理を高水準レベルで記述できる関数型言語と,その関数型言語で記述したプログラムを,本稿で示した基本プリミティブ呼び出しを含む並行動作プログラムの集合に変換する言語処理系の設計と製作を進めている.

本稿では,大量データを扱う関数計算を疎結合型並列処理環境上で実現する方法を示したが,この方法は,同一プロセッサ内の通信と異なるプロセッサ間の通信を行うために,チャネル内のチャネル・ポート間の通信としてメッセージ・パッシングによる通信と共有メモリによる通信を使用している.このため,本稿で示した実現法を,共有メモリをもつ密結合型並列処理環境上での実現に拡張することは比較的容易である.現在,疎結合型並列処理環境と疎結合型並列処理環境が融合した並列処理環境が現実のものとなりつつある.本稿で示した並列処理方式をそのような並列処理環境においても実現していく予定である.

参考文献

[1] Amamiya, M. and Hasegawa, R.: Dataflow Computing and Eager and Lazy Evaluations, New Generation Computing, Vol. 2, No. 2, pp. 105-129 (1984).
[2] Burstall, R. M. and Darlington, J.: A Transformation System for Developing Recursive Programs, JACM, Vol. 24, No. 1, pp. 44-67 (1977).
[3] Huet, G. and Lang, B.: Proving and Applying Program Transformations Expressed with Second-Order Patterns, Acta Informatica, Vol. 11, pp. 31-55, (1978).
[4] 加藤,清木,益田:並行プログラムのストリーム型並列処理方式,情報処理学会計算機アーキテクチャ研究会62-2, (1986).
[5] Kiyoki, Y., Kato, K. and Masuda, T.: A Relational Database Machine Based on Functional Programming Concepts, Proc. of ACM-IEEE Computer Society Fall Joint Computer Conf., pp. 969-978 (1986).
[6] Kiyoki, Y., Kato, K. and Masuda, T.: A Stream-Oriented Approach to Distributed Query Processing in a Local Area Network, Proc. of 1986 ACM SIGSMALL/PC Symp. on Small Systems, pp. 146-155 (1986).
[7] Kiyoki, Y., Kato, K., Yamaguchi, N. and Masuda, T.: A Stream-Oriented Approach to Parallel Processing for Deductive Databases, Proc. of 5th International Workshop on Database Machines, pp. 102-115 (1987).
[8] 清木,劉,益田:関係演算のストリーム指向型並列処理における資源割り当て方式,情報処理学会論文誌, Vol. 28, No. 11 (1987).
[9] Treleaven, P. C., Brownbridge, D. R. and Hopkins, R. P.: Data-driven and demand-driven computer architecture, ACM Computing Surveys, Vol. 14, No. 1, PP. 93-144 (1982).
[10] Vegdahl, S. R.: A Survey of Proposed Architectures for the Execution of Functional Languages, IEEE Trans. on Computer, Vol. C-33, No. 12, pp. 1050-1071 (1984).