計算機アーキテクチャ 124－5
（1997．6．27）

SPMD モデルによる関数型プログラム実行の一検討

中泉光広† 沈紅†
小林広明† 中村維男†

関数型言語は，手続き型言語と異なり，参照透明性や高いプログラムの生産性，検証容易性など，多くの有用な特徴を持つ．しかし，通常の計算機上では十分な処理速度が得られないため，その使用が大きく制限されてきた．これに対し，関数型言語の特徴の一つである並列実行の容易性を利用することによって，並列計算機上で関数型言語の高速実行を図ることが可能である．本稿では，SPMD モデルを用いてグラフ簡約を並列化し，関数型プログラムを並列計算機上で実行する方式を提案する．提案した手法を並列計算機 IBM SP2 に実装し，評価を行なう．ベンチマークプログラムによる実験結果は本手法の見通しを示した．

Implementing Functional Programs Based on the SPMD Model

MITSUHIRO NAKAIZUMI,† HONG SHEN,† HIROAKI KOBAYASHI† and TADAO NAKAMURA†

Functional languages, which are different from the imperative ones, are characterized with the referential transparency, high programming productivity, and the ease of program verification. However, they are prevented from wide acceptance due to the inefficiency of their implementation on conventional computers. Parallel execution of functional programs utilizing their potential parallelism is a promising way to solve this problem. This paper studies the parallel execution of functional programs based on the SPMD model. We realize the parallel execution of functional programs on parallel computer IBM SP2. The experimental results of benchmark programs reveal the perspective of the execution model.

1. はじめに

関数型言語は，手続き型言語と異なり，参照透明性や検証容易性，プログラムの高い生産性など，多くの有用な特徴を持つ．しかし関数型言語のセマンティクスと通常の計算機アーキテクチャとの間にギャップが存在するため，通常の計算機上では十分な処理速度が得られないという問題がある．

関数型言語を計算機上で実現する方法として，主にグラフ簡約[5]が利用されている．グラフ簡約を高速に実現する手法として，並列処理が有効だと考えられる．しかしながら，並列計算機上でグラフ簡約を実現する場合，簡約グラフを如何に分割し並列タスクを生成するかが実行速度に大きな影響を与え，関数型プログラムの高速実行を実現するための重要な課題となる．

そこで，本研究では関数型プログラムの並列グラフ簡約を実現することを目的とし，SPMD (Single Program Multiple Data) モデルを用いてグラフ簡約の並列化を行なう実行方式を考察，検討する．更に提案した並列実行方式を並列計算機 IBM SP2 上に実装し，その評価を行なう．

はじめに，第2章では本研究で扱う関数型言語 FL およびそのグラフ表現である簡約グラフについて説明を行ない，簡約グラフにおいて並列性がどのように現われるかを考察する．また，考察に基づきグラフ簡約の SPMD モデルによる並列化の手法を述べる．第3章では，第2章の手法を並列計算機 IBM SP2 に実装し，ベンチマークプログラムを用いて実験を行なった結果を示し，これを検討する．第4章ではまとめおよび今後の課題を述べる．

2. 関数型プログラムの SPMD 実行モデル

2.1 関数型言語 FL および簡約グラフ

本研究では，関数型言語として FL[2),3)] を使用する．FL は FP[1)] を基礎とし，実用性を考慮して J. Backus らによって開発された関数型言語の一つである．FL は関数型言語が本来持つ参照透明性や並列実行の容易性などの特徴を持つ．

† 東北大学情報科学研究科
Department of Computer and Mathematical Sciences Graduate School of Information Sciences Tohoku University

Combining Forms

apply	∘	f ∘ g : x ≡ f : (g : x)
apply-to-all	α	α: f : <x1, ..., xn> = <f : x1, ..., f : xn>
cons	[ ]	[f, g] : x = <f:x, g:x>

Primitive Functions

```
id : x = x
add : <x1, ...xn> =x1 +x2+ ... +xn
mul : <x1, ...xn> = x1 * x2 * ... * xn
distr : <<x1, ..., xn>, y> = <<x1, y>, ..., <xn, y>>
trans : <x1, ..., xn> = <y1, ..., yn>        ; yj is the sequenc
                                               of jth elements of xi 's
```

図1 FL プログラムの典型的な結合形式と基本関数
Fig. 1 Typical combining forms and primitive functions of FL

```
def mmul ≡ α:(α:IP)∘(α:distl)∘distr∘[s1, trans∘s2] where
(
    def IP ≡ add∘(α:mul)∘trans
).
```

図2 行列乗算 FL プログラム
Fig. 2 FL program Performing matrix multiplication

基本的に FL プログラムは一つの関数であり，通常この関数は複数の関数の結合関係から構成される．関数の結合関係は，基本関数を結合形式 (Combining form) によって結合することによって表される．図1に結合形式と基本関数の一部を示す．FL プログラムの例として，図2に二つの行列の乗算を行なうプログラムを示す．ここで，*def* 文節により定義された *mmul* が行列乗算を行うユーザ定義関数であり，同様に *IP* は2つのベクトルの内積を求める関数である．FL プログラムでは，関数を値に作用させて新しい値を得ることによって演算を進める．この書き換え操作を簡約と呼び，簡約される関数と値の対を可簡約項と呼ぶ．簡約は，最終結果としての正規形が得られるまで繰り返される．

FL プログラムでは，関数の結合関係と簡約に使用される値から，プログラム実行時の逐次性と並列性が決定される．逐次性は結合形式 "apply" に表れる．例として，"$f \circ g : x$" というプログラムでは，はじめに関数 "g" が値 "x" に作用することによって可簡約項 "$g : x$" の簡約が行われ，その結果に "f" が作用する．一方，並列性は結合形式 "cons"，"apply-to-all"，"cond" により現われる．例として，結合形式 "cons" によるプログラムを挙げる．"$[f, g, h] \circ k : x$" というプログラムを考える．まずはじめに，"$k : x$" が簡約される．その結果を "y" とすれば，次に "$[\, f, g, h\,] : y$" が "$f : y$"，"$g : y$" および "$h : y$" の3つの可簡約項に分割され簡約される．これらは，相互にデータの依存性がなく，並列に簡約が可能である．最後に，これらの並列簡約の結果が1つのリストとして統合され，プログラムの実行結果が得られる．

FL プログラムの関数の結合関係はグラフによって表現することができる．これを簡約グラフと呼ぶ．簡約グラフには，結合形式による関数の結合関係がノードの結合関係として表される．図3に，図2に示した行列乗算プログラムの簡約グラフを示す．"@" は関数の作用関係を表わすためのノードであり，左側子ノードである関数が右側子ノード関数の結果に作用する．

簡約グラフには，FL プログラムに現われる逐次性と並列性がノードの結合関係として表される．逐次性を表す結合形式 "apply" は "@" ノードが右側子ノードとして再び "@" ノードを持つことにより表される．従って，複数の逐次的な作用関係は複数の "@" ノードの右側子への連なりによって表現される．並列に簡約が可能な可簡約項は "*cons*"，"*apply-to-all*"，"*cond*" ノードの左側子に結合された部分グラフとして表現される．

可簡約項は左側子ノードに関数を表す部分グラフを持ち，右側子ノードに値を持つ "@" ノードとして表される．プログラム実行時の簡約は，可簡約項を表すグラフを，簡約を行なった後の結果の値に書き換える操作にあたる．関数型プログラムを実現する手法として広く用いられている並列グラフ簡約[5] は，概念的に簡約グラフに表される複数の可簡約項に対する並列な書き換え操作からなる．

並列グラフ簡約において，並列に簡約が可能な可簡約項の検出を "並列化"，同じ "*cons*"，"*apply-to-all*"，"*cond*" により生成された可簡約項の結果を一つのリストにまとめることを "統合" と呼ぶことにする．並列グラフ簡約では，根の "@" ノードから右側子へと連なった各 "@" ノードに対して，並列化と統合を繰り返しながら簡約を行なう．

並列グラフ簡約を並列計算機に実装することにより，関数型プログラムの高速実行が期待できる．

2.2 SPMD 実行モデル

現在，並列計算機における並列プログラミングはほとんどが SPMD モデルによる並列化を行なっており，この際に必要となるノード間の通信は MPI や MPL のようなメッセージ交換ライブラリによって提供されている．並列グラフ簡約では，並列化とその結果の統合を繰り返すことによってプログラムが並列実行される．この実行過程における並列化と統合は，メッセージ交換ライブラリの集合通信と対応することができる．SPMD モデルに基づいて並列化を行なう場合，並列計算機の全てのプロセッサが同一の簡約グラフとデータを持ち，各プロセッサが簡約グラフに基づいてデータの異なる部分について処理を行なう．各プロセッサで行なう処理を定義する部分グラフと，それと対応する部分データをタスクと呼ぶ．

並列グラフ簡約では，各プロセッサが行なう簡約の操作を集中して管理する必要がない．また，簡約の実行や並列化，統合にともなって必要となる通信は簡約グラフ

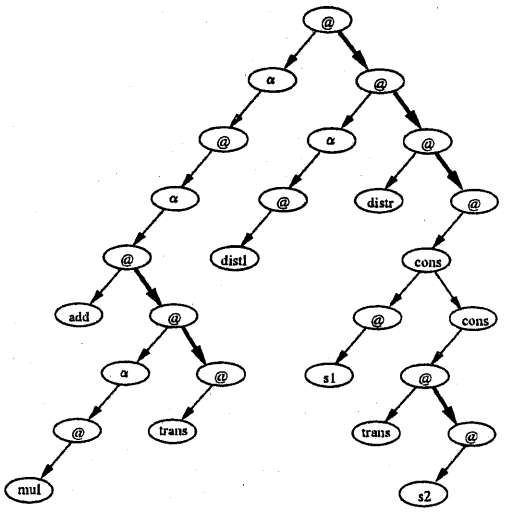

図3 行列乗算プログラムの簡約グラフ
Fig. 3 Reduction graph of the matrix multiplication program

を媒体として統一的に行なうことができる．従って，並列グラフ簡約を SPMD モデルを用いて並列化することは容易であり，更にその特徴から分散メモリ型の並列計算機上への実装に適していると考えられる．

以下，並列グラフ簡約を SPMD モデルにより並列化し，並列計算機上で実行する過程を示す．ここで，プロセッサ数を N とし，各プロセッサを $P_0, P_1, ...P_{N-1}$ とする．

まず，各プロセッサは簡約グラフとデータを持つ．簡約グラフには，その簡約に同時に使えるプロセッサのグループを対応させる．初期状態では，最初の簡約グラフに対して全プロセッサを含むグループ $G = \{P_0, P_1, ..., P_{N-1}\}$ が対応している．

次に，各プロセッサで簡約グラフの根のノードへと連なっている "@" ノードによって示される可簡約項の簡約を行なっていく．この際，n 個の可簡約項による並列化が可能である場合は次のように実行を行なう．

まず，可簡約項に対して $0, 1, ...n-1$ の番号をつける．この際，$N < n$ の場合は n 個の可簡約項をプロセッサ数 N で分割し，各可簡約項の組に対して再び 0 から $N-1$ の番号を割り当てる．

各プロセッサには，その番号と等しい番号を持つ可簡約項がタスクとして割り当てられる．例えば P_0 は 0 の番号の可簡約項をタスクとして持つことになる．

プロセッサが持つタスクの並列化が不可能な場合は，他のタスクが並列化可能な場合に備え，そのタスクをスタックに蓄え，タスクを持たない状態となる．タスクを持たないプロセッサには，更に並列化が可能なタスクがある場合，それが割り当てられる．また同時に，そのタスクを実行する新しいプロセッサのグループが作られる．このようなタスクに対しては，再帰的に同様な過程で実行を行なう．

例えば，並列実行が可能なタスクに対して割り当てられたプロセッサ数が $N = 3$ で，並列なタスクの数が $n = 3$，であり，0 番のタスクが更に並列化が可能な場合を考える．このとき，P_1 と P_2 はそれぞれに対応する番号 1, 2 のタスクを一旦スタックに蓄え，タスク 0 に対応するプロセッサのグループとして，$\{P_0, P_1, P_2\}$ が作られる．次に，タスク 0 に対して，このグループ内で並列化，簡約を行なう．

タスクを割り当てられなかったプロセッサはスタックからまだ実行していないタスクがあればとりだし，簡約する．簡約が終了したら，タスクに対応したグループの先頭のプロセッサに結果を集め，統合を行ない，その結果をグループ内のプロセッサ全てに送り，次の簡約を行なう．

このような処理を繰り返し，簡約グラフの根の "@" ノードの簡約が終了した時点で正規形が得られ，実行は終了する．

実行の過程の例として，図 2 および図 3 に示した行列乗算プログラムの一つめの可簡約項 $[s1, trans \circ s2]$ の簡約を挙げる．

行列として二行二列の行列の組を表すデータを D と仮定する．また，プロセッサ数を $N = 3$ とする．このとき，この可簡約項にはプロセッサのグループ $\{P_0, P_1, P_2\}$ が割り当てられる．"@" ノードは左側子として "$cons$" ノードを持つので，並列化が可能であることがわかり，"$cons$" ノード同士の右側子への結合をたどることによって，並列実行が可能なタスクの数は $n = 2$ であることがわかる．これより，並列実行可能な 2 つのタスク $s1 : D$ と $trans \circ s2 : D$ が検出でき，前者には 0，後者には 1 の番号が与えられる．

次に，プロセッサ P_0 は番号が 0 であるタスク $s1 : D$ を，プロセッサ P_1 は番号が 1 であるタスク $trans \circ s2 : D$ の簡約を行なう．タスクを持たないプロセッサ P_2 は何もしない．各プロセッサにおける簡約が終了したら，グループの先頭のプロセッサである P_0 に結果を集め，統合を行なう．0，1 のタスクの結果をそれぞれ D_1, D_2 とすると，プロセッサ P_0 では統合によって $< D_1, D_2 >$ という結果が得られる．並列実行の終了として，プロセッサ P_0 は統合によって得られた結果をグループ内の全てのプロセッサに通信し，全てのプロセッサが通信によって得た値を用いて次の "@" ノードの簡約を行なう．

3. 並列計算機への実装

3.1 実装環境

本研究では 2.2 節に示した並列グラフ簡約の SPMD モデルによる実行方式を，東北大学大学院情報科学研究科の並列計算機 IBM SP2 に実装した．

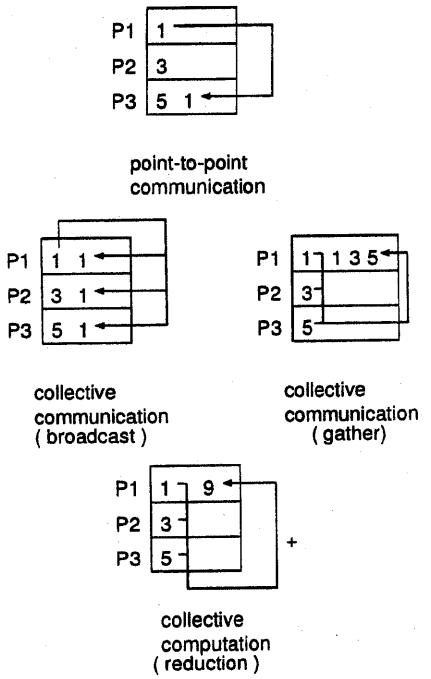

図4 メッセージ交換の例
Fig. 4 Examples of message passing subroutine

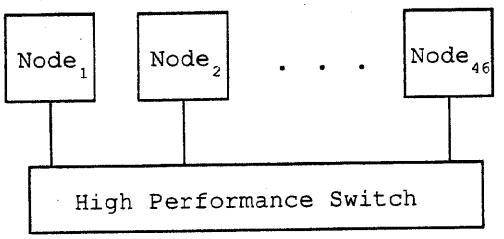

図5 東北大学大学院情報科学研究科 IBM SP2 の構成
Fig. 5 IBM SP2 at Graduate School of Information Sciences Tohoku University

IBM SP2 は分散メモリ型並列計算機であり，プロセッサ間の通信はメッセージ交換により行なう．メッセージ交換は MPI や MPL に代表されるメッセージ交換ライブラリによって供給される．その通信形式には，図4 に示すような 1 対 1 通信 (point-to-point communication)，集合通信 (collective communication) および集合演算通信 (collective computation) の三つがある．IBM SP2 における，これらの通信性能についての評価が行なわれている[4),6)] が，これによると，1 対 1 通信によって複数のプロセッサ間の通信を実現するよりも，集合通信を用いた方が，高速に通信処理が行なえることが報告されている．前述のように本研究で扱う並列グラフ簡約の SPMD モデルによる並列実行は，

表1 ベンチマークプログラム
Table 1 Benchmark program

プログラム	基本関数数	最大並列度
10x10	1214	1000
20x20	8824	8000
30x30	28834	27000
40x40	67244	64000
50x50	130054	125000
4Queens	1442	827
5Queens	6952	4083
6Queens	35768	21165
7Queens	170148	101215
8Queens	873318	520831

1 対 1 通信を必要としない．この点においても，本手法は分散メモリ型並列計算機への実装に適していると考えられる．

本研究で用いた並列計算機 IBM SP2 の構成を図5 に示す．この SP2 は 46 台のプロセッサと，これらを結合する多段スイッチングネットワークである HPS (High Perfornce Switch) で構成されている．

実装には，C++ 言語を用いた．また，各プロセッサ間の通信には，メッセージ交換ライブラリ MPL を使用した．MPL のメッセージ交換サブルーチンのうち，並列グラフ簡約における統合には，グループ内の部分結果を一つのプロセッサに集める *gather* を用い，その後に *bcast* によってグループ内の全てのプロセッサに統合した簡約結果の通信を行なった．

3.2 実験結果

実験には，図2 に示した行列乗算プログラムと，N-クイーン問題を解くプログラムをベンチマークプログラムとして使用した．行列乗算はデータの依存性が低く並列性が高いという特徴を持ち，N-クイーン問題はデータの依存性と並列性を兼ね備える．表1 にベンチマークプログラムの特性を示す．10x10 から 50x50 は行列乗算プログラムであり，4Queens から 8Queens は N-クイーン問題である．表1 において，基本関数数は，プログラムにおいて実行される基本関数の数であり，最大並列度は一度に並列実行が可能な基本関数の数を示す．

実験においては，各ベンチマークプログラムの実行時間を簡約処理時間と通信時間に分けて測定した．ベンチマークプログラムをそれぞれ 3 回実行を行ない，その平均時間を結果とした．プロセッサ数は 1，2，4，8，16，32 に変化して考察を行なった．表1 の最大並列度から，各ベンチマークはプロセッサ数に対して十分な並列度を持つと考えられる．

まず，並列化による効果を確かめるために，簡約処理時間について考察を行なった．図6 にその速度向上を示す．

図6 に示したように，プロセッサの台数の増加に応じて処理速度が向上していることがわかる．その中で，

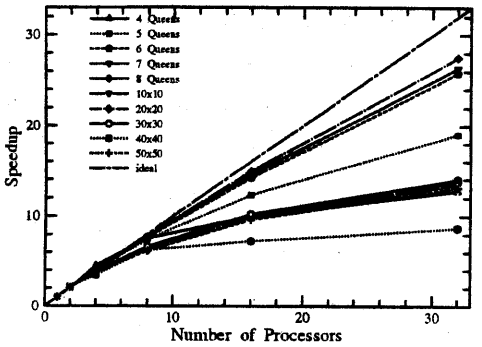

図6 1プロセッサからの簡約処理の速度向上

Fig. 6 Speedup of reduction time

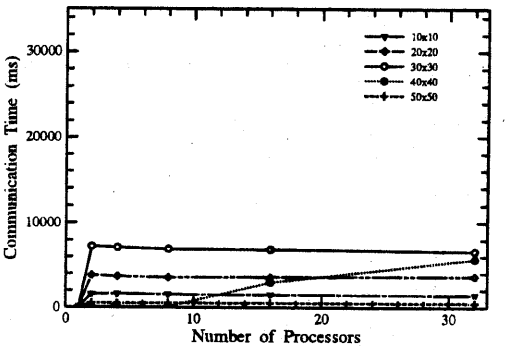

図7 行列乗算の通信時間

Fig. 7 Communication time of matlix multiplication

10x10 から 50x50 は，4Queens から 8Queens と比較すると速度向上が小さい．表1の基本関数数と最大並列度を比較すると，行列乗算のプログラムは基本関数のほとんどを並列に実行することができることがわかる．このため，行列乗算プログラムを実行する際，並列処理の粒度が小さくなり，簡約処理に必要な簡約グラフの走査や可簡約項の検出によるオーバーヘッドが生じるためと考えられる．一方 N-クイーン問題では，行列乗算プログラムと比べ，基本関数の並列化が可能である割合が小さい．その結果，N-クイーン問題の実行では並列処理の粒度が大きくなり，より良い速度の向上が得られたと考えられる．図6では，プロセッサ 32 台，8-クイーン問題において最大の速度向上を得られ，理想値 32 に近い約 27 の値を示す．これより，各プログラムの並列化は効果的に行なわれていると考えられる．

次に，各ベンチマークプログラムの通信時間を図7と図8に示す．図7および図8から，通信時間は行列乗算プログラムではほぼ一定の値であり，N-クイーン問題はプロセッサ数が増加すると通信時間が減少していることがわかる．

行列乗算プログラムの実行では，通信するデータが大きく，通信回数が少ない．また，一度に多くの並列化を行なう．一方，N-クイーン問題では通信するデータは行列乗算プログラムと比較すると小さいが，プログラム実行中に生じる通信の回数が多いという特徴がある．

プロセッサ数の増加に連れて，通信するデータが分割されることによって通信時間が小さくなると考えられるが，行列乗算プログラムでは，処理するデータの大きさが非常に大きいため，1 から 32 台程度のプロセッサ数による並列化では通信するデータの分割の効果が小さく，その結果プロセッサ数に対してほぼ一定の通信時間が必要となっている．

一方，N-クイーン問題においては，行列乗算プログラムほど大きくないデータを扱い，多数の並列化を行なうことによって通信データを分割した結果，プロセッサ数の増加に従って通信時間が短くなっていると考えられる．

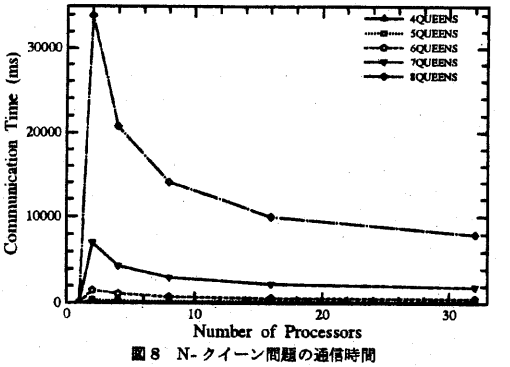

図8 N-クイーン問題の通信時間

Fig. 8 Communication time of n-queens problem

これらの特徴は，プログラムの性質に依存する．従って，多数の並列性と連した大きさのデータを扱うようなプログラムを記述することによって，並列処理による効果が更に得られると考えられる．

プログラム全体の実行時間について1プロセッサの実行時間に基づいて正規化して得られた速度向上を図9に示す．

図9を見ると，行列乗算プログラムではわずかな速度の向上しか得られていない．一方，N-クイーン問題では，プログラムに含まれる基本関数が多くなるほど大きな速度向上が得られている．行列乗算プログラムでは，並列処理の粒度の小ささのため，簡約処理の時間と比べ通信処理の時間が非常に大きくなることによって，並列処理により得られた速度向上を下回ったことが原因である．最大の速度向上はプロセッサ 32 台において 8-クイーン問題のプログラムを実行した時に得られている．しかしながら，図9に示した理想値と比較するとわかる

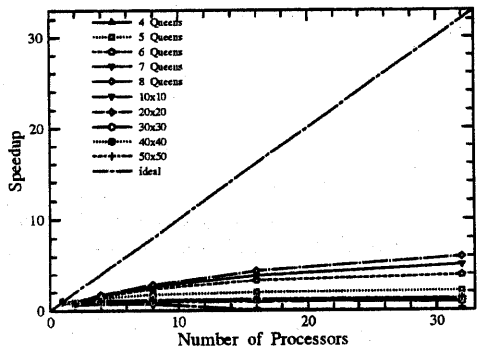

図 9　1 プロセッサからの速度向上
Fig. 9　Speedup

ように，理想値 32 より小さい 8 程度の値しか得られていない．

図 6 と図 9 を比較すると，通信のオーバーヘッドが大きな問題となることがわかる．同時に，並列処理の粒度が小さくなるようなプログラムでは，タスク検出などのオーバーヘッドが問題となる．本研究では，グラフを分割し並列化する手法の純粋な効果を考察するため，処理粒度の調整を行なっておらず，通信オーバーヘッドが大きく影響している．従って，並列処理の粒度を考慮に入れることにより性能を改善をしなければならない．また，図 7 および図 8 からプログラムの性質に依存する並列性によって並列処理の効果が変わるため，適したプログラムの記述を行なうことによって並列処理の効果がより現われることがわかった．

4. おわりに

本研究では，関数型言語の並列実行の容易性という特徴を利用して，並列グラフ簡約を SPMD モデルを用いて並列化を行ない，実行する手法を考察した．また，この手法を並列計算機 IBM SP2 に実装し，その評価を行なった．その結果，通信のオーバーヘッドのため十分な処理速度の向上は得られなかったものの，プログラムの並列化は効果的に行なわれていることがわかった．従って，通信に伴う処理を減少させることや，通信オーバーヘッドそのものを減少させることによって，処理速度の向上が望めると考えられる．

また，異なるベンチマークプログラムの実験結果の比較から，適した並列度およびデータの大きさのプログラムを記述することによって並列処理による効果がより現われることがわかった．

今後の課題として，並列処理の粒度を考慮に入れて並列化を行なうことによって通信オーバーヘッドを減少すること，より多くのベンチマークプログラムを用いて実験を行なうことによって効果的な並列処理が可能なプログラムの記述方法を考察することが挙げられる．

参考文献

1) Backus, J.: 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 (1978).
2) Backus, J., Williams, J. H. and Wimmers, E. L.: FL Language Manual (Preliminary Version), Research report, IBM Almaden Research Center (1986).
3) Backus, J., Williams, J. H., Wimmers, E. L., Lucas, P. and Aiken, A.: FL Language Manual, Parts 1 and 2, Research report, IBM Almaden Research Center (1989).
4) Hwang, K., Xu, Z. and Arakawa, M.: Benchmark Evaluation of the IBM SP2 for Parallel Signal Processing, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 7, No. 5, pp. 522–536 (1996).
5) Wadsworth, C. P.: *Semantics and pragmatics of the lambda calculus*, PhD Thesis, Oxford Univ., England (1971).
6) Xu, Z. and Hwang, K.: Modeling Communication Overhead: MPI and MPL Preformance on the IPM SP2, *IEEE Parallel and Distributed Technology*, Vol. 4, No. 1, pp. 9–23 (1996).