システムソフトウェアと
オペレーティング・システム 74−28
マルチメディア通信と分散処理 81−28
（1997. 2. 28）

データ並列言語におけるベクトルプロセッサ向きコード生成

幕田好久 貴島寿郎 梅村恭司
豊橋技術科学大学情報工学系
湯淺太一
京都大学大学院工学研究科情報工学専攻

概要

ベクトルプロセッサのデータ処理に対して逐次処理言語を用いる研究は多くなされているが，データ並列言語でデータを処理することもベクトルプロセッサの有効利用に対する一つのアプローチである．本研究ではベクトルプロセッサに対するデータ並列言語を用いたコード生成を考察した．原則としてデータ並列演算をベクトル化することにした．その原則に沿ったコード生成方針を示し，マトリクス乗算とLU分解についてベクトル処理可能なコード生成が実現できた．この実行時間を計測し通常のWSでの逐次実行と比較して，ベクトル化効果が得られることを確認した．

Code Generation of a Data-Parallel Language for Vector Processors

Yoshihisa Makuta, Toshiro Kijima, Kyoji Umemura
Department of Information and Computer Sciences, Toyohashi University of Technology
Taiichi Yuasa
Department of Information Science, Kyoto University

abstract

There have been done many studies on code generation of ordinary programming languages for vector processors. Using parallel languages is another approach to using vector processors effectively. This paper discusses code generation of a data-parallel language for vector processors. We propose the principle that data parallel operations should be vectorized. We describe a code generation method based on this principle, and show that we can successfully generate code for matrix multiplication and LU decomposition. We compare execution times for these applications on a vector processor with those on ordinary work-stations and show that both applications have been vectorized effectively.

1 はじめに

科学技術演算にはベクトルプロセッサを搭載したスーパーコンピュータが使用されることが多い．素子技術の向上でワンチップ化されたベクトルプロセッサも出現している．今後ますますベクトルプロセッサの需要は増大すると考えられ，その性能を発揮できる開発環境を整備する必要がある．

そこでベクトルプロセッサに対してもデータ処理をするための言語が必要である．しかし，一般的な逐次処理言語をベクトルプロセッサで使用するには，データをベクトル化するような変換が必要となる．この変換には様々な工夫がなされているが複雑であり，必ずしも成功するとは限らない．

一方でデータ並列言語があり，データを一つの固まりとして記述可能である．データ並列言語をベクトルプロセッサに対応させる場合，データの固まりを分割して演算を行なう処理となり，容易にベクトル化できることが予想される．

そこで本研究では，データ並列言語でどのようなベクトルプロセッサ用コードを生成すればよいかを検討する．原則として，データ並列演算をベクトル化することにする．以下では，ベクトルプロセッサとデータ並列言語を解説し，データ並列言語からベクトルプロセッサ向きのコードを生成する方針を示す．そして具体的な問題についてコード生成を行ない，ベクトル演算の数を検討し，プログラムの実行計測を行なった結果を報告する．

2 ベクトルプロセッサ

ベクトルプロセッサは複数のデータに対して同時に処理が可能なプロセッサである．

これらはSIMD型のプロセッサであり，ベクトルを構成するデータに対する演算を高速に実行するために，パイプライン処理方式を取り入れている．

ベクトルプロセッサはベクトルデータを格納するためにベクトルレジスタを持つ．ベクトルレジスタは複数個のレジスタが結合したレジスタで，ベクトル間の演算やベクトルとスカラとの演算の結果を格納する．また一部の要素だけ演算するような条件付きベクトル処理をするためにマスクレジスタが用意されている場合もある．

スーパーコンピュータ用ベクトルプロセッサ並の性能を持つワンチッププロセッサにμVP[1]がある．このプロセッサはベクトルレジスタとマスクレジスタを持ち，複数の演算パイプラインで構成されている．動作周波数は50MHzであり，最大で128要素の倍精度浮動小数点のベクトル演算が可能である．

本研究では，このμVPを搭載したスーパーパーソナルコンピュータATERLボード[2](モノリス社製)を使用した．このボードは汎用の外部接続バスを持つので，WS等からの制御が可能である．またμVPや外部機器からの制御が可能な2メガバイトのメモリを搭載している．

3 データ並列言語

データ並列言語は1命令で複数のデータに対する処理が可能な言語である．

データ並列言語の1つであるNCX[4]はデータ並列計算を主とする超並列計算のための拡張C言語である．NCXは言語構造上はSIMDモデルを採用しているが，SIMD型，SPMD型あるいはMIMD型のいずれの並列計算機に対しても利用可能な言語である．また，計算機の実プロセッサ数や物理的結合に依存しないコード化が可能である．

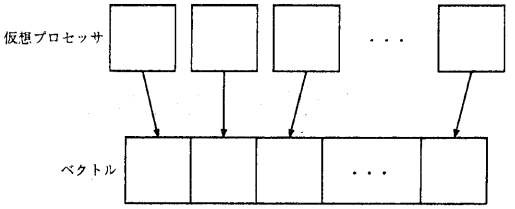

図1: 仮想プロセッサとベクトルデータの対応

NCXでは複数のデータを扱うために仮想プロセッサが用意されている．仮想プロセッサはNCXのプログラムを実行する主体である．原理的に無限個の仮想プロセッサの存在を仮定でき，任意の仮想プロセッサのみを操作するアクティビティ制御も可能である．また，仮想プロセッサが自分以外の仮想プロセッサに格納されている変数を参照するような遠隔参照の機能もある．

4 生成方針

NCXは柔軟な言語セマンティクスを持つので，ベクトルプロセッサに対してもNCXを実装することが可能である[5, 6]．そこでμVPをターゲットとしたコード化を考える．

まず，NCXで定義されている各仮想プロセッサを1本のベクトルとして考える．すると各仮想プロセッサ上のデータは，μVP上のベクトルレジスタで演算可能なベクトルデータとみなすことができる(図1)．

しかし，μVPのベクトルレジスタは8キロバイトしか用意されていないため，全てのデータを格納することができない場合がある．そこで，ベクトルデータをボードに搭載されたメモリ上にマッピングして，メモリとベクトルレジスタ間の転送を繰り返すことで実現する．

4.1 1次元の変数のベクトル化

1次元に配置された仮想プロセッサ上の変数をベクトル化する場合は，基本的に仮想プロセッサの数だけベクトル長をとる．

しかしベクトルレジスタには長さの制限があるので，一度にベクトル化できる仮想プロセッサ数に

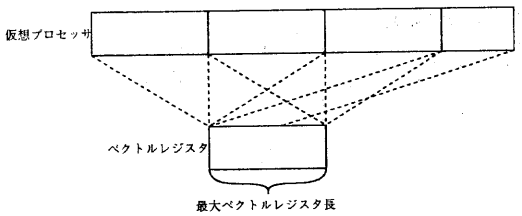

図 2: 仮想プロセッサの分割

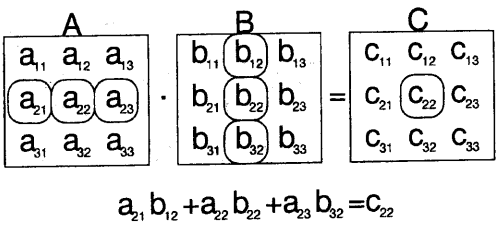

図 3: 逐次言語向けマトリクス乗算 (n=3 の場合)

は限度がある．そのため仮想プロセッサを最大ベクトルレジスタ長で区切って演算する必要がある (図 2)．その際に，区切りの先頭をマッピングアドレスとよぶことにする．

ここでは代入文

$$y = x + 1;$$

を例として説明する．x, y とも 1 次元に配置された仮想プロセッサ上の変数とし，仮想プロセッサ数が最大ベクトルレジスタ長よりも大きいと仮定する．

ベクトルデータ内のマッピングアドレスの相対位置を表すカウンタを用意する．以下の処理は，カウンタを最大ベクトルレジスタ長ずつ増加させながら，カウンタが仮想プロセッサ数を越えるまで繰り返される．

1. x のマッピングアドレスから最大ベクトルレジスタ長だけベクトルレジスタに読み込む．
2. 全てのベクトルレジスタにスカラ値 1 を加算する演算を行なう．
3. 演算結果を y のマッピングアドレスに格納する．

この方法は逐次言語で配列を扱う場合の拡張と考えることができる．逐次言語ではループカウンタが 1 ずつ増加していくが，並列処理で実現するためにカウンタを最大ベクトルレジスタ長ずつ増加するようにしたものである．

4.2 多次元の変数のベクトル化

仮想プロセッサ集合の次数が増えた場合に変数をベクトル化するときは，もっとも効率の良いインデックス方向にベクトル化し，余った次元はループで繰り返す．ここで作成したベクトルに対しては 1 次元変数と同様の処理が可能である．

ベクトル化する方向には，最大の要素数を持つインデックス方向を選択する．これにより一度にベクトル化できる要素数が多くなり，ベクトルレジスタを効率良く使用することができる．

例えば 5×10 の 2 次元に配置された仮想プロセッサ集合を考え，1 次元目のインデックスを i，2 次元目を j とする．この場合は 2 次元目の方が要素数が多いので，j の方向にベクトル化をする．

5 実問題に対するコード生成

実問題では仮想プロセッサ間の遠隔参照や仮想プロセッサのアクティビティ制御がある．これらが含まれる式では前節の方針のみでは実現できない．

以下ではこれらを適用しているマトリクス乗算と LU 分解について実際のコード生成を試みる．

5.1 マトリクス乗算

$n \times n$ の正方マトリクス A と B の乗算 ($C = A \cdot B$) をベクトルプロセッサで処理するための方法を考える．

逐次言語向けのマトリクス乗算 [7] は，A の行の要素と B の列の要素を取り出して乗算し，それらを足し合わせて C のある一つの要素を得るような処理をするのが一般的である (図 3)．

この方法は一見ベクトルデータとして演算できそうだが，この演算結果はスカラデータとなるので，本研究の原則から離れた処理になる．そこで並列言語向けの処理法を考察する．

この並列言語向けの方法はマトリクス乗算の規則性を利用して処理を並列に行なうものである．図 3から明らかなように，ある 1 つの要素を得るには n 個の項の加算が必要である．この項を C の全ての要素について集めて n 個の項マトリクスと考えれば，乗算結果は項マトリクスすべてを加算すると得られる．

以下ではこの方法を実際に μVP でコード生成した場合の流れで説明する．例として $n = 3(3 \times 3)$ のマトリクス同士の乗算を図 4〜図 6に示す．

表 1: マトリクス乗算の演算数

	load	store	add	mul
並列言語向けパイプライン使用数	$5n$	$n(n+1)$	$n(n+1)$	n^2
逐次言語向けパイプライン使用数	$n(n+1)$	n^2	n	n^2
スカラ処理での演算数	$n^2(2n+1)$	$2n^2$	n^3	n^3

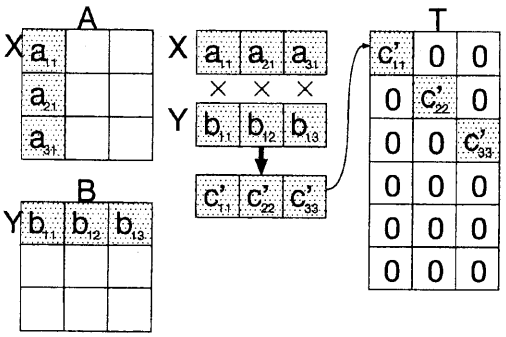

図 4: 並列言語向けマトリクス乗算 (1)

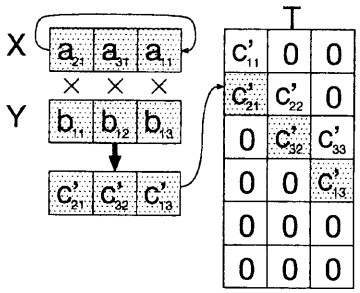

図 5: 並列言語向けマトリクス乗算 (2)

はじめに $2n \times n$ のテーブル T を確保し，ゼロクリアする．そして以下の操作を k について 1 から n まで繰り返す．

1. A の k 列目のベクトル X と B の k 行目のベクトル Y を取り出して，この両ベクトルを乗算し，テーブル T に保存する (図 4)．この保存先は実際の C の項マトリクス要素の位置に対応している．

2. 同様に A の列ベクトルをローテートしたものと B の行ベクトルを乗算したものもテーブル T に保存する (図 5)．この操作を合計 $n-1$ 回行なう．

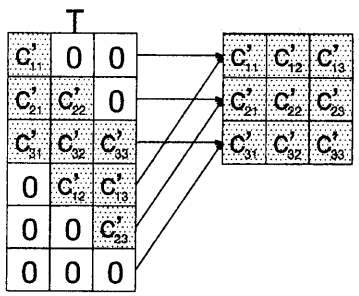

図 6: 並列言語向けマトリクス乗算 (3)

3. こうしてできたテーブルから C の項マトリクスを作成する (図 6)．

繰り返し終了後，それぞれの項マトリクスを足し合わせたものが C の要素となる．

n が最大ベクトル長よりも小さいときに，逐次言語向けの方法でベクトル化した場合と並列言語向けの方法でベクトル化した場合について，パイプラインの使用回数を比較する (表 1)．どちらの場合もベクトル化によりスカラ処理よりもオーダーが減少している．

並列言語向けの方法は逐次言語向けの方法と比べてロード回数のオーダーが一つ減少し，その分 ADD パイプラインの使用回数のオーダーが増加している．全体で比較すると，パイプラインの使用量はほぼ同じなので，本研究ではベクトルデータを保った方法である並列言語向けの方法を採用した．

5.2 LU 分解

n 元連立 1 次方程式を解くために LU 分解 [8] を使用する方法がある．LU 分解は方程式から作成した $n \times n$ のマトリクス A を下三角行列 L と上三角行列 U に分解する処理である．この LU 分解を実際にコード生成したときの流れを追って説明する．

まず L と U をゼロクリアし，L の主対角線上の要素に 1 を代入する．そして以下の 2 つの代入を k について 1 から n まで繰り返す．

表 2: LU 分解の演算数

	load	store	add	mul
ベクトル処理演算数	$3n^2+10n$	$3n^2+5n$	$3n^2+6n+1$	n^2+2n
スカラ処理演算数	$(4n^3+2n^2-13n)/6$	n^2+2n	$(2n^3-3n^2-5n)/6$	$(2n^3-6n^2-8n)/6$

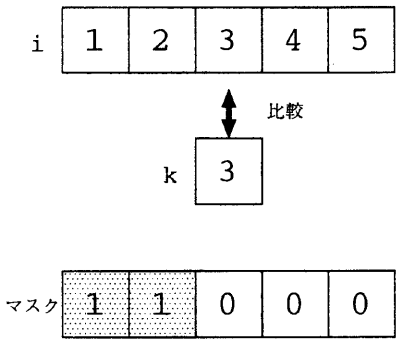

図 7: LU 分解時のマスク作成

$$l_{ik} := a_{ik} - \sum_{m=1}^{k-1} l_{im}u_{mk} \qquad (i = k, \ldots, n) \tag{1}$$

$$u_{kj} := (a_{kj} - \sum_{m=1}^{k-1} l_{km}u_{mj})/l_{kk} \quad (j = k+1, \ldots, n) \tag{2}$$

(1) は，i についてベクトル化をすればもっとも効率良く演算できる．しかし，i の範囲が k から n の間に限定されているので，仮想プロセッサのアクティビティ制御が必要となる．

そのためにベクトルプロセッサの機能であるマスク演算を使用する．まず仮想プロセッサ番号を要素とした長さ n のベクトルを用意する (図 7)．このベクトルと k を比較し，仮想プロセッサ番号 i が範囲外である場合にはマスクをかける．例えば k が 3 のとき，i が 1 と 2 の要素に対してマスクをかける (図 7)．

以降の演算ではこのマスクを使い，範囲外の要素については演算を行なわない．

$\sum$ の中の演算は，m について 1 から $k-1$ まで繰り返して処理する．l_{im} をベクトル，u_{mk} をスカラとしてメモリから読み込み，ベクトル演算をして足し込む (図 8)．$\sum$ の演算結果はベクトルである．

最後にベクトル a_{ik} から $\sum$ の結果を減算して右辺の演算は終了する．

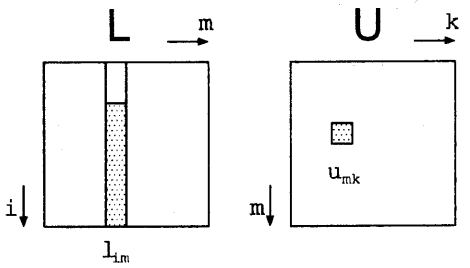

図 8: $\sum$ 内の演算

(2) も同様である．(2) については j についてベクトル化すると効率が良い．

n が最大ベクトルレジスタ長よりも小さい場合に，LU 分解で使用するパイプラインの回数を表 2に示す．LU 分解のアルゴリズムは $O(n^3)$ であるが，ベクトル化でオーダーが下がり $O(n^2)$ で処理ができる．

6 性能評価

前章でコード化した 2 つのプログラムを実際に μVP(50MHz) で実行し，従来の C で記述したプログラムと比較を行なう．比較対象として WS (SPARCstation 5: 110MHz, Memory 192Mbyte) を使用する．

両者は内部構造や動作周波数が違うため単純な比較はできないが，実時間情報は参考になるので汎用的なマシンである WS と比較することにする．

マトリクスの乗算については，μVP 版は本研究の方針に沿ってコード生成したもの，WS 版は逐次処理向けの演算法を C で記述したものを使用した．マトリクス乗算，LU 分解とも 128 × 128 のマトリクスで実行した．これらの結果を表 3に示す．

マトリクス乗算，LU 分解とも従来の WS の倍程度の性能が出ている．このことから，今回のコード化の方針は実用に耐え得るものであるといえる．

表 3: 実行時間の比較

	マトリクス乗算 [s]	LU 分解 [s]
μVP 版	0.4	0.3
WS 版	1.4	0.6

7 まとめ

データ並列言語でベクトルプロセッサに対応したコード生成処理を記述した．ここではデータを一つの固まりとして扱い，並列に処理することを原則とした．この原則に沿ったコード生成の方針を示し，マトリクスの乗算と LU 分解についてコード化を実現した．その結果この方針で生成したコードは，実用性があることを示せた．

参考文献

[1] 「パーソナルスーパーコンピュータユーザーズマニュアル」，株式会社モノリス，1992.

[2] 「シングルボードスーパーコンピュータプログラマーズリファレンスマニュアル」，株式会社モノリス，1992.

[3] 上村 和人，清水 俊幸，石畑 宏明，堀江 健志，「LINPACK ベンチマークの並列ベクトル処理」，並列処理シンポジウム JSPP'94，1994.

[4] 湯淺 太一 他，「超並列 C 言語 NCX 言語仕様書 (version 3)」，文部省重点領域研究「超並列原理に基づく情報処理基本体系」第 4 回シンポジウム予稿集，pp7-54，1994.

[5] 高橋 大介，「超並列 C 言語 NCX のベクトル化コンパイラの実現」，修士論文，豊橋技術科学大学湯淺研究室，1995.

[6] 渡邊 誠也，「SIMD 型超並列計算機用 NCX コンパイラの設計と実現」，修士論文，豊橋技術科学大学湯淺研究室，1995.

[7] G. Strung 著，山口昌哉他訳，「線形代数とその応用」，産業図書，pp.14-33，1978.

[8] 戸川 隼人，電子情報通信学会編，「数値計算法」，コロナ社，pp.104-124，1981.