計算機アーキテクチャ 117－8
（1996. 3. 5）

共有メモリ型並列計算機における多重ループステージングによるパイプライン実行

金丸 智一† 古関 聰† 小松 秀昭‡ 深澤 良彰†
†早稲田大学理工学部 ‡日本IBM(株) 東京基礎研究所

本稿では、共有メモリ型並列計算機のためのコンパイラにおける、新たな多重ループ並列化技法を提案する。ループ並列化において従来のループ本体をタスクとする方式は、依存制約の厳しいDOACROSS型ループに対しては十分な並列性を引き出せないという欠点があった。我々は命令レベル並列を利用したステージングという手法により、DOACROSS型ループの実行効率向上を図る。ステージングとはループ本体を構成する命令を複数の実行段階に分割し、そのパイプライン実行によって高速化を果たす方式である。本手法における、実行効率向上を考慮に入れたループ変換と、ループ本体中の命令を分割するアルゴリズムを示し、その評価を行う。

Pipelined Execution of Nested Loops by Staging Technique for Shared Memory Multiprocessors

Tomokazu Kanamaru† Akira Koseki† Hideaki Komatsu‡ and Yoshiaki Fukazawa†
†School of Science and Engineering, Waseda University
‡IBM Japan.Ltd. Tokyo Research Laboratory

This paper proposes a new loop parallelization technique for shared memory multiprocessors. For DOACROSS loops, existing methods that choose tasksize loop body cannot derive high parallelism. For high-speed execution of DOACROSS loops, a technique named 'staging' is proposed. Staging divides instructions in nested loop body into some execution class, and they are executed by pipelined parallelism. In this paper, we present a loop transformation and dividing instruction algorithms, and evaluate them finally.

1 はじめに

共有メモリ型並列計算機のための従来のループ並列化技法としては、ループ本体をタスクとしてイタレーション間の並列性を利用する方式がよく知られている。これはDOALL型ループに対しては非常に有益な手法であり、プロセッサ台数分の飛躍的な高速化が見込まれる。しかしイタレーション間依存（loop-carried dependence）が複雑に存在するDOACROSS型ループの場合、ことにその依存距離が短いケースでは、実行時にその部分が処理時間のボトルネックとなり、プロセッサ台数を増加させたとしても実行高速化を果たせない。DOACROSS型ループに対して高速化を達成するためには、より細かなタスク粒度である命令レベル並列の利用が必要になる。

我々は命令レベル並列を利用し実行高速化を達成するために、ステージングという新たなループ並列化技法を提案する。本手法では、ループ本体を構成する命令群を幾つかのステージ（実行段階）に分割し、元のループからステージを本体に持った複数のループを作る。プロセッサへの命令配置をこのループ単位で行い、各ループをオーバラップさせてパイプライン実行することにより高速化を果たすことが可能になる。

ステージングによる並列化はDOACROSS型ループに対しての高速化を目指したものである。DOALL型ループに対する従来の処理とステージングの利点を融合させ、大規模並列計算機上で利用できるループ並列化技法を構築するのが、本研究の最終的な目標であ

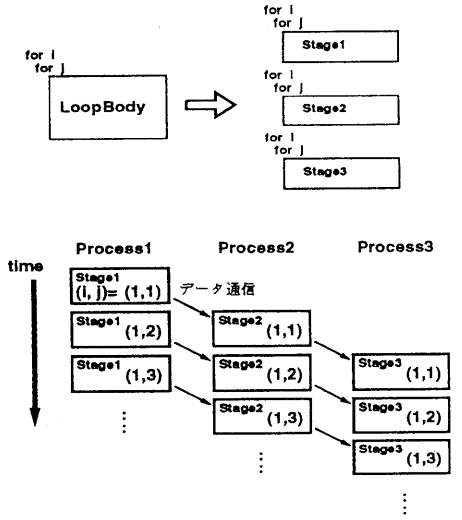

図 1: ステージングによる実行の概念図

る。本稿ではステージング技法に関して提案を行うとともに、その高速化のためのループ変換について説明し、それらのアルゴリズムを示す。

2 ステージングの概要

本手法では、まず、ループ本体を構成している命令を複数のステージ（実行段階）に分割する。この分割は、各ステージの持つ命令の実行処理時間ができる限り均等になるように行われる。また、分割の数は使用可能なプロセッサ資源数によって決定される。

次に元のループから、ステージを本体に持った複数のループを作る（図1上）。プロセッサへの命令配置はこのループ単位で行う。すなわち、各プロセッサは担当するステージのループのみを実行する。各プロセッサ上ではループイタレーションが順次実行されていくが、ステージ間に依存関係が存在する場合、それを保証するためにプロセッサ間で通信が行われる。この通信はループのイタレーション毎に規則的に発生する。

このように命令分割を行うことで、各プロセッサの持つループをオーバラップさせ、パイプライン並列を利用して実行することが可能である（図1下）。

本手法は、一つのループを複数に分割するという点でループ分割（loop distribution）の概念を含んでいる。ループ分割は、ループ本体から DOALL 型の実行が可能な部分を取り出すために用いられてきたが、ステートメント間に循環依存が存在する DOACROSS 型ループの場合、そこからそれ以上の並列性を引き出すことはできなかった。

ステージングはステートメント単位ではなく命令レベルでの依存関係を調べる。そのためより細かなレベルで DOALL、DOACROSS 型の部分を取り出し、ループを分割することができる。さらにステージングはパイプライン並列実行による高速化を前提とした手法であるため、ループ分割では並列実行できない循環依存を持った DOACROSS 型の部分からも、ループ変換を組み合わせて適用することで並列性を引き出すことができる。この点に関しては3章で詳しく述べる。以下では、ステージングを簡単な例を用いて説明する。

```
for i:=1 to IMAX do
  for j:=1 to JMAX do
  begin
    Y[i,j] := 1/3*(X[i-1,j]+X[i,j-1]+X[i,j]) ;
    Z[i,j+1] := (1-Y[i,j]+Z[i,j])/2 ;
  end
```

図 2にこのプログラムのループ本体に対するプログラム依存グラフ [2] を示す。

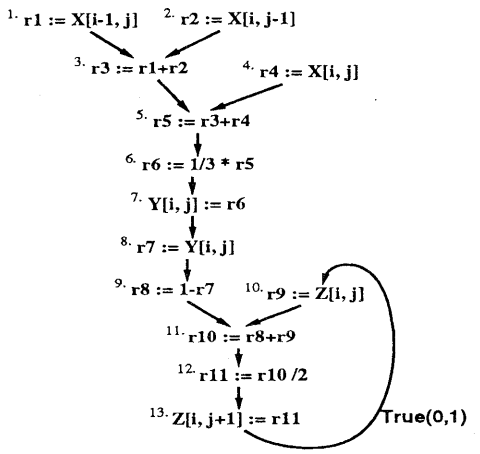

図 2: ループ本体の依存グラフ

グラフ中ラベル付けされていないエッジは、同じ繰り返しの中で発生するイタレーション内依存（loop-independent dependence）を意味している。一方、イタレーション間依存に関してはラベル表記を行う。True(d_i,d_j) という表記は、ループインデックス上、i 方向（i の値が増える方向）に d_i 回、j 方向（j の値が増える方向）に d_j 回後のイタレーションに対して真依存を持っていることを意味している。この依存距離の値の表示は、外側のループから順に左から右へと記していく。

この依存グラフをもとに、ループ本体を構成する命令をステージに分割する。仮に分割ステージ数を 3 とすると、図 2は例えば、

- stage[1]={1,2,3,4,5}
- stage[2]={6,7,8,9}
- stage[3]={10,11,12,13}

のように分割ができる。ここでstage[S]=$\{c_1, c_2, \cdots, c_k\}$という表現は、S番目のステージが命令$c_1, c_2, \cdots, c_k$を含んでいることを示している。

プロセッサ間のデータ通信はステージ間に跨って存在する依存に沿って発生する。この例の分割では、stage1-2間では命令5-6間の依存エッジ、stage2-3間では命令9-11間の依存エッジに沿って、それぞれデータ通信が発生する。この例での通信は各プロセッサ上、同じループインデックスを持つステージ間で行われることになる。

3 通信と実行のオーバラップ

3.1 パイプライン並列による高速化のための考慮点

パイプライン並列を利用した実行において高速化を達成するためには、パイプラインピッチの短縮が最も重要となる。しかしDOACROSS型ループのステージングには、プロセッサ間の通信・同期に伴い、パイプラインピッチを長くする様々な要因が存在している。本方式の性能を十分に発揮するためには、これらの影響を極力抑えなければならない。

我々が今回重視したのは、プロセッサ間通信と命令実行のオーバラップである。DOACROSS型ループ本体の命令を、各々の持つ処理時間が均等になるようにステージに分割しようとした場合、そのステージ間に厳しい依存制約が発生してしまい、パイプライン並列性を引き出せない場合が多い。この制約を緩和するために、イタレーション間依存の依存距離に注目する。依存距離とはその依存がどのくらい後の実行に対して制約を与えるかを示す値であるから、この値を大きく取ることができれば、通信時間の隠蔽、つまり通信と実行をオーバラップさせることが可能となる。

本手法では依存距離の調整のために、ループ変換技法としてユニモジュラ変換[1]を利用する。依存距離を変化させる場合、その依存がループネストのどのレベルに存在しているかという点が重要になる。互いに交換可能なループを多くし、例えば最外にするループを選択できれば有利である。ユニモジュラ変換を多重ループに適用することにより、イタレーション空間に張られる依存ベクトル(dependence vector)[1]の方向を変化させ、依存距離を長くすることが可能になる。

3.2 実行インターバル短縮のためのループ変換

ステージングでは、各プロセッサは担当するステージを繰り返し実行する。同一プロセッサにおいて、前のイタレーションの実行が終了してから次のイタレーションの実行が開始されるまでの時間を、本手法では実行インターバルと呼ぶことにする。この実行インターバルの長さは、ステージ間に存在する依存制約と密接な関連を持つ。

DOACROSS型多重ループの本体を依存グラフにした場合、イタレーション間依存によって循環依存が発生する場合がある。それを構成している命令ノードと依存エッジの集合を強連結部分(strongly connected component)[3]と呼ぶ。先例の図2の依存グラフでは、命令10,11,12,13と、それらを繋ぐエッジが一つの強連結部分を構成している。

命令分割において、ある強連結部分を構成している命令を2つ以上のステージに分割して配置すると、ステージ間に循環した通信が発生し、パイプライン並列実行を阻害する可能性が大きい。例として次のプログラムを考える。

```
for i:=1 to IMAX do
  for j:=1 to JMAX do
  begin
    Y[i,j] := 1/3*(X[i-1,j]+X[i,j-1]+X[i,j]) ;
    X[i,j] := (1-Y[i,j]+W[i,j])/2 ;
  end
```

この例における依存グラフを図3に示す。

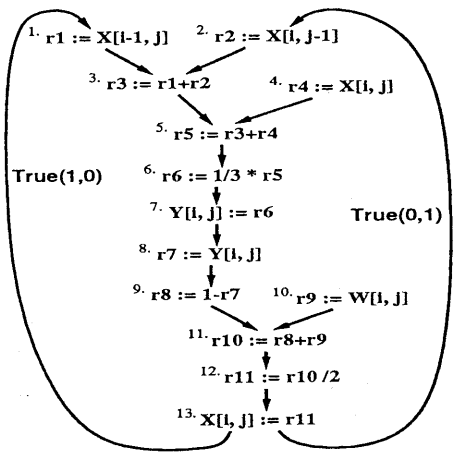

図3: 大きな強連結部分を持つ依存グラフ

この例では、命令1,2,3,5,6,7,8,9,11,12,13が一つの強連結部分を構成している。この場合にはステージへの命令分割は先のように単純にはいかない。強連結部分を無視し、先程と同様の方針でステージを作るとすると、例えば次のような分割が得られる。

- stage[1]={1,2,3,4,5}
- stage[2]={6,7,8,9}
- stage[3]={10,11,12,13}

これによりTrue(0,1)とTrue(1,0)というイタレーション間依存に沿って、stage3からstage1への通信

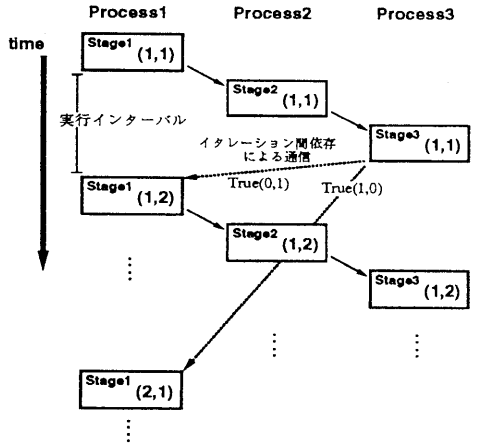

図 4: 依存距離が短い場合のステージング

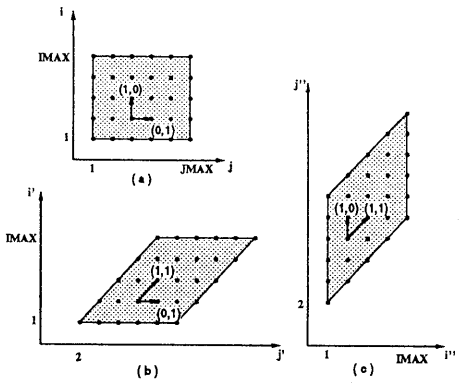

図 5: イタレーション空間中の依存ベクトル

が発生する。特に True(0,1) という依存のため、命令 13 で生産されたデータは、j に関して +1、つまりループインデックス上わずか 1 回後のイタレーションの命令 2 で消費される。この制約のため stage1 の実行はインデックス上 1 回前の stage3 の終了を待たなくてはならず、図 4で見られるように実行インターバルが大きく空く。結果として、強連結部分を分割して命令を配置したために、パイプライン並列による高速化は阻害されてしまうことになる。

しかし、複雑な依存を持つ DOACROSS 型ループは、依存グラフ中で大きな処理時間を占める強連結部分が存在するケースが多く、そのような場合には強連結部分の命令を分割して並列性を取り出すことを考えなければ、ループの高速化には繋がらない。

本手法ではこれらの依存距離を長くするために、以下のような方針でループ変換を適用する。この例で強連結部分を構成しているイタレーション間依存は、イタレーション空間中、$\vec{d_a} = (1,0)$、$\vec{d_b} = (0,1)$ という 2 つの依存ベクトルで表現できる (図 5(a)) 。

まず、最内ループに関して、全依存ベクトル中のその方向要素の値を正にする。このためにスキューイングを用いる。これはイタレーション空間の座標系を変換する。$\vec{d_a}$ と $\vec{d_b}$ の j 方向要素を正値にするために、次の行列形式で示される新たな座標系を導入する。

$$\begin{bmatrix} i' \\ j' \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ f & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix}$$

この変換によって依存ベクトル (d_i, d_j) は (d_i', d_j') $= (d_i, d_j + f d_i)$ に変化する。ここで f はスキュー係数と呼ばれる定数である。この変換は依存ベクトルの参照順序の整合性を保持し、プログラムの意味を保存する。図 3の例は f=1 で次のループに変換される。

```
for i':=1 to IMAX do
  for j':=i'+1 to i'+JMAX do
  begin
    Y[i',j'-i'] :=
        1/3*(X[i'-1,j'-i']+X[i',j'-i'-1]+X[i',j'-i']) ;
    X[i',j'-i'] := (1-Y[i',j'-i']+W[i',j'-i'])/2 ;
  end
```

この変換で依存ベクトル $\vec{d_a}$ と $\vec{d_b}$ はそれぞれ、$\vec{d_a'} = (1,1)$、$\vec{d_b'} = (0,1)$ に変化する (図 5(b)) 。

次に、方向要素が正になったループに関して、そのループを外側に移動する。このためにループ交換を用いる。これは行列形式で次のように示される。

$$\begin{bmatrix} i'' \\ j'' \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} i' \\ j' \end{bmatrix}$$

```
for j'':=2 to JMAX+IMAX do
  for i'':= max(1,j''-JMAX) to min(j''-1,IMAX) do
  begin
    Y[i',j'-i'] :=
        1/3*(X[i'-1,j'-i']+X[i',j'-i'-1]+X[i',j'-i']) ;
    X[i',j'-i'] := (1-Y[i',j'-i']+W[i',j'-i'])/2 ;
  end
```

元の i',j' ループは可換であるから、この変換は正当でありプログラムの意味を保存する。依存ベクトル $\vec{d_a'}$ と $\vec{d_b'}$ は、それぞれ $\vec{d_a''} = (1,1)$、$\vec{d_b''} = (1,0)$ に変化する (図 5(c)) 。

このようなループ変換を行った後に、先と同じ分割でステージを作る。命令 13 で生産されたデータが消費されるのは、少なくとも外側ループ j'' に関して +1 後に実行されるイタレーション内の命令 2 である。依存距離が十分に長い場合のステージングによる実行の例を図 6に示す。このようにステージ間に跨るイタレーション間依存の依存距離を大きくとることにより、通信時間を隠蔽して無用な実行インターバルをなくした実行が可能になる。

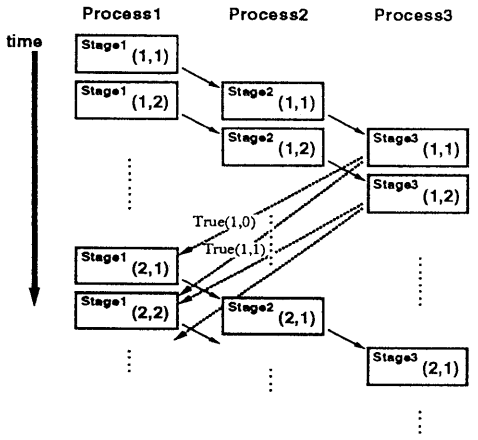

図 6: 依存距離が長い場合のステージング

4 命令分割の決定アルゴリズム

n 重ループ $L = (l_1, l_2, \cdots, l_n)$ のループ本体を構成する依存グラフから、命令をプロセッサ資源数と同数のクラスタに、各々ができる限り均等な処理時間を持つように分割するアルゴリズムについて説明する。

アルゴリズムは次の 3 つのステップに分かれる。

・ステップ 1 : 依存グラフの特性の判定
・ステップ 2 : 強連結部分の縮小
・ステップ 3 : クラスタの作成

クラスタ作成の方針として、同じ強連結部分を構成している命令は同じクラスタに分割する。しかし, 依存グラフ中に大きな処理時間を占める強連結部分が存在する場合には、そのままでは均等な命令分割ができない。ステップ 1 ではその判定を行う。

グラフ中に大きな強連結部分が存在すると認められた場合、アルゴリズムはステップ 2 を行う。このステップではループ変換を用い、強連結部分を構成しているイタレーション間依存の依存距離を大きくすることで、強連結部分の（疑似的な）縮小を行う。

ステップ 3 では、依存グラフ中の命令からクラスタを作成する。これはグラフ中の命令ノードの融合操作を繰り返すことによって行われる。P 個のクラスタが得られた時点でアルゴリズムを終了する。

本アルゴリズムは、パラメータとして、使用可能なプロセッサ資源数 P と、依存グラフ中の全ての命令の処理時間の総計 T_{all} を考慮する。

4.1 依存グラフの特性の判定

このステップでは、均等な分割のために障害になるような大きな強連結部分が依存グラフ中に存在しているか否かを判定する。強連結部分を発見する方法としては、Tarjan のアルゴリズム [3] が知られている。これによって得られる、最も大きな強連結部分の処理時間を T_{max} とする。ここでは判定のために以下の式を用いる。

$$T_{max} \leq T_{all}/P \qquad \cdots\cdots (1)$$

依存グラフが式 (1) を満たしていれば、均等な分割のために障害になる強連結部分はないと判断し、ステップ 3 へ進む。そうでなければステップ 2 を行う。

また、多重ループネストでない単一ループには、ステップ 2 のループ変換は適用できない（依存距離を大きくする変換を行うことができない）ので、式 (1) にかかわらずステップ 3 へ進む。

4.2 強連結部分の縮小

このステップでは、イタレーション間依存エッジの除去操作により、依存グラフの強連結部分の縮小を行う。ここでいう依存エッジの除去操作とは、エッジの出向側のノードにそのノードが生産したデータを送る send 命令、エッジの入向側のノードにエッジが運搬するデータを受け取る receive 命令を付加することである。つまりエッジを通信命令に置き換えることでこのエッジを疑似的に除去できる。

このステップでは、まず除去するイタレーション間依存の集合 D を先に決定する（ループ本体中に存在する全イタレーション間依存の依存ベクトルの集合を C とすると、$D \subseteq C$）。その後、D に属する全ての依存の依存距離が長くなるようにループ変換を行う。ここで、依存ベクトル $\vec{d} = (d_1, d_2, \cdots, d_n)$ において、零値でない最上位成分の次数をその依存の依存レベルと呼ぶことにする。変換は、ループネストの中から変換対象ループ l_k を選択し、D に属する依存の全ての依存レベルを k 以下（l_k より外側ループのレベル）にするスキューイングを繰り返しながら、ループ交換によって変換対象ループを順次外側ループへと移動させていく。この手順を、変換対象ループが最外になるまで繰り返す。

4.3 クラスタの作成

このステップでは依存グラフ中の全命令を P 個のクラスタに分割する（P 個のステージを作成する）。ステップの最初に、依存グラフ中に存在する強連結部分を全て 1 ノードに置き換える操作を行う。これにより対象依存グラフは無閉路有向グラフとなる。この操作によって生じるノードの個数を m とする。初期状態としてこのグラフは m 個のクラスタに分割されている。

ここで $m \leq P$ の場合、これ以上の操作は行わず、P 台のプロセッサに対する m 個のクラスタが得られたとして、アルゴリズムを終了する。

$m>P$ の場合、グラフ中のノードの融合操作を繰り返す。ノードaとノードbの融合操作とは、有向辺(a,b)または(b,a)が存在する時に有効で、ノードa,bとこれらを繋ぐ有向辺（融合エッジと呼ぶことにする）を一つのノードに置き換える操作のことである。これにより各ノードに属していた命令は新たな一つのノードに属することになる。これをノードの総数が P になるまで繰り返す。具体的な手順を以下に示す。

(1) グラフ中の全ノードに番号付け（$1 \sim m$）を行い、各ノードに重み付けを行う。そのノードに含まれる全命令の合計処理時間をそのノードの重みとする。ノード k の重みを w_k と書く。

(2) グラフ中の全ノードにトポロジカルなレベル付けを行う。ノード k のレベル v_k は、次の規則によって算出される。

  (a) 入向辺を持たないノード k については $v_k=1$

  (b) ノード k が1つ以上の入向辺を持つ時、それらに繋がるノードを $a, b, \cdots$ とすると、$v_k=max(v_a, v_b, \cdots)+1$

(3) グラフ中から融合するノードの組を選択する。選択可能性はグラフ中のエッジの数だけ存在する。ただし、そのエッジに繋がる2ノードを融合することでグラフが無閉路でなくなる場合がある。そのようなエッジは選択候補から除外する。融合エッジ (a, b) の選択基準は次の通りとする

  (a) $w_a+w_b < T_{all}/P$ を満たし、w_a+w_b が最大であるエッジ。ただし、グラフ中にこれを満たすエッジが一つもない場合には、w_a+w_b が最小であるエッジを選択する。

  (b) 条件(a)において候補が複数存在する場合、$min(v_a, v_b)$ が最小のエッジ。

  (c) さらに複数存在する場合、$max(v_a, v_b)$ が最小のエッジ。

  (d) それでも複数存在する場合、任意のエッジ。

(4) ノード a,b の融合操作を行う。新たに得られたグラフに関して、$m=P$ になったらアルゴリズムを終了。そうでなければ(1)に戻る。

5 評価

本稿で提案したアルゴリズムをDOACROSS型多重ループに適用し、どの程度の高速化が見込めるかを調べた。評価に用いたプログラムは、2次元配列に計算を行う簡単なSOR法のコードである。

表1に実行性能の評価を示す。ここで言う実行性能とは、対象コードに本手法を適用し、得られたクラスタ中で処理時間が最大となるものを調べ、スカラプロセッサ1台による実行に要する時間と比較して何倍の高速化が得られるかを表したものである。命令処理時間は、通常の演算命令を1、ローカルメモリへのload,storeを2とし、プロセッサ間通信のためのsend,receive命令に実行に要する時間をcとする。プロセッサ資源数 P を2～7、通信命令の処理時間cを0、2、4とした時のそれぞれについて評価を行った。

表1: SORコードに対する実行性能見積り

P	2	3	4	5	6	7
c=0	1.769	2.091	2.875	2.875	4.600	4.600
2	1.353	1.533	1.917	1.917	2.556	2.556
4	1.095	1.210	1.438	1.438	1.769	1.769

プロセッサ台数がいずれの場合も、cの値が大きくなると極端な性能低下が見られる。現行の方式では、プロセッサ間通信がイタレーション単位で発生するため、通信命令実行に要するコストは、パイプラインピッチを長くする原因となる。これを抑えるためには、イタレーション複数回分の通信をまとめて行うループブロッキングなどの手法を適用することが必要になる。ループ変換のアルゴリズムに、通信のベクトル化のための手法を組み合わせて改良していくことが、これからの課題と思われる。

6 おわりに

本稿では、共有メモリ型並列計算機上で多重ループを並列化するステージング技法と、その効率を上昇させるループ変換技法について述べた。

今後、DOALL型ループに対する従来の処理と本方式の利点を組み合わせたアルゴリズム、通信のベクトル化を考慮に入れたループ変換などについて研究を進めていく予定である。

参考文献

[1] M.E.Wolf and M.S.Lam, "A Loop Transformation Theory and an Algorithm to Maximize Parallelism", IEEE Trans. Parallel and Distributed Systems, Vol.2, No.4, pp.452-471(1991).

[2] J.Ferrante, K.J.Ottenstein and J.D.Warren, "The Program Dependence Graph and Its Use in Optimization", ACM Trans. Programming Languages and Systems, Vol.9, No.3, pp.319-349(1987).

[3] R.Tarjan, "Depth first search and linear graph algorithms", SIAM J.Computing, Vol.1, pp.146-160(1972).

[4] 古関, 小松, 深澤, '多重ループにおける最適ループ展開数算定技法とその評価', 並列処理シンポジウム JSPP'95 論文集, pp.193-200 (1995).