多重ループステージングにおける通信遅延隠蔽技法

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

ステージングは、共有メモリ型並列計算機のためのコンパイラにおけるループ並列化技法である。これ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、それらのパイプライン実行によって高速化を果たす方式であり、これにより従来のループイタレーションをタスクとする方式では並列性を引き出すことが困難であったDOACROSS型ループを高速実行させることが可能になる。本稿では、ステージングを用いた並列実行における、通信と計算のオーバラップの効果について説明し、そのためのループ変換アルゴリズムについて述べ、その評価を行なう。

Overlapping Communication with Computation in Staging Technique

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

We proposed a new loop parallelization technique named 'staging' for shared memory multiprocessors. Staging divides instructions in loop body into some execution classes, and they are executed with pipelined parallelism. This technique is proposed for the high-speed execution of DOACROSS loops, for which existing methods that choose loop bodies as tasksize cannot derive high parallelism. In this paper, we describe an effect of overlapping communication with computation for staging technique, present a loop transformation algorithm, and evaluate it finally.

1 はじめに

共有メモリ型並列計算機のためのコンパイラにおける従来のループ並列化技法としては、ループイタレーション単位をタスクとする並列処理が一般的によく用いられる。しかしこの方式は、ループ運搬依存（loop-carried dependence）が複雑に存在するDOACROSS型ループの場合には、並列性を十分に利用した実行ができないという欠点を持っていた。

我々はこのようなループに対して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した[6]。これは従来のようにループイタレーション単位で並列性を引き出す方式ではなく、ループ本体を構成する命令を複数の実行段階に分割し、それらをパイプライン実行することによって並列性を引き出す方式である。これにより従来の方式では効率を上げることが困難であった、複雑なループ運搬依存を持つDOACROSS型ループを高速実行させることが可能になる。

ステージングによる高速化の効果を最大限に引き出すためには、通信と計算のオーバラップによる通信遅延の隠蔽、通信のベクトル化といった手法が必須となる。特に本手法は、複雑な依存を持つDOACROSS型ループへの適用を意図したものであり、このようなループの実行では、プロセッサ間通信遅延の隠蔽が困難になる場合が多い。本稿では、ステージングを用いた並列実行における、通信と計算のオーバラップのためのループ変換アルゴリズムを示す。

2 本研究の背景

2.1 従来のDOACROSS型ループの並列実行

複雑なループ運搬依存（loop-carried dependence）が存在するDOACROSS型ループ、例えば複数次元にわたって依存関係が存在するような多重ループは、通常のDOALL型の実行は不可能であるため、プロセッサ間通信を用いながら並列実行される。この際、従来はループイタレーション単位をタスクサイズとする方式が用いられてきた。

図1のプログラムはDOACROSS型多重ループの例

```
for i:=1 to IMAX do
  for j:=1 to JMAX do
  begin
    A[i, j] := A[i, j-1] ;
    B[i, j] := (A[i-1, j]+2)*w ;
    C[i, j] := B[i+1, j-1]+z*C[i, j-2];
  end
```

図 1: サンプルプログラム

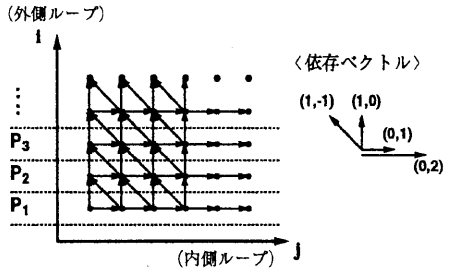

図 2: イタレーション空間の分割

である。このループのイタレーションの集合は、図 2に示すような 2 次元のイタレーション空間を構成する。個々のイタレーションは点で示されている。このとき、ループ運搬依存はイタレーション空間上、依存ベクトル (dependence vector) [3] として表現される。本稿では依存ベクトルを、各ループ方向要素の数値の組として、外側ループ方向から順番に並べ記述することにする。上例のループは $(0,1),(0,2),(1,0),(1,-1)$ という依存ベクトルを持っている。

これを並列実行する場合、従来の方式では、イタレーション空間をある次元で分割し (図 2で示した例ではループ i について分割) 、分割単位毎に各プロセッサ $P_1,P_2,\cdots$へと割り付けを行なう。ループ運搬依存による実行順序制約の都合上、場合によっては依存ベクトルの方向を変化させるためのプログラム変換が用いられる。これまでの研究では、タイリングとイタレーション空間の座標変換 (スキューイング) を組み合せたイタレーション空間の分割 [3] や、並列性と局所参照性を最適にする命令・データ分割を求める方法 [4] などが提案されてきた。

しかしこのようにイタレーション単位でプロセッサへの割り付けを行う方式は、全ての DOACROSS 型ループに対して必ずしも良い性能を与えることができるわけではない。これは次のような理由による。

1. ループ中にループ運搬依存が多次元にわたって複数存在している場合、またはコンパイル時に正確な依存解析ができないループ運搬依存が存在している場合、どのようにイタレーション空間の分割を行なってもプロセッサ間で発生する通信が繁雑になり、並列実行ができない。

2. イタレーション空間の座標変換 (スキューイング) を用いた後、イタレーション空間を分割してプロセッサに割り当てると、一つのキャッシュラインに複数のプロセッサが書き込みを行う場合が起こってしまう。この場合複数のキャッシュが無効化要求 (invalidation) を発行しあうため、高速実行できない。

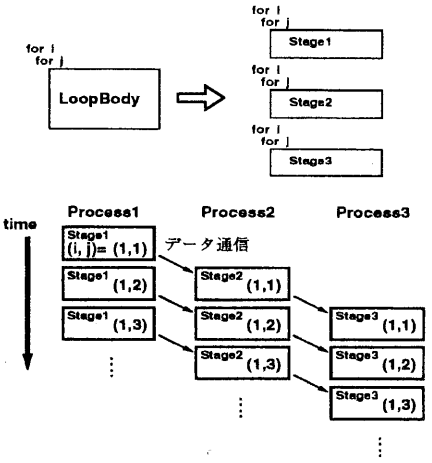

図 3: ステージングによる実行の概念図

2.2 ステージングによる並列実行

我々はこのようなループに対して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した [6]。

ステージングでは、ループ本体を構成している命令群を、ステージと呼ばれる幾つかの実行段階に分割する。そして元のループから、それぞれのステージを本体としたループを作る (図 3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。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。本手法による実行の様子は図 3下に示すようになる。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の例における、ループ本体を構成する命令のプログラム依存グラフ [1] を図 4に示す。図においてループ運搬依存を表すエッジには、その依存がイタレーション空間上で構成する依存ベクトルの値を付記している。図 4のように 3 つのステージに命令分割を行った場合、プロセッサ間通信は、stage1-2 間では命令 3-4、stage2-3 間では命令 7-10 の間にある依存関係に関して行なわれる。

ステージング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) の手法を含んでいる。従来のイタレーション空間を分割する並列処理に比べ、ステージングは次の利点を持っている。

1. プロセッサ間通信を、ステージ間に跨って存在する依存関係を保証するためにのみ行なえばよい。

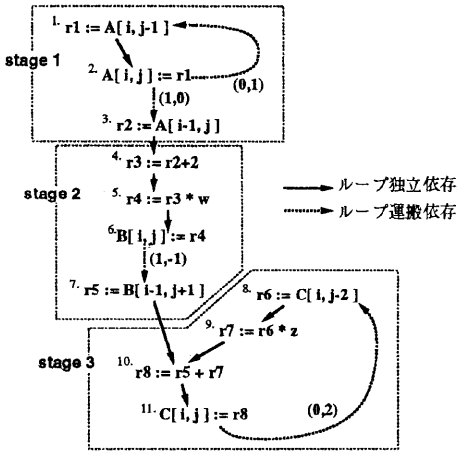

図 4: ステージ分割の例

つまり、必ずしもループ運搬依存について通信を行う必要はなく、命令分割次第で、プロセッサ間通信させるデータを任意に選択することができる。これは前出 1 の問題点を解決する。

2. 同一の命令は同じプロセッサで実行されるため、イタレーション空間の座標変換に関係なく、キャッシュラインに書き込みを行うプロセッサを一つに定めやすい。これは前出 2 の問題点を解決する。

3 パイプライン並列による高速化のための考慮点

一般に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になる。ここで言うパイプラインピッチとは、ステージを構成する命令群の実行時間以外に、参照命令に伴うキャッシュ制御、プロセッサ間の通信・同期に要する時間等を含んでいる。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。本研究では以下の点を重視する。

1. 全ステージ中最大になる実行時間の値をできるだけ小さくする。つまり、各ステージの実行時間ができるだけ均等になるように命令分割を行う。

2. キャッシュコヒーレンス制御に要するコストの最小化。キャッシュ上近隣に位置するデータを参照する命令群は、同じプロセッサで実行する。

3. 通信のベクトル化。プロセッサ間でのデータ通信を複数イタレーション回毎にまとめて行う。

4. 通信と計算のオーバラップ。プロセッサ間通信によって起こる遅延を隠蔽する。

1,2 については命令分割のアルゴリズム [6] で考慮する。2 に関して、同じ配列を参照する命令が近隣のデータを参照するか否かは、コンパイル時に配列添字を解析することによって判別することができる [5]。例えば、データが下位次数優先順でストアされていくと仮定すると、図 4の例では、命令 1 と命令 2、命令 8 と命令 11 が、近隣のデータを参照する関係にある。このような命令は、命令分割のアルゴリズム上、優先的に同じステージに配置するようにしている。

3 についてはループブロッキングを適用する。この段階の詳細は本稿では省略する。

4 に関しては、命令分割の前段階にループ変換を適用することで実現する。これについては次章で詳しく説明する。

4 通信と計算のオーバラップの効果

ステージングにおいて、各ステージの実行処理時間が均等になるように命令分割を行なうことは、高い実行速度向上を得る上で重要である。しかしそうした場合、DOACROSS 型ループの中には、ステージ間に循環依存が発生してしまうケースが存在する。この循環依存は、プロセッサ間通信遅延の隠蔽を困難にし、並列実行を大きく阻害する。

この問題を解決するために本研究では、ステージ間で循環依存を構成しているループ運搬依存の依存距離に着目している。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。この値が十分に大きければ、通信と計算をオーバラップさせ、通信遅延を隠蔽することが可能になる。

本手法では、依存距離の値を変化させるために、ループ変換技法としてユニモジュラ変換 [3] を用いる。依存距離の値は、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に大きく影響を受ける。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上の依存ベクトルの方向を変化させ、依存距離を大きくすることができる。通信と計算のオーバラップによる高速化の具体例を以下に示す。

4.1 循環依存を持つ DOACROSS 型ループの実行

DOACROSS 型多重ループの本体を依存グラフにした場合、ループ運搬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は、依存グラフ中で強連結部分 (strongly connected component) [2] を構成する。

```
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]) ;
    X[i, j] := (1-Y[i, j]+W[i, j])/2 ;
  end
```

図 5: サンプルプログラム (SOR 法)

図 5に示したプログラムのループ本体を構成する依存グラフを図 6に示す。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。

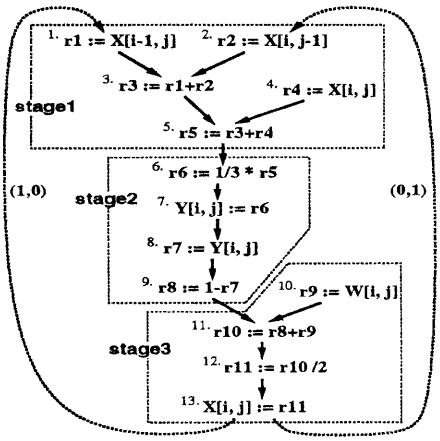

図 6: 大きな強連結部分を持つ依存グラフ

強連結部分を構成している命令群を 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。

この例で、図 6のようなステージ分割を行ったとすると、実行の様子は図 7に示すようになる。ここで $(0,1)$ と $(1,0)$ という依存ベクトルを持つループ運搬依存に沿って、stage3 から stage1 への通信が発生する。特に $(0,1)$ 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 7で見られるように stage1 ループのイタレーションの実行間隔（実行インターバルと呼ぶ）が長くなる。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、高速実行が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い。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つながらない。

4.2 通信遅延隠蔽のためのループ変換

本手法では、ループ運搬依存の依存距離を大きくするために、以下のような方針でループ変換を適用する。この例でステージ間の循環依存を構成しているループ運搬依存は、イタレーション空間上、$\vec{d_a}$=(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 8(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、

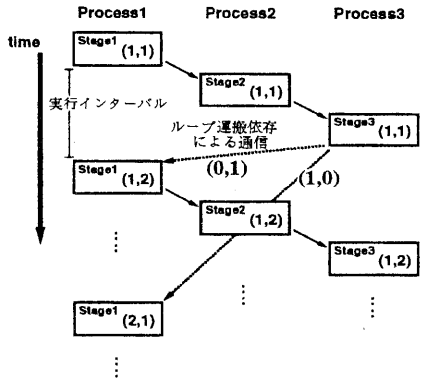

図 7: 依存距離の値が小さい場合のステージング

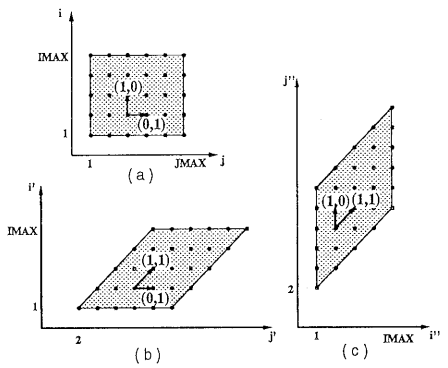

図 8: イタレーション空間中の依存ベクトル

次の行列形式で示される新たな座標系を導入する。

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

これにより依存ベクトル (d_i,d_j) は $(d'_i,d'_j) = (d_i, d_j + sd_i)$ に変換される。ここで s 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 5の例は s=1 で図 9に示す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
```

図 9: スキューイング適用後のプログラム

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) に変換される（図 8(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
```

図 10: ループ交換適用後のプログラム

これにより、図9のループは図10に示す通りに変換される。元の図9の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 $\vec{d_a''} = (1,1)$、$\vec{d_b''} = (1,0)$ に変換される（図8(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。依存距離が十分に大きい場合のステージングによる実行の様子を図11に示す。このようにステージ間に跨るループ運搬依存の依存距離を大きくとることで、実行インターバルをなくした通信遅延の隠蔽が可能になる。

5 通信遅延隠蔽のためのアルゴリズム

本章ではステージング技法における、通信と計算のオーバラップによる通信遅延隠蔽のためのループ変換アルゴリズムを示す。n 重ループ $L=(l_1, l_2, \cdots, l_n)$ 中、ステージ間通信を構成しているループ運搬依存の依存ベクトルの集合を D とする（ループ本体中に存在する全てのループ運搬依存の依存ベクトルの集合を C とすると,$D \subseteq C$）。ここで説明のため、依存ベクトル $(d_1, d_2, \cdots, d_n)$ 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。

本アルゴリズムは、D に属する依存の依存距離の値を大きくするループ変換を順次行なっていく。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。

アルゴリズムの具体的な処理手順を以下に示す。

(1) 変換対象ループを選出する。D に含まれる依存ベクトルの中で最も大きい依存レベルの値を k とすると、最初の対象ループを l_k に設定する。

(2) 依存レベルが k あるいは $k-1$ である依存ベクトルの集合 $D' \subseteq D$ を求め、D' に属する全依存ベクトルの第k次成分が正値になるスキューイングを行う。既に全て正値になっている場合はこの変換は行わない。

このスキューイングによる新たなインデックスは

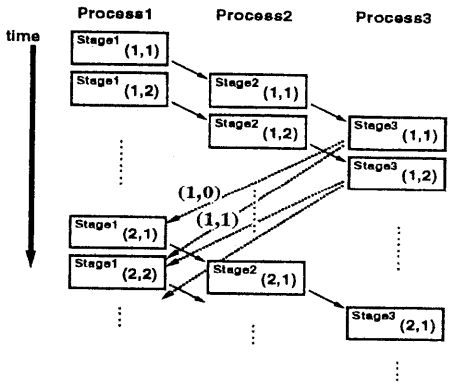

図 11: 依存距離の値が大きい場合のステージング

$$(i'_1, \cdots, i'_{k-1}, i'_k, \cdots, i'_n) = (i_1, \cdots, i_{k-1}, i_k + s i_{k-1}, \cdots, i_n)$$

と表現される。この変換は、$n \times n$ 単位行列中の k 行 $k-1$ 列要素をスキュー係数 s で置き換えたユニモジュラ行列で表すことができる。ここでスキュー係数 s は、D' に属する依存ベクトルのうち、依存レベルが $k-1$ で第 k 次成分が零値以下である集合 $D'' \subseteq D'$ から求めることができる。

D'' に属する任意の依存ベクトル $\vec{d}$ は、

$\vec{d} = (0, \cdots, 0, d_{k-1}, d_k, \cdots, d_n)$, $d_{k-1} \geq 1$, $d_k \leq 0$

と書ける。全ての $\vec{d} \in D''$ について

$d_k + s d_{k-1} \geq 1$　つまり　$s \geq (1 - d_k)/d_{k-1}$

を満たす最小の整数を s とすれば良い。

この変換はCに属する全ての依存ベクトルに関して参照順序の整合性を保持する。

(3) ループ l_k と l_{k-1} に関してループ交換を行う。この変換が正当であるためには両ループが可換である必要があるが、そうでない場合も C に属する依存ベクトルの情報からスキューイングを用い可換にすることが可能である [3]。このループ交換により D' に属する全依存の依存レベルは $k-1$ になり、D に属する全依存の依存レベルは $k-1$ 以下（l_{k-1} より外側ループのレベル）になる。

(4) 変換対象ループのレベルを一つ外側に移す（k:=k-1）。もし k=1 になったならアルゴリズムを終了する（D に属する全依存の依存レベルは最外ループに移った）。そうでないなら (2) に戻る。

6 評価

本稿で提案したアルゴリズムを含めたステージング技法をDOACROSS型ループに適用し、どの程度の高速化が見込めるかを調べた。評価にはSOR法、スプライン関数による補間法の2つのプログラムを用いた（図5,12）。

表 1: SOR コードに対する実行性能見積り

P		2	3	4	5 ～
C=2	B=1	1.353	1.533	1.917	1.917
	5	1.667	1.949	2.614	2.614
	10	1.716	2.018	2.738	2.738
C=8	B=1	0.793	0.852	0.958	0.958
	5	1.420	1.620	2.054	2.054
	10	1.575	1.825	2.396	2.396
	50	1.727	2.032	2.764	2.764
C=32	B=1	0.299	0.307	0.319	0.319
	5	0.891	0.966	1.106	1.106
	10	1.186	1.322	1.597	1.597
	50	1.611	1.873	2.478	2.478

表 1,2に各々の実行性能の評価を示す。ここで言う実行性能とは、対象プログラムに本手法を適用し、得られたステージ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ロード・ストアを 2 とする。プロセッサ資源数を P、プロセッサ間通信のための send,receive 命令実行に要する時間を C、最内ループに関して行ったブロッキングの展開回数を B とし、各値を変化させて評価をとった。

SOR 法のプログラムに関しては、4 章で説明した通り、依存グラフ中の強連結部分がループ本体中の大部分を占めるため、通信と計算のオーバラップのためのループ変換を試みなければ全く並列性を取り出すことができない。本稿で提案したアルゴリズムを適用することで表 1のような性能を出すことができる。

スプライン補間のプログラムは 1 重の DOACROSS 型ループであり、従来のイタレーション分割による並列化ができないループとして位置付けられる。ループが多重でないため、本稿で提案した依存距離を大きくするアルゴリズムを適用することはできないが、依存グラフ中の強連結部分を分割してステージを作ることはせずにそのままステージに割り付ける [6] ことで、表 2のような並列性を出すことができる。

並列性が一桁台に留まっているのは、いずれもループ本体の命令数自体が十数命令程度と少なく、ステージ分割の際に、最大になるステージの実行時間がそれ以上縮まらなくなるためである。ステージングにおいて高い並列性を引き出すためには、ループアンローリング等との組み合わせにより、複数イタレーションまでを考慮に入れたステージ分割が必要になってくると思われる。

1 イタレーション毎に通信を行う方式 (B=1) では、通信オーバヘッドが大きくなるにつれ性能が非常に悪くなる。通信オーバヘッドが大きい場合には、それに合わせた展開数のブロッキングが必須であると言える。

```
for i:=1 to IMAX do
  begin
    t := h[i]/d[i] ;
    Z[i+1] := Z[i+1]-Z[i]*t ;
    d[i+1] := d[i+1]-h[i]*t ;
    W[i+1] := W[i+1]-W[i]*t ;
  end
```

図 12: サンプルプログラム (スプライン補間)

表 2: 補間法コードに対する実行性能見積り

P		2	3	4 ～ 6	7 ～
C=2	B=1	1.450	1.611	2.416	2.900
	5	1.726	1.959	3.295	4.265
	10	1.768	2.014	3.452	4.531
C=8	B=1	0.906	0.966	1.208	1.318
	5	1.510	1.686	2.589	3.152
	10	1.648	1.859	3.021	3.816
	50	1.777	2.025	3.486	4.589
C=32	B=1	0.363	0.372	0.403	0.414
	5	1.007	1.082	1.394	1.543
	10	1.295	1.422	2.014	2.339
	50	1.678	1.898	3.125	3.984

7 おわりに

本稿では、ステージング技法を用いた並列実行における、プロセッサ間通信遅延の隠蔽技法と、そのアルゴリズムについて説明した。ステージングにおいて高い並列性を引き出すためには、ループ本体を構成する命令が多い方が有利である。並列性を高めるためのループアンローリングとの組み合せ、及び命令分割アルゴリズムの改良などが、今後の課題である。

参考文献

[1] 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).

[2] R.Tarjan, "Depth First Search and Linear Graph Algorithms", SIAM J.Computing, Vol.1, pp.146-160(1972).

[3] 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).

[4] Jennifer M.Anderson and M.S.Lam, "Global Optimizations for Parallelism and Locality on Scalable Parallel Machines", Conference on Programming Language Design and Implementation, pp.112-125(1993).

[5] M.E.Wolf and M.S.Lam, "A Data Locality Optimizing Algorithm", Proceedings of ACM SIGPLAN'91 Conf., pp.30-44(1991).

[6] 金丸、古関, 小松, 深澤, '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', 情報処理学会研究報告,96-ARC-117, pp.43-48(1996.3)