ハイパフォーマンス
コンピューティング 74−13
（1998. 12. 11）

計算機資源を考慮したパイプライン並列性抽出手法

湯田正人[†] 古関聰[††]
小松秀昭[††] 深澤良彰[†]

本稿では、細粒度ループ並列化技法であるループステージング技法における新たな命令分割アルゴリズムについて述べる。我々は、イタレーション分割手法において並列実行の阻害となっていたプロセッサ間通信を考慮し、命令間依存関係を通信コストで重み付けすることにより命令分割を行なっていた。しかしながら、実際には、プロセッサ間の通信やキャッシュ制御にかかる時間が解消されず、大きな効果が得られなかった。本手法では、依存関係に優先度を設けるのではなく、各種依存関係にある命令群を異なるプロセッサに割り付けた場合に要する通信時間及びキャッシュ制御時間をその依存の「コスト」と考え、これを命令分割の評価関数のパラメータに加えた新たな命令分割アルゴリズムを提案する。

A pipelining parallelism extraction technique considering machine resources

MASATO YUDA,[†] AKIRA KOSEKI,[††] HIDEAKI KOMATSU[††]
and YOSHIAKI FUKAZAWA[†]

In this paper, we describe a new instruction clustering algorithm for Loop Staging, which is one of fine-grain loop parallelization techniques. By this time, in Loop Staging, instructions have been clustered by prioritizing program dependences that become the communication cost between processors, which deteriorate the performance of parallel execution when applying existing iteration clustering methods. However, in practice, a large performance-up can't be achieved because of time for communication between processors and for controlling cache. In this method, we consider time for communication between processors and for controlling cache when we partition instructions in different processors as a 'cost', and propose a new clustering algorithm using this cost as a parameter of evaluating function of instruction clustering.

1. はじめに

高速実行を要求される科学技術計算プログラムなどでは、実行時間の大部分がループに費やされることが多い。我々は、ループ部分を高速実行するために、マルチプロセッサシステムにおけるループ並列化に着目し、研究の対象としている。この際、プロセッサ間の通信コストをいかに抑えてプログラムを分割し、プロセッサに割り付けていくかというプログラム分割の方法が大きな問題となる。

細粒度並列化技法の一つとして、ループステージング技法[1]がある。この技法は、ループ本体を構成している命令群をいくつかの「ステージ」という実行段階に分割してプロセッサに割り付け、それらをパイプライン実行することによって並列性を抽出する手法である。

ループステージング技法により、アクセスすべきデータが実行時に決定するようなデータ参照命令や、キャッシュラインの書き換えを頻繁に起こす可能性のあるデータ参照命令[2]など、並列処理を行なう際にボトルネックと成り得る命令を同じプロセッサに割り付けることが可能となる。これを実現するため、このような命令間には依存があるとし、各種依存に対して優先度を設け優先度が高い順に評価関数を適用する。

しかし、ループステージング技法における命令分割アルゴリズムは、多くのプログラムに見られる条件分岐を含むループに対してのサポートがなく、また命令の実行時間のみをパラメータとしてステージ分割の評価関数としているため、実際に分割してみるとプロセッサ間の通信や同期に費やすコストが大きく、期待したほどの性能向上度は得られなかった。

そこで、本稿では、ループステージング技法における命令分割アルゴリズムを改善することを提案する。具体的には、命令の実行時間の他に、各種依存の通信コストをパラメータとした新たな評価関数を用いて命令分割を行なっていくことによって、ループステージング技法による並列性抽出効果を上げることを試みる。

本稿では以下の順に説明を行なっていく。第二章で

† 早稲田大学理工学部
School of Science & Engineering, Waseda University
†† 日本IBM東京基礎研究所
Tokyo Research Laboratory, IBM Japan, Ltd.

は、従来のループ並列化技法を挙げてそれぞれの利点と欠点を説明し、その中で本研究の位置付けをする。第三章では、第二章で述べるループステージング技法における新しい命令分割アルゴリズムについて説明する。第四章では、新しい命令分割アルゴリズムを用いてループステージング技法を実際のループに適用することによって、従来手法では得られなかった実行速度の向上が得られることを示す。

2. 従来のループ並列化技法との関連

ループ並列化の際にはどのような単位でプログラムを分割し、プロセッサに割り付けていくかを決めなければならない。以下に、現在までに提案されてきたループ並列化手法を挙げる。

2.1 イタレーション分割手法

イタレーション分割方式[3]はループイタレーションをタスクとしたループ並列化技法の一つである。イタレーション空間をある次元で分割し、その分割片をそれぞれのプロセッサに割り当てる手法である。

分割片の間にデータ依存が存在する場合は、その依存関係を保証するために分割片間、すなわちプロセッサ間通信が必要である。多次元に渡り同時に複数のループ運搬依存が存在する場合には、どの次元でイタレーション分割を行なってもプロセッサ間通信が繁雑になり高速実行が望めない。そこでイタレーション空間をスキューイング[3]などのループ変換を用いて変形し、依存方向を変化させることによって高い並列性を引き出すことが可能である。しかしその結果、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なうことになり、複数のキャッシュが無効化要求 (invalidation)[2] を発行しあうという現象が生じてしまう。例えば同名の配列を参照する2つ以上の命令が、同じキャッシュライン上に存在するデータを参照、更新する場合、そのような命令を異なるプロセッサに配置すると一つのキャッシュラインの頻繁な書き換えが起こり、互いに無効化要求を発行し合うため高速実行が妨げられる。本稿では、このような命令間にリソース依存 (resource depnedence) があるものとする。

また、この手法は、コンパイル時に解析できない依存を持つループに対してはどの次元に関してループ変換を行なうべきかわからないため、並列性を引き出すことができない。例えば配列の添字内に配列によるインデックス参照が存在する場合などは、アクセスすべきデータが実行時に決定されるためにコンパイル時には解析できずデータ依存の有無を知ることができない。このような依存を本稿では動的依存 (dynamic dependence) と呼ぶ。

よって、この手法はイタレーション間に依存のないDOALL型ループに対しては高い並列性を抽出できるが、複雑なループ変換が必要とされるループ、すなわち多次元に渡りループ運搬依存が複数存在するDOACROSS型ループに対してはリソース依存によるコストが大きくなり高速実行が望めない。また、動的依存をもつループに対してもプロセッサ間通信の有無は実行時にならなければわからないため、ほとんど並列化効果は得られない。

2.2 ループステージング技法

ループステージング技法[1]は命令レベルのループ並列化技法の一つである。ループ本体を構成している命令群を「ステージ」といういくつかの実行段階に分割し、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再構成し各プロセッサに割り付ける。ステージ間に依存関係が存在する場合にはそれを保証するためにプロセッサ間で通信を行なう必要がある。このように、ループ本体の命令群を分割して作られた新しい複数のループを各プロセッサに割り当てることによって、それぞれのループをオーバーラップさせたパイプライン並列実行が可能となる。

この手法は、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良いという利点がある。マルチプロセッサシステムにおいてはループ運搬依存や動的依存の通信コストが高速化を妨げる原因であるが、ループステージング技法では、命令分割を上手に行なうことにより、必ずしもそれらに関してのプロセッサ間通信を行なう必要はない。

さらに、ループ中のデータ参照命令は繰り返し同じプロセッサで実行されるので、イタレーション空間を変形させるループ変換に関係なく、キャッシュラインに書き込みを行なうプロセッサを一つに定めやすい。つまりデータを私有化 (privatalization)[4] することが可能となるので、リソース依存による無効化コストを抑えることができる。

よってループステージング技法は、イタレーション分割手法とは逆にDOALL型ループの並列化には向いていないが、イタレーション分割手法で高速実行が望めなかったループ、すなわち多次元に渡る多数のループ運搬依存を持つループや動的依存を含むループに対して効果がある、といった特徴を持つ。

ループステージング技法における命令分割手法はPDG(Program Dependence Graph)[5] 上の融合操作を繰り返すものである。PDGとは命令をノード、命令間の依存をそれを繋ぐエッジで表現したグラフである。また融合操作とは、同じプロセッサ (ステージ) に割り付けるべき依存関係にある二つのノードを一つのノードにする操作である。

融合すべき2ノードを選択する基準として、ループ本体を構成している全ての依存エッジに優先度を設ける。依存のある2ノードを異なるプロセッサに配置したときのレイテンシの大きさを元に依存の種類ごとに優先度を設ける。リソース依存を最優先し、動的依存、ループ運搬依存、ループ独立依存の順で優先度を与え、優先度の

高い依存エッジから以下の評価式によって融合するべき依存エッジを選択し、繋がっている2ノードを融合する。

ループステージング技法はパイプライン並列性を利用しているため、各ステージの実行時間中、最大のものの値をできるだけ小さくすることが必要である。そこで全ノードの実行時間の総計をT_{all}、依存エッジで結ばれている2ノードの実行時間をw_s, w_eとすると、まず高い優先度を持つエッジの集合の中から、$w_s + w_e < T_{all}/P$かつ$w_s + w_e$が最大であるエッジを選択する。条件を満たすようなエッジがない場合は$w_s + w_e$が最小であるエッジを選択する。このような融合操作を(ノード数)≦(プロセッサ数)を満たすまで繰り返す。ただし、融合した結果循環グラフになってしまうようなエッジは融合対象から除く。

しかしこの命令分割手法は、次の問題点を持つ。

(1) 命令の実行時間のみを評価関数のパラメータとしているため、実際に分割を行なってみるとプロセッサ間のデータ通信や同期に費やされるコストがボトルネックとなり、高速実行ができない場合がある。

(2) ループ中に条件分岐が存在する場合、その分岐方向が決定するまでそれ以降の命令を実行することができないため、スケジュールの自由度が小さくなりプロセッサ資源の有効活用ができない。

(3) 並列実行するのに最適なプロセッサ数に分割することができない。ループによってはプロセッサ数が少ない方が高速に実行できるが、このアルゴリズムの性質上、ステージは必ずプロセッサ数個になる。

我々は、ループステージング技法における上記の問題点を解決する新しい命令分割アルゴリズムを提案する。(1)の問題点は、すべての依存エッジに対して通信コストを、すべての命令ノードに対して総実行時間を算出し、それらをパラメータとした新しい評価関数を用いることによって解決する。(2)の問題点は、条件分岐以降の命令を投機的実行[6]することによって解決する。(3)の問題点は、ノード融合操作の終了条件を改善することによって解決する。

3. 本命令分割手法

本章では、ループステージング技法における命令分割手法を改善し、さらに高い並列性を抽出する新しい命令分割手法を提案する。

新しい命令分割手法を適用するにあたり、我々はGPDG(Guarded Program Dependence Graph)[7]と呼ばれるグラフ構造を用いる。GPDGは命令をノード、命令間の依存をそれを繋ぐエッジで表現したものであり、各ノードに制御依存による条件コードを付加したものである。GPDGを用いることによって条件分岐による制御依存をデータ依存などと同様に扱うことが可能となる。

本分割手法は、従来アルゴリズムで優先度という形でしか考慮しなかった各種依存のコストをパラメータ化し、GPDG上の全ての依存エッジに対して評価関数を適用していく。また、条件分岐などによる制御依存の解決として投機的実行を適用する。

3.1 SSA変換と投機的実行

SSA(Single Static Assignment)[8]変換は、プログラム内において同一変数の定義が1回しか行なわれない形式へ変換するものである。各変数Vに代入が起こる度に新しい変数名V_iが使用される。また、異なる制御の流れからの変数を区別するために制御の合流点にϕファンクションと呼ばれる特別な命令を挿入する。SSA変換によって出力依存を除去することができる。

条件分岐の結果を待たずに以降の命令を実行することを投機的実行[6]という。本来、ループ中に条件分岐などの制御依存が存在する場合、分岐方向が確定するまでそれ以降の命令を実行できないため十分な並列性を得ることはできない。そこで制御依存による実行順序の制約を無視して条件分岐命令よりも先に後続命令を実行し、分岐方向が確定した時点で投機的実行した命令が有効であるかを判断する。投機的実行される命令は条件分岐などの結果を反映せずに実行されてしまうので、除算命令やϕファンクション命令、ストア命令などのメモリの状態を変える可能性がある命令は投機的実行することはできない。また、それらの命令からのデータ依存を持つ命令も投機的実行することができない。投機的実行によって、条件分岐を越えてスケジューリングをすることが可能となるため各CPUにおけるALUの充填率が向上し、命令レベルの並列性を向上できる。投機的実行を行なう際には出力依存を除去するSSA変換は必要不可欠であり、ステージング対象となるループの並列性を向上できる。

3.2 命令分割アルゴリズム

本手法は、ループ本体をGPDGで表現し、その上でノード融合を行なっていく。順方向エッジは、GPDG上におけるTOPノードからあるノードに至る最長パスの長さ(レベル)の小さいノードからレベルの大きいノードへの有向エッジ、逆方向エッジはレベルの大きいノードからレベルの小さいノードへの有向エッジと定義する。レベルlevel(A)とは、GPDG上におけるTOPノードからノードAに至る最長パスの長さである[1]。

命令分割を行なう前に、ループ本体に対してループステージング技法向きの変換を行なっておく。具体的には短い依存距離を長くするためにループ変換技法[1]を適用し、さらに投機的実行を行なうためにSSA変換を行なっておく。

図1, 図2に我々が本稿で提案する命令分割アルゴリズムを示す。以下では図1,2の各部分について説明を行なう。

```
void staging(GPDG gpdg){
  Edge fusion_edge;
  while(1){
    initialize(gpdg); .....(A)
    fusion_edge
    = find_edge(all_edge∈gpdg); .....(B)
    if(終了条件) .....(E)
      終了;
    else
      fusion(fusion_edge);
  }
}
```

図1 命令分割アルゴリズム

```
Edge find_edge(Edge all_edge){
  Node t_node;
  int partition_cost,fusion_cost,profit[];
  for(edge∈all_edge){
    partition_cost = calc_cost(edge); ..(C)
    t_node = tmp_fusion(edge); .....(D)
    fusion_cost = calc_weight(t_node);
    profit[edge] = partition_cost-fusion_cost;
  }
  fusion_edge
    = 最大の profit[edge] のときの edge;
  return fusion_edge;
}
```

図2 最適エッジ選択アルゴリズム

(A) 関数 initialize

まず命令分割関数 staging では、関数 initialize によって、GPDG 上の全てのノードに対してレベル付けと重み付けを行ない、エッジに対してコスト付けを行なう。ノード A の重み weight(A) とは、ノード A の総実行時間である。重み付けする際に、全てのノード中の命令に対して投機的実行可能であるかを調べ、投機的実行可能な命令はすべて投機的移動[6] を行なう。以降は投機的移動済みの GPDG を対象とする。また、エッジコストとは、そのエッジで結ばれた二つのノードを異なるプロセッサに配置したときに要するコストである。エッジコストは、依存エッジ別コスト付けの後、エッジの接続形態によるコスト付けを行なうことによって求められる。

- エッジの種類
  まずは依存エッジ別コストを求める。動的依存エッジやループ独立依存エッジ、そして依存距離[3] の小さいループ運搬依存エッジのコストをデータ通信コスト $cost_d$、リソース依存エッジのコストを無効化コスト $cost_i$ とする。$cost_d, cost_i$ はアーキテクチャの仕様に準拠するものとする。ただし、制御依存エッジコストは投機的実行を適用するため 0、依存距離が十分大きいループ運搬依存エッジのコストは 0 とする。リソース依存エッジに関しては、キャッシュライン上におけるデータ間の位置的な差をリソース依存の依存距離とすると、依存距離がキャッシュライン長よりも大きいのであればその命令間には実質上同一キャッシュラインの書換えはないのでコストは 0 とする。
- 接続形態
  ノード S からノード E へのエッジが n 本存在するとき、それらの依存エッジ別コストを $cost_1, cost_2, ..., cost_n$ とすると、そのエッジコスト edge_cost は、
  edge_cost = $\max(cost_1, cost_2, ..., cost_n)$
  で求める。

(B) 関数 find_edge

全てのノードの重みとエッジコストを算出した後、GPDG 上の全ての依存エッジに対して、融合するべきエッジを選択するアルゴリズム find_edge を適用する。以下、関数 find_edge(図 2) の詳細を説明する。

(C) 関数 calc_cost

関数 calc_cost は、引数エッジで繋がれた 2 ノードを異なるプロセッサに配置した場合の 1 イタレーション当たりのコストを算出する評価関数である。

まず、評価に必要なパラメータを、ノード内の命令に関する依存解析を行なうことによって求める。

関数 calc_cost の引数として与えられたエッジの始点ノードを S、終点ノードを E とする。

ノード S 中の命令のうち、ノード E 中の命令への依存を持つものを選択しさらにその中で最後に実行される命令を選択する。これを sendS 命令とする。

次にノード E 中の命令のうち、ノード S 中の命令からの依存を持つものを選択し、さらにその中で最初に実行される命令を選択する。これを receiveE 命令とする。

もし、ノード E 中の命令のうち、ノード S 中の命令へのイタレーションを跨ぐ依存が存在すれば、それを持つものを選択し、さらにその中で最後に実行される命令を選択する。これを sendE 命令とする。

ここで、ノード S、ノード E 間の実行タイミングについて説明する。ノード S は、先頭から sendS 命令までを実行しノード E に通信すべきデータ (すなわち共有データ) をアグリゲイティング、コアレイシングによって 1 パケットにまとめてデータを送信する。一方、ノード E は receiveE 命令の前まではノード S に関係なく命令を実行できるが、receiveE 命令以降を実行するためにはノード S からのデータ送信を待つ必要がある。イタレーションに跨るような依存がノード E からノード S に存在する場合には、sendE 命令で作られたデータをノード S に送信する必要がある。

これらの命令を選択した後、それぞれの情報からノード S の先頭から sendS 命令終了までの重み weight(sendS)、ノード E の先頭から receiveE 命令実行前までの重み weight(receiveE)、ノード E の先頭から sendE 命令終了までの重み weight(sendE) をそれぞれ算出する。これらは評価のパラメータとして用いられる。

ノード内の命令に関する依存解析の終了後、ノード S

とノードEを異なるプロセッサに配置した場合の1イタレーションのコスト評価式を、ノード間のエッジの接続形態によって区別して適用する。

エッジで繋がれている二つのノードがループ独立依存などのイタレーション内で閉じている依存のみで接続されている場合、パイプライン実行が可能であるため高い並列性を引き出すことが可能となる。逆に、その他に動的依存などのイタレーション間に跨る依存がノード間に存在する場合は、双方向データ通信が必要になるためパイプライン実行が阻害される。前者のようにパイプライン実行が可能であるエッジを片方向エッジ、後者のように双方向データ通信が必要であるエッジは双方向エッジと呼ぶことにする。片方向エッジのコストを edge_cost_p、双方向エッジのコストは、それぞれ順方向エッジコスト edge_cost_f, 逆方向エッジコスト edge_cost_r とする。これらのコストはすでにエッジのコスト付けの時点で求められている。

(1) 対象エッジが双方向エッジでありかつエッジコストが片方向、順方向ともに0でない場合、ノードEからノードSへのイタレーション間に跨る(またはその可能性のある)依存が存在するためにスムーズなパイプライン実行をすることができない(図3)。そのためコスト評価式は
weight(sendS)+edge_cost_f+weight(sendE)
-weight(receiveE)+edge_cost_r
とする。

(2) (1)でない場合、主に対象エッジが片方向エッジの場合には、ノードSからノードEへの依存のみを保証するだけなのでパイプラインピッチの短いパイプライン実行が可能となる(図4)。よってコスト評価式は、ループ回数が非常に大きいものとすると、
max(weight(sendS), (weight(E)
-weight(receiveE)))+edge_cost_p
とする。

これらのコスト評価式の適用によって得られた値を *partition_cost* とする。

(D) 関数 tmp_fusion

ノードSとノードEを同じプロセッサに配置した場合のコスト評価式を適用する。ノードSとノードEを融合して命令をスケジューリングしたときのノード t_node=(S,E) の重み weight(t_node) をそのコストとし、その値を *fusion_cost* とする。

二つのコスト評価式によって求めた *partition_cost* と *fusion_cost* の差を *profit* とすると、*profit* は融合することによって通信コストを低減できるコスト利益を意味するため、*profit* が最大であるエッジを選択し関数 find_edge の戻り値とする。

(E) 終了条件

関数 find_edge によって選択されたエッジは、以下に述べる融合終了条件を満たさなければ融合を行なう。選

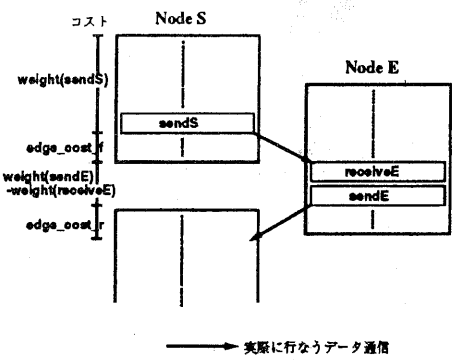

図3 双方向エッジのコスト評価

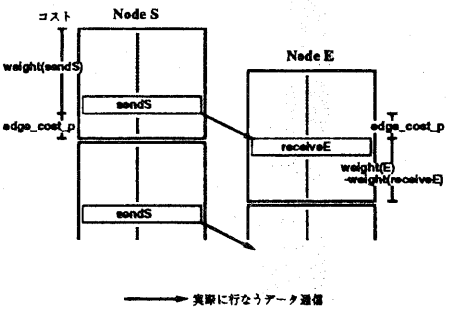

図4 片方向エッジのコスト評価

択されたエッジを融合した結果、循環グラフを作ってしまう場合には、そのサイクルを構成している全命令を一つのノードにする。ここで本分割手法の融合終了条件を

(number_of_node≤1)
|| ((2≤number_of_node≤P)
&&(profit[α]≤0,∀α ∈all_edge))

とする。ただし number_of_node、all_edge をそれぞれ GPDG 上のノード数、依存エッジの集合とする。

以上の命令分割操作を終了条件を満たすまで繰り返す。

融合操作終了後、各ステージの必要な位置に通信、同期コードを挿入し、それらを本体としたループを作りプロセッサに割り付ける。

4. 評　　価

本章では、動的依存、リソース依存を含むループを対象に本手法を適用し、逐次実行時とイタレーション分割手法適用時、ループステージング技法の従来分割手法適用時と比較した結果により、本手法の有効性を示す。

評価対象のループは、配列間接参照を含むGaussコード, ピボット付き lu 分解コード, spline コード, FFT コードである。これらのコードの特徴を表1に示す。

4つの Pentium Pro™200MHz プロセッサを用いた共有メモリ並列計算機を用いてそれぞれのコードに対し

表1 評価対象のコード

コード	ループの深さ	動的依存	リソース依存	依存距離の短いループ運搬依存	ループ独立依存	制御依存
gauss 法	3重	有	有	無	有	有
lu 分解	2重	有	有	無	有	無
spline 法	1重	無	有	有	有	無
FFT	3重	有	有	無	有	有

て並列処理を行なった。命令分割のパラメータである命令の実行時間や通信コストなどは、プロセッサの仕様に準拠するものとする。また一次キャッシュの連想方式は4ウェイセットアソシエイティブであり、キャッシュライン長は32バイトで実現されている。

逐次処理時の実行時間を基準としたときの各コードの処理速度の向上比を表2に示す。表中の数字は逐次処理時に対する速度向上の割合、括弧内の数字は実行プロセッサ数を表す。また、実行における通信コストを低減し局所性を高めるため、全ての並列化手法に対してループブロッキング技法[9]を適用した。

gauss 法においては、従来の分割手法と比較しても大幅な性能向上を実現できた。これは動的依存を持つ命令群を同じプロセッサに配置することによって共有データを最少限に抑えることができたためである。

lu 分解においては、条件分岐が存在しないために投機的実行による効果が得られず、全く同じ命令群がプロセッサに割り付けられたために従来分割手法とほぼ同じ性能しか得られなかった。

spline 法は一重ループであるため、ループ変換によってループ運搬依存を解消できない。そのためイタレーション分割手法ではイタレーションに跨る依存に関しての通信が高速実行を妨げる原因となっていた。この関係にある命令群を同じプロセッサにうまく配置できたため従来分割手法と比較しても高速実行を実現できた。

FFT においても条件分岐による制御依存制約と動的依存による通信コストにより、イタレーション分割手法では全く並列性を引き出すことができなかったが、投機的実行による制御依存遅延の解消と依存のある命令群を一つのプロセッサに配置することができたので、大きな性能向上を得ることができた。

表2 処理速度比

コード	イタレーション分割手法	従来分割手法	本分割手法
gauss 法	0.824(4)	1.057(4)	1.610(3)
lu 分解	1.521(4)	2.333(4)	2.333(3)
spline 法	1.126(4)	1.733(4)	2.128(3)
FFT	0.133(4)	1.426(4)	2.033(4)

5. おわりに

本稿では、ループステージング技法における命令分割手法を改善することによってパイプラインによる並列性を最大限に引き出すことに成功した。具体的には、(1) 投機的実行を行なうことによる GPDG 上におけるノード間の並列性を高め、(2) 命令の実行時間だけでなくデータ通信コストや無効化コストをパラメータとした評価関数を提案した。その結果、イタレーション分割手法では全く高速化できなかったループに対して、逐次実行時よりも1.6~2.3倍、イタレーション分割手法よりも約2倍、従来手法による分割時よりも最大1.6倍程度の処理速度向上を実現することができた。

参考文献

1) 金丸, 古関, 小松, 深澤, ループステージング: 共有メモリ型並列計算機を対象としたループ並列化技法とその評価, 情報処理学会シンポジウム, pp.197-pp.204, 1997.

2) A.Gupta and W.D.Weber, *Cache Invalidation Patterns in Shared-Memory Multiprocessors*, IEEE Transactions on Computers, Vol.41, No.7, pp.794-pp.810, 1992.

3) M.E.Wolf and M.S.Lam, *A Loop Transformation Theory and an Algorithm to Maximize Parallelism*, IEEE Transactions on Parallel and Distributed Systems, Vol.2, No.4, pp.452-pp.471, 1991.

4) J.Gu, Z.Li, and G.Lee, *Experience with Efficient Array Data Flow Analysis for Array Privatization*, Proceedings of the ACM SIGPLAN Symposium on PPoPP, pp.157-pp.167, 1997.

5) J.Ferrante,K.J.Ottenstein, and J.D.Warren, *The Program Dependence Graph and Its Use in optimization*, ACM Trans.Prog.Lang.and Syst, Vol.9, No.3, pp.319-pp.349, 1987.

6) 山名, 安江, 石井, 村岡, 並列処理システムにおけるマクロタスク間先行評価方式, 電子情報通信学会論文誌, Vol.J77-D-I, No.5, pp.343-pp353, 1994.

7) 古関, 小松, 深澤, 命令レベル並列アーキテクチャのための大域的コードスケジューリング技法とその評価", 並列処理シンポジウム JSPP'94 論文集, pp.1-pp.8, 1994.

8) R.Cytron, J.Ferrante, B.K.Rosen, M.N.Wegman, and F.K.Zadeck, *An Efficient Method of Computing Static Single Assignment Form*, In Proceeding of 16th Annual ACM Symposium on Principles of Programming Languages, pp.25-pp.35, 1989.

9) M.E.Wolf and M.S.Lam, *A Data Locality Optimizing Algorithm*, In Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation, pp.30-pp.44, 1991.