計算機アーキテクチャ　107－2
（１９９４．　７．　２１）

細粒度コードスケジューリング手法の提案とその評価

浜田智雄, 野口善昭, 前川孝徳,岩根雅彦
九州工業大学工学部

細粒度コードスケジューリング手法としてTree Weight Reduction（ＴＷＲ）とマイクロタスクグラフ（ＭＴＧ）を提案する．ＴＷＲは同一基本ブロックに属する代入文を並列化しプロセッサ割り当ての最小単位となるマイクロタスクブロック（ＭＴＢ）を抽出する．ＭＴＢの依存関係はＭＴＧによって表わされる．ＭＴＧは異なった基本ブロックに属するＭＴＢ間の並列実行可能性を検出する．これらを用いることで逐次コードの並列化が行える．またループにも応用できる．手法の評価のためにベンチマークプログラムを並列化し，並列計算機ＭＳＢＭで実行した結果1から2.3倍の速度向上が得られた．

A proposal of fine-grain scheduling method and its evaluation

Tomoo Hamada, Yoshiaki Noguchi, Takanori Maekawa,Masahiko Iwane
Faculiy of Engineering ,Kyushu Institute of Technology

To exploit fine grain parallelism, this paper proposes the Tree Height Reduction algorithm (TWR) and the Micro Task Graph (MTG) scheduling. The TWR compact some assignment statements in one basic block and extract some Micro Task Blocks (MTBs). The dependence of MTBs is expressed by the MTG. The MTG can detect parallelism of MTBs belonging to different basic blocks. The TWR and MTG can parallelize sequencial code. And these are applicable to software-pipelining. To evalute of these methos, we parallelized benchmark programs. And we get speedup ratio from 1.0 to 2.3.

第１章　はじめに

マルチプロセッサ計算機で，プログラムを並列化しその処理速度を向上させようとする場合，プログラムの意味的同一性を保持しつつ並列実行可能な部分を最大限に抽出していく必要がある．特にプロセッサ数にほぼ比例した速度向上率が得られるDOALLループをプログラム中からより多く見い出すことはプログラム全体の速度向上に大きく寄与する．しかしながら一般的なプログラム中にはイタレーション間に依存性の存在するループや逐次コードも数多く存在するため，これらの並列化を行わない限りプログラム全体の処理速度向上は望めない．従ってそれらのコードに対して機械語命令あるいはサブステートメントレベルでの並列処理を行い，性能向上を計る必要がある．

原始プログラムの代入文の並列化手法としてTree Height Reduction（以下THR）アルゴリズムがある[1][2]．このアルゴリズムは代入文を表わす計算木を完全二分木に変換することによって１つの代入文の並列処理可能性を高めているが，ステートメント間の並列処理可能性について考慮していない．またこのアルゴリズムによって生成される中間コードをプロセッサ割り当ての最小単位として用いると，プロセッサ間の同期／通信のオーバーヘッドが考慮できない．

マクロフローグラフ[3][4]はイリノイ大学のCEDARプロジェクトで用いられたグラフである．このグラフはその名の通り粗粒度タスクのスケジューリングに用いられるもので，細粒度のタスクのスケジューリングには用いられなかった．しかしこのグラフを用いることで並列性抽出の範囲を単一の基本ブロックから条件分岐文を含んだ複数のそれに広げることが出来る．

以上の事から本論文では原始プログラムの代入文内/間の並列実行可能性を抽出し，さらにプロセッサ割り当ての最小単位となるマイクロタスクを抽出するTree Weight Reductionアルゴリズムと，それらの間の並列実行可能性を抽出するマイクロタスクグラフについて述べる．

本稿第２章ではスケジューリングの基本概念および第３章以降で用いられる用語について述べる．第３章でマイクロタスクブロックの抽出法について述べ，第４章でそれを使用した逐次コードのスケジューリング手法について述べる．またこの手法がループのソフトウェアパイプライン処理に応用できることを示す．また第５章でベンチマークプログラムを並列化し単一プロセッサで実行し場合との速度向上比を求め，スケジューリングアルゴリズムの有効性の評価を行った．評価用の計算機としては並列計算機MSBM[5]を用いた．

第２章　基本概念

2.1　プログラムの構造

Pascalなどの手続き型言語で記述された原始プログラムを手続き（関数）およびループの境界で区切った際のコード領域を原始タスクブロックと呼ぶ．原始タスクブロックはその内部に原始タスクブロックを包含したループあるいは関数であると言える．例えば図１ではfor文を構成するT1が関数funcを構成するT2に包含され，T2はメインプログラムT3に包含されている．

```
program prog(input,output);
var ...
 function func(...) begin (T2)
  (non loop code)
  forserial i:=1 to N do  begin(T1)
   (non loop code)
  end ;
  (non loop code)
 end
begin                              (T3)
 (non loop code)
 func(...)
 (non loop code)
end.
```

図１：原始タスクブロック

原始タスクブロックのスケジューリング結果をタスクブロックとする．スケジューリングとは対象となる原始タスクブロックが包含するタスクブロックを１つの文とみなし，代入文,条件分岐文と併せて並列化することにより新たな１つのタスクブロックを生成する作業である．この作業は最も内側の（すなわちタスクブロックを包含しない）タスクブロックから開始され，プログラム全体が一つのタスクブロックとなるまで階層的に続けられる．

タスクブロックと他の文の並列実行は許されない．しかしながらそれを１つの文として取り扱うことで，その前後に存在する代入文の並列処理可能性を考慮できることになる．

2.2　中間コードと変数取得時間

原始タスクブロック中の代入文からサブステートメントレベルの並列性を抽出するために，スケジューリングの最小単位として中間コード（３番地コード[6]）を用いる．文中のｖは通常変数（プログラム中に明示的にかかれた変数）あるいは一時変数（コンパイラによって生成される変数）を表わしている．

［Ｃ１］演算コード：

$$v1:=op_1\ v2\ ,\ v3:=\ v4\ op_2\ v5$$

op_1,op_2はそれぞれ単項,二項演算子を表わす．

［C２］複写コード,配列参照コード：

v1:=v2, v3:=v4[v5], v6[v7] := v8

［C３］制御コード：

goto L, if v1 relop v2 L, ifnot v3 relop v4 L

relopは関係演算子,Lはラベルを表わす.

［C４］関数,手続き呼び出しコード：

v_o := func_name{v_{i1},v_{i2},...,v_{iN}},{v_{m1},v_{m2},...,v_{mM}}

proc_name {v_{i1},v_{i2},...,v_{iN}},{v_{m1},v_{m2},...,v_{mM}}

関数,手続き呼び出しを行う．変数v_oは関数の戻値，v_{i1},v_{i2},...,v_{iN}は関数,手続きに値渡しされる変数，{v_{m1},v_{m2},...,v_{mM}}は参照渡しされる変数を表わす．

中間コードｃの実行時間をexec(c)で表わす．また中間コードのオペランドに用いられる各変数に取得時間を与える．変数ｖの取得時間はタスクブロックの実行開始時点を０とした時にｖの値が算出される時間であるとし，get(v)と表記する．オペランド中の定数はは取得時間が０の変数として取り扱う．

2.3 部分ブロック

演算コード,複写コード,配列参照コードのシーケンスを部分ブロックとする．部分ブロックｂ内で定義されることなく参照される変数の集合を入力変数と呼びinput(b)で表わす．同様に部分ブロックｂ内で定義あるいは再定義され，ｂの外部で参照される変数の集合を出力変数と呼びoutput(b)で表わす．また中間コードと同様部分ブロックｂの実行時間をexec(b)で表わす．部分ブロックｂの実行時間は，部分ブロックｂが含んでいる中間コードの実行時間の和で表わされる．

部分ブロックは入力変数がすべて得られた時点で実行可能になる．そこで部分ブロックｂの入力変数の取得時間の最大値を実行可能時間とよび，allget(b)と表記する．

さらに部分ブロックｂが実行開始される時間をbegin(b)，終了する時間をend(b)で表わす．これらの関係は次のようになる．

begin(b)≧allget(b)) ...(1)

end(b)=begin(b)+exec(b) ...(2)

部分ブロックｂの出力変数はｂの実行が終了した時点で得られるとするため，ｂの出力変数の取得時間はｂの終了時間と同じになる．

2.4 基本ブロックとタスクブロック

原始タスクブロックは条件分岐文によって幾つかの基本ブロック[6]（連続した文からなり，制御は必ず先頭の文から与えられ最後の文まで停止，分岐しない文の並び）に分けられる．さらに代入文中の関数呼び出しを代入文とは分離して考える事により，基本ブロックの構成要素はタスクブロックを表わす文と変数,四則演算子および括弧からなる代入文に分けられる．今後代入文は関数呼び出しを含まないと考える．

タスクブロックｔが使用するプロセッサの数ｎをn=punum(t) (1≦n≦PUNO) とする．タスクブロックｔの入力/出力変数をinput(t)/output(t)実行時間をexec(t)で表わす．またタスクブロックｔと同じ回数だけ同期命令を発するコードを空のタスクブロックとよびnull(t)で表わす．基本ブロックｂにも同様にinput(b), output(b), exec(b)を定義する．

関数,手続き呼び出しを表わすタスクブロックｔやnull(t)は実際には関数,手続き呼び出しコードしか含んでいない．しかしながらその情報（入出力変数および実行時間）は呼ばれる側の関数,手続きのそれと同じであると考える．システムに用意されている関数,手続き（ＦＰＵを使用する標準関数等）をシステム関数と呼び，１つの中間コードとして取り扱う．

第３章　マイクロタスクブロック

3.1 Tree Weight Reduction アルゴリズム

並列化によって処理速度の向上を計る場合，プロセッサ間の同期/通信のオーバーヘッドを考慮しなければならない．さらに分割すると逆に処理速度が低下してしまう部分ブロックをマイクロタスクブロックとする．マイクロタスクブロックの抽出は原始タスクブロックを構成する各基本ブロック毎に行われる．これに対して変数の取得時間は基本ブロックを超えて伝搬する．そこで条件分岐によって変数ｖの取得時間が複数得られた場合に，制御の結合点における変数ｖの取得時間を

get(v)=max(get(v1),get(v2),...,get(vn),get(c))

とする．get(v1),get(v2),...,get(vn)は各分岐先における変数の取得時間，get(c)は分岐制御式（条件式）の演算結果の取得時間とする．

マイクロタスクブロックは基本ブロック内部のデータ先行グラフを生成する過程で抽出される．データ先行グラフのノードはタスクブロック,マイクロタスクブロック,部分ブロックの何れかを表わし，辺はそれらのデータ依存関係[注1]を表わす．文中のt_iは値を割り当てられる度に生成されるユニークな一時変数，v_iは通常変数，x_iはそのいずれかを示す．また"→［ｎ］"は現在実行中の処理を終えた後，［ｎ］に進む事を示す．

［１］準備：グラフはダミーノードとして使用されるエントランスノード１つからなるとする．エントランスノードは実行時間０のタスクブロックで，出力変数として基本ブロックの入力変数（取得時間

注1 本論文ではデータ依存として真依存関係しか考慮しない．それ以外の逆/出力依存は変数リネーミング等を用いて回避されているものとする．

0）を持つとする．→［2］

［2］文の取り出し：基本ブロックから順に文を取り出す．もしその文がタスクブロックを表わすならば［3］，代入文ならば［4］で処理する．基本ブロック中の文が全て処理された時点でアルゴリズムは終了する．

［3］タスクブロックの処理：タスクブロックtを表わすノードをグラフに付加し，グラフ中のデータ依存する，即ちinput(t)内の変数を算出するノードからtへの辺を描く．tの出力変数の取得時間をexec(t)＋allget(t)とする．→［2］

［4］代入文の処理：括弧を含まない式を部分式，部分式を右辺に持つ代入文を部分代入文と呼ぶ．代入文の最も内側の括弧で括られた部分式をPとし，t_i:=Pなる部分代入文を［5］で処理する．取り出した部分式（およびそれを括る括弧）をt_iに置き換える．この作業を代入文の右辺全体が部分式で表わされるまで繰り返す．最後にv:=Pなる式を［5］で処理する（vは代入文の左辺の通常変数）．→［2］

［5］部分式の処理：変数$v_1,v_2,...,v_n$の加減算を行う関数を$\sigma(v_1,v_2,...,v_n)$,乗除算を行う関数を$\pi(v_1,v_2,...,v_n)$とした時，部分式Pは

$$P = \sigma(\pi(...),\pi(...),...)$$

で表わされる．まずσ関数内の各π関数を取り出し$t_i:=\pi(...)$なる式を［6］で処理し$\pi(...)$をt_iに置き換える．π関数の置き換えはσ関数内にπ関数が無くなるまで続ける．最後にx:=$\sigma(...)$なる式を［6］で処理する．（xは与えられた部分代入文の左辺変数）．→［4］

［6］σまたはπ関数の処理：σまたはπ関数は処理する演算内容以外差異はないので以下では単にσ関数と記述する．σ関数内の$v_1,v_2,...,v_n$の内で符号変換を伴う変数v_iを中間コードa（t_i := op v_i）で表わされるt_iで置き換える．またaを部分ブロックとしてグラフに付加し，データ依存するノードからaへ辺を描く．この時aの出力変数t_iの取得時間を

$$exec(a)+get(v_i)$$

とする．同様にσ関数中で配列アクセスを行う変数$v_j[v_k]$も中間コードb（$t_i:=v_j[v_k]$）で表わされるt_iで置き換える．bを部分ブロックとしてグラフに付加し，データ依存するノードからbへ辺を描く．bの出力変数t_iの取得時間を

$$exec(b)+max(get(v_j),get(v_k))$$

とする．

次にσ関数内の変数で他の変数より取得時間が大きくない変数を2つ選びx_{s1},x_{s2}とする．x_{s1}とx_{s2}の演算内容をopで表わすと，これらの演算結果を求める中間コードcは$t_i:=x_{s1}$ op x_{s2}となる．cについて［7］で処理したあとx_{s1},x_{s2}を一時変数t_iで置き換える．この作業をσ関数内の変数が2つ以上ある間続け，最後にx:=x_{s1} op x_{s2}なる中間コードを［7］で処理する（xは［5］でu :=σ (...)として与えられた変数）．→［5］

［7］Weight Reduction:［6］で与えられた中間コードcを部分ブロックとしてグラフに付け加え，データ依存するノードと接続する．cのオペランドx_{s1},x_{s2}の値を出力するノードをそれぞれp1,p2とし，その性質にしたがって次の処理を行う．

（Ⅰ）p1,p2がいずれも部分ブロックではない場合：cの演算結果の取得時間はp1,p2を並列実行すると考えて

$$allget(c)=max(end(p1),end(p2)+O_s)+O_b+O_l$$
$$end(c) = allget(c) + exec(c)$$

（O_sの加算はp1,p2の内終了時間が小さい方に行われる．）　...(3)

と表わされる．ただしO_s,O_b,O_lはそれぞれ共有メモリ書き込み，同期に要するオーバーヘッド，共有メモリ読み込みに必要な時間である．

（Ⅱ）p1,p2のいずれか一方のみが部分ブロックである場合：p1が部分ブロックであるとする．この場合（a）p1とp2を並列実行する場合（演算結果の取得時間は式(3)で表わされる）と，（b）p2の実行後続けてp1,cを実行する場合の2通りが考えられる．（b）の場合のcの演算結果の取得時間は

$$allget(c) = max(end(p2),allget(p2))+exec(p2)$$
$$end(c) = allget(c) + exec(c)$$　...(4)

で与えられる．cの取得時間には両者の内値の小さい方を選択すればよい．（a）が選択された場合p1を今後マイクロタスクブロックとして取り扱う．逆に（b）が選択された場合，p1,cはp1の後ろにcを付加した1つの部分ブロックとなる．

（Ⅲ）p1,p2がいずれも部分ブロックである場合：（Ⅱ）の場合と同様p1,p2を並列実行する場合と逐次実行する場合の2通りが考えられる．並列実行した場合のcの演算結果の取得時間は式(3)で表わされる．逐次実行した場合の取得時間は

$$allget(c) = max(allget(px)+exec(px) ,allget(py)) + exec(py),\ end(c) = allget(c) + exec(c)$$　...(4')

で表わされる．ただしpxにはp1,p2の内実行可能時間が小さい方，pyにはもう一方を与える．逐次実行を選択した場合はP1,P2,Cはpx,py,cの順に統合され，1つの部分ブロックに置き換えられる．並列実行が選択された場合p1,p2は今後共にマイクロタスクブロックであるとされる．

（Ⅱ）（Ⅲ）で部分ブロックが他の部分ブロックと統合される際に統合される部分ブロックの出力変数に通常変数が含まれる場合，その変数は別の代入文で参照される可能性があるためそのコピーを統合の対象とする．→［6］

3.2 マイクロタスクブロック抽出例

```
S1: x1 := x1 + x2 + x3 - x4
S2: x2 := x1 + x2 - x3 + x4
```

図2：代入文1

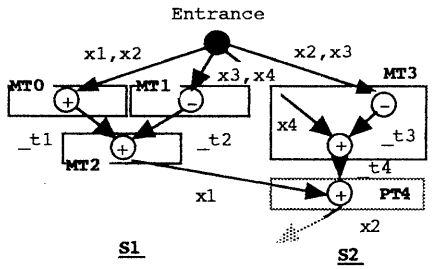

図3：代入文1の計算木

図3は図2で示される代入文S1，S2からマイクロタスクを抽出した例である．実線の枠で囲まれた計算木がマイクロタスクブロック,点線の枠が部分ブロックを構築する．抽出されたマイクロタスクブロックを以下に示す．

```
MT0: Begin  MT2: Begin    PT4: Begin
 _t1:=x1+x2  x1:=_t1+_t2   x2:=_t4+x1
End (20)     End (50)      End (80)
MT1: Begin  MT3: begin
 _t2:=x3-x4  _t3:=x2-x3
End (20)      _t4:=_t3+x4
             End (40)
```

図4：マイクロタスクブロック

図の括弧内の数字は加減算に20，共有メモリアクセス及び同期に要するオーバーヘッドを10とした時の各出力変数の取得時間を表わしている．TWRアルゴリズムにおけるx2の取得時間が80であるのに対し，THRアルゴリズムではその取得時間が110になる．

逆に原始プログラム中の1ステートメントがマイクロタスクを構成しない場合がある．この場合TWRアルゴリズムは1つのステートメントを複製し，同期／通信のオーバーヘッドを削減する．

```
S1: a := a + 1
S2: b := a * 2
S3: c := a * 3
```

図5：代入文2

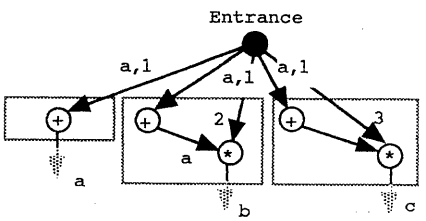

図6：マイクロタスクブロック抽出例2

この例ではS1の演算がS2,S3に吸収されている．

第4章　スケジューリング

4.1　逐次コードのスケジューリング

4.1.1　マイクロタスクグラフ

前節で述べた方法で抽出された中間コードの依存関係を表わすマイクロタスクグラフを生成する．このグラフでは中間コード間のデータの依存関係と制御依存関係を一意に表わす事が出来る．

マイクロタスクグラフでは制御依存する中間コードを真に制御依存するものとそうでないものに分けている．ここで言う真に制御依存する中間コードとは，その実行条件が明らかになる前に実行するとプログラムの意味が変わってしまうものを指す．真に制御依存するコードには（1）制御の合流点以降に参照される通常変数への書き込み,（2）外部入出力命令および（3）タスクブロックがある．

（1）の依存性を削減するために通常変数書き込みを行う中間コードは演算結果を一度一時変数に蓄え，後続の複写文のみが真に制御依存に依存するよう変更する．

図7,8にサンプルプログラムとそのマイクロタスクグラフを示す．

```
S1:  x1:=x1+x2+x3-x4
S2:  x2:=func(x3,x4)
S3:  if x1>0 then
S4:   x3:=x1+x2-x3+x4
      else
S5:   x3:=x1-x2+x3+x4
S6:  x4:=-x1+x2+x3+x4
```

図7：サンプルプログラム

図8中の実線で囲まれた中間コードはマイクロタスクブロック,点線は制御コード,斜線はタスクブロックを表わす．またグラフ中の各ノード（中間コードあるいはタスクブロック）にノードからイグジットノードまでの各パスの処理時間の最大値を与える．この値を最大パス長と呼ぶ．

条件分岐先の最大パス長は条件分岐コードよりも低いほうが望ましい．そこで最大パス長の設定時にはブロック中の全てのコードが真に制御依存するものとして考える．（図8にはこのためにダミー依存を示す辺と各ノードに与えられたパス長を表わしている．この最大パス長は加減算の実行時間を20,複写コードを5,分岐コードを10,関数funcの実行時間を50として計算している．）

以降ノードnのプレディセッサとはnが読み込む値を算出するノード，逆にサクセサとはnが算出する値を読み込むノード，フリーなノードとはプレディセッサが全て既にスケジューリングされたノードを指す．

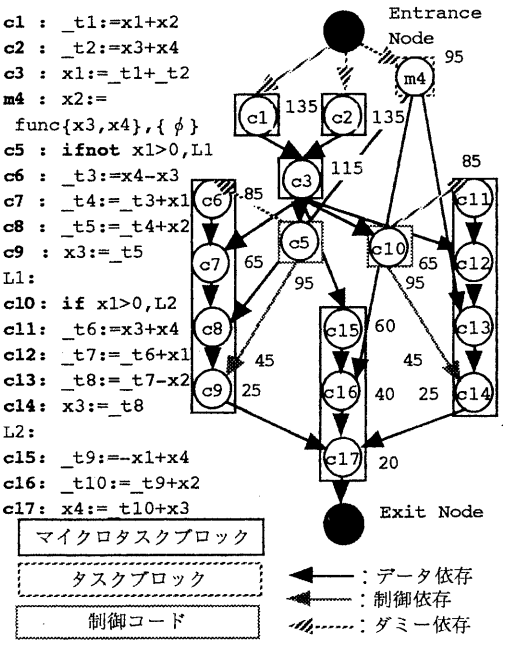

図8：サンプルプログラムのマイクロタスクグラフ

4.1.2　プロセッサへの割り当て

最大パス長の与えられたマイクロタスクグラフを用いて各中間コードあるいはタスクブロックをプロセッサに割り当てる．プログラムが使用可能なプロセッサ数をPUNOとし，プロセッサを$proc_i$(0≦i≦PUNO)でする．$proc_0$は空のタスクブロック生成に用いる．

［1］**準備**：delay(i)　(1≦i≦PUNO)をプロセッサiに割り当てられた中間コードおよびタスクブロックの実行時間であるとする．（初期値0）→［2］

［2］**ノードの取り出し**：タスクグラフ中のフリーなノードの内最も最大パス長の大きいものを取り出す．該当するノードが複数ある場合その後続ノード数が多いものを選択する．（それでも1つに定まらないときはランダムに選択する．）取り出されたノードをnとする．全てのノードをスケジューリングした時点でアルゴリズムは終了する．→［3］

［3］**ノードの判別**：もしノードnが分岐文ならば［4］，タスクブロックならば［5］，それ以外なら［6］へ進む．→［2］

［4］**制御コード**：nに制御依存するノードの集合をNとする．N中にn,N以外にスケジューリングされていないプレディセッサがある場合nのプロセッサへの割り当ては保留し［2］に戻る．該当するプレディセッサがない場合，nを$proc_i$(0≦i≦PUNO）に割り当てる．その後N内のノードを［2］で述べた優先順位順に取り出し，［6］によって処理する．nの分岐先のラベルを$proc_i$(0≦i≦PUNO）に割り当てる．各プロセッサに挿入したnの並びを分岐コード列と呼ぶ．→［2］

［5］**タスクブロック**：nがタスクブロックである場合，グループ全体のプロセッサの制御がそれに移るため，他のフリーなノードを，delay(i) (1≦i≦PUNO)の最大値が増加しない限り先に［2］によって割り当てる．その後タスクブロックをdelay(i)が他より大きくないpunum(n)個のプロセッサに割り当てる．タスクブロックが割り当てられなかったプロセッサには空のタスクブロックを割り当てる．各プロセッサに挿入したタスクブロック,空のタスクブロックの並びをタスクブロック列と呼ぶ．→［2］

［6］**ノードの割り当て**：ノードnのプレディセッサの集合をPとする．もしPの中でnと同一のマイクロタスクブロックに属するコードがある場合それが割り当てられているプロセッサにnを割り当てる．もしそれに該当するプレディセッサが存在しない場合，Pの属するプロセッサの中で最もdelay(i) (1≦i≦PUNO）が大きいものにnを割り当てる．Pが空の場合delay(i)が最小であるプロセッサにnを割り当てる．nを割り当てられたプロセッサのdelay(i)をexec(n)だけ増加させる．→［2］

4.1.3　同期命令挿入

ここでは同期命令の挿入アルゴリズムについて述べる．文中の同期命令列とは実行時に同期がとられる各プロセッサに挿入された同期命令，ノードnの支配バリア[7]をnの直前に存在する同期命令列，off_delay(n)をnの支配バリアの後ろの命令からnの実行が終了するまでの実行時間であるとする．（該当する同期命令が存在しない場合は最初のコードからnまでの実行時間とする．）

［1］**サクセサノードの取り出し**：全中間コード中のサクセサ命令の集合をSとする．Sの中からoff_delay(s)が一番小さいノードsを1つ取りだし［2］で述べる処理を行う．全てのSを処理した時点でアルゴリズムは終了する．

［2］**プレディセッサノードの取り出し**：sのプレディセッサ命令の集合P，Pの中でsと支配バリアが同一で別のプロセッサに割り当てられているノードの集合をP'とする．P'の要素の中からoff_delay(p)が大きいノードpから順に取りだし，［3］で処理する．→［1］

［3］**同期命令の挿入**：同期命令列をpの後ろ，

sの直前に挿入する．proc(s),proc(p)をs,pが割り当てられているプロセッサとする．以下にプロセッサproc(s)，proc(p) およびその他のプロセッサに分けて考える．

proc(s)：sの直前に同期命令を挿入する．sの直前に存在する分岐コード列をb，タスクブロック列をtとする．

proc(p)：proc(p)中でbに含まれるコードをb'，tに含まれるコードをt'とする．proc(p)に既に割り当てられている命令でb'，t'より後ろに存在し，off_delay(s)に最も近くかつ小さい命令nを求める．もしnがpより後ろに存在するならばそこに同期命令を挿入する．nがpより前に存在するならばpの直後に同期命令を挿入する．

その他：proc(p)内の命令でb'，t'より後ろに存在し，off_delay(s)に最も近くかつ小さい命令nを探し，その後に同期命令を挿入する．

4.1.4 タスクブロックの情報

スケジューリング結果において条件分岐先（if文のthen部,else部）に同期命令が挿入されなかった場合に分岐元と分岐先の間に中間コードが存在しない分岐コードが生じるのでこれを取り除く．また$proc_0$と全く同じコードのみが割り当てられてたプロセッサはタスクブロックの実行には関与しない．これらのプロセッサの台数を除いたものがスケジューリングされたタスクブロックが必要とするプロセッサ台数になる．生成されたタスクブロックの実行時間はタスクブロック中に挿入された同期命令列間のクリティカルパス，即ち同期命令列間の実行時間の最大値の和となる．

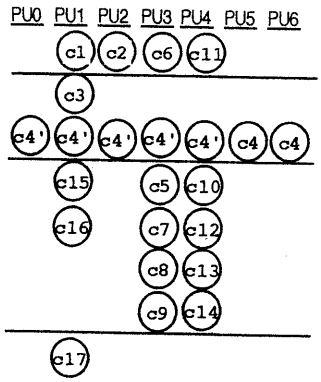

図9：サンプルプログラムのスケジューリング結果

図9にサンプルプログラムのスケジューリング結果について示す．タスクブロックm4は2プロセッサを用いて実行され，プログラムに与えられたプロセッサ台数は16とした．図のノード間に挿入された実線は同期命令列を表わしている．

4.2 ソフトウェアパイプライニングへの応用

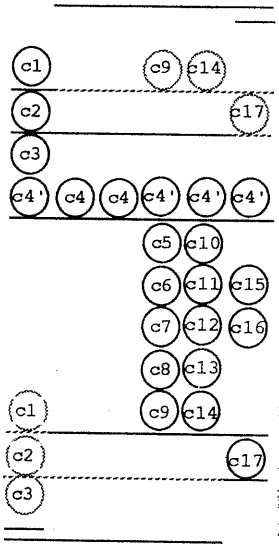

図10：ソフトウェアパイプライニング

TWRアルゴリズムがDOSERIALループのソフトウェアパイプライン処理にも応用できることを示す．DOSERIALループにおいてイタレーション間依存する各変数の取得時間はその直前に実行されるイタレーションの出力変数のそれであることが望ましい．そこでイタレーションを逐次コードと見なしてスケジューリングし，それによって得られた変数の取得時間をあらためて入力変数の取得時間としてスケジューリングを行う．

図10はサンプルプログラムをループのイタレーションとしてスケジューリングした例である．図中の実線は同期命令を表わしている．ループの実行開始前と後には全プロセッサの同期命令実行回数を同一にするために同期命令が補填されている．

第5章 並列計算機MSBMによる評価

5.1 ベンチマークプログラム

Whetstoneベンチマークプログラム[8]は主に科学技術計算用のプログラムを分析して，各種命令の実行頻度を考慮して作成された合成ベンチマークプログラムである．プログラムは以下に示す特徴を持った8個のモジュールから構成されている．

No.	処理内容
2	実数配列の四則演算
3	関数呼び出しと実数配列の四則演算
4	条件分岐文の性能測定
6	整数演算および変数の型変換
7	三角関数の演算
8	関数呼び出しおよび実数四則演算
9	関数呼び出しと実数配列参照
11	標準関数（sqrt,exp,ln）

各モジュールはfor文で表わされており，命令の実行頻度分析結果を反映するようなループの反復回数によって重み付けされている．全モジュールを構成す

るループは意図的にベクトル化出来ないよう設計されており，DOSERIALループのソフトウェアパイプライン処理でしか速度向上は計れない．全モジュール中の整数には16ビット固定小数点,実数には32ビット単精度実数を用いた．

Dhrystoneベンチマークプログラム[9]は整数演算を多用するコンパイラやエディタの命令実行頻度を解析した結果作られた合成ベンチマークである．プログラムは非常に処理内容の少ない手続きを多く含んでいる．ソースプログラムにはC言語で記述されたVer.2を使用した．プログラムは関数呼び出しやメモリコピー等の並列実行で効率を上げることの出来ないルーチンが多用されている．

スケジューリングは元のプログラムの構成/意味を忠実に反映するよう行った．逐次/並列版ともレジスタ割り当て以外の最適化（例えばインライン展開，式の数学的変換）は一切行っていない．並列化に際して各プログラムには１６台のプロセッサを与えてスケジューリングした．しかし実際に各プログラムが使用するプロセッサ台数は1～7台と低い台数に留まっている．これにはプログラム中にDOALLループが存在しない事も起因している．

5.3 結果及び検討

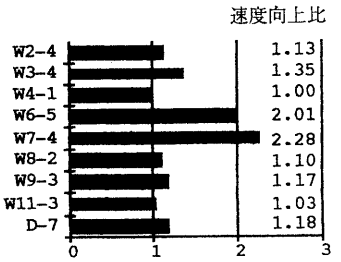

図11：速度向上比

図11にベンチマークプログラムをハンドコンパイルし，単一プログラム上の実行時間に対する速度向上比を求めた結果を示す，Wm-nのmはWhetstoneのモジュール番号，nは使用したプロセッサ数を表わしている．D-7は７台のプロセッサを用いて実行されたDhrystoneを表わしている．Dhrystoneプログラムの測定結果は原始プログラムから予想される速度向上率よりかなり低くなっている．現在原因を究明中である．

5 結び

原始プログラムのをタスクブロックなるコード領域に分割し，階層的にスケジューリングを行う方法について述べた．タスクブロックは幾つかの基本ブロックから構成される．TWRアルゴリズムは各基本ブロック中の代入文間/内の依存関係を考慮した上でマイクロタスクブロックを抽出する．抽出されたマイクロタスクの依存関係はマイクロタスクグラフで表わされ，制御依存性を軽減した上でプロセッサに割り当て同期命令を挿入するアルゴリズムについて述べた．またベンチマークプログラムを実際に並列化して速度向上率を測定したところ並列化が困難なプログラムであるにもかかわらず最大2.3倍の速度向上率を得た．

現在今回述べた並列化アルゴリズムを用いた自動並列化Pascalコンパイラを開発中である．またこれと同時に動的なバリア書換を考慮したタスクブロックレベルの並列実行可能性と各種ループの効率良い並列実行手法の追及が今後の研究課題となる．

参考文献

(1) 村岡洋一 "超並列処理コンパイラ",サイエンス社 1990

(2) A.D.PAUDA,M.J.WOLFE "Advanced Compiler Optimization for Supercompiters", 1986 CACM Vol.23, No.12, Dec. 1986[pp.1184-1201]

(3) J.Ferrante, K.J.Ottenstein, J.D.Warren "The Program Dependence Graph and Its use in optimization" ACM Trans. on Programming Lnag. and Syst, Vol.9, No.3, July 1987 [pp.319-349]

(4) 笠原博徳 "並列処理技術",コロナ社,1991

(5) 本石彰,浜田智雄,岩根雅彦,米沢俊夫 1994 "細粒度マルチマイクロプロセッサMSBMの構成と性能評価" SwoPP 琉球 '94

(6) Aho,A.V., Sethi,R., Ullman,J.D. "コンパイラ 原理・技法・ツール", サイエンス社,1986

(7) A.Zaafrani, H.G.Diets, M.T.O'Keefe "Static Scheduring for Barrier MIMD Architectures", 1990 international Conference on Parallel Processing,1990 [pp.II.187-II .194]

(8) H.J.CURNOW, B.A.WITCHMANN,"A Synthetic benchmark", The Computer 1976 J.19:1

(9) WEICKER,R.P. "Dhrystone Benchmark: Rationale for Version 2 and Measurement Rules" ACM SIGPLAN Notices,Vol. 23,No.8,1988 [pp.49-63]