RC-005

スレッドレベル並列化のためのスレッド間依存関係の分類
Classification of Inter-Thread Dependencies for Thread-Level Parallelization

森田 清隆† 布目 淳ǂ 平田 博章ǂ 柴山 潔ǂ
Kiyotaka Morita Atsushi Nunome Hiroaki Hirata Kiyoshi Shibayama

1. まえがき

マルチスレッドプロセッサ[1][2]やマルチコアプロセッサが商用のマイクロプロセッサとして一般に使用される時代を迎え，並列プログラム開発の重要性が高まっている．一方，プログラムの並列化については古くから研究され，おもに数値計算分野において配列を対象とした自動ベクトル化や並列化[3]で大きな成果が得られている．また，数値計算分野に限らず，命令レベルの並列化についても多くの研究がなされてきた．しかし，一般的なプログラムを対象に，スレッドレベルの並列性を抽出することは未だに困難な状況にある．その原因には，まず，コンパイラの観点では，入力データの値に基づく条件分岐などによってプログラムのどの部分が実行されるのかを静的に把握できない点や，ポインタや変数のシノニムの問題など，動的に生じ得るスレッド間の依存関係を静的に把握できない点などが挙げられる．また，動的に依存解析を行う場合には，並列性抽出のための解析範囲を広くすることが困難である．

プログラムからどの程度の並列性が抽出できるかを調査した報告[4][5]はあるが，どのような並列性抽出の技術を用いることを想定するかで結果は異なる．また，プロセッサアーキテクチャ研究の立場から，動的に依存解析を行いながら投機的な並列実行を行う方式[6][7]も提案されているが，本質的に逐次実行を想定してコンパイルしたコードを用いる限りは，その効果にも限界があるものと考えられる．

そこで，本稿では，プログラムがどの程度の並列性を内在するかを直接的に調査するのではなく，どのような要因が並列性抽出の障害となっているかを調べることで，スレッドレベル並列性の抽出可能性を探る．

2. 依存関係の分析方法

2.1 対象プログラム

本稿では，依存関係を分析するための対象プログラムとして，SPEC CPU2006 ベンチマークプログラム[8]の中の 453.povray を用いる．このプログラムはレイトレーシング法[9]を用いた画像生成プログラムである．レイトレーシング法とは，図 1 に示すように，視点に届く光線の経路をたどることにより，投影面の各ピクセルの色を計算する画像生成アルゴリズムである．各ピクセルの色を求める計算は互いに独立に行うことができ，本来，並列処理に向いたアルゴリズムである．しかし，アルゴリズムのレベルで並列性を有するにもかかわらず，そのアルゴリズムを用いて並列に処理することを前提として記述されていないプログラムから並列性を抽出することはいまだに不可能である．453.povray も逐次実行を前提として記述されており，各ピクセルごとの処理は，図 2 に示したループの本体として記述されている．本稿では，実際にこのプログラムを逐次実行してメモリアクセスに関するすべての履歴を採取し，図 2 のループ本体をそれぞれスレッドとして並列実行することを想定した場合にどのような並列実行の阻害要因が現れるかを調査･分析する．

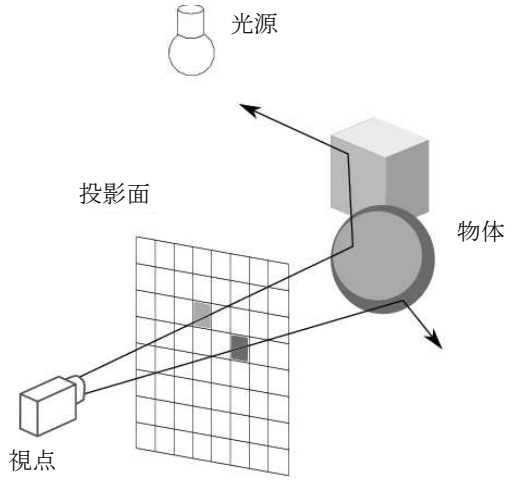


図 1　レイトレーシング法

```
for(line=First_Line; line < Last_Line; line++){
  for(clm=First_Clm; clm < Last_Clm; clm++){
        1 ピクセル分の処理        ループ本体
  }
}
```

図 2　並列実行対象のループ

2.2 メモリアクセス履歴の採取

453.povray を GNU C++コンパイラで -O2 の最適化オプションをつけて PowerPC[10]用にコンパイルし，GYPSI[11]を用いて作成したマシン命令レベルの機能シミュレータ上で実行する．入力には test データセットを使用する．このシミュレーションにより，実行時にアクセスしたメモリロケーションごとに，どのようなデータの値に対してどのようなアクセス（読み出しか書き込みか）をどのような順序で行ったかを，図 2 のループのイタレーションと関連付けて採取する．

† 京都工芸繊維大学大学院 工芸科学研究科 情報工学専攻
ǂ 京都工芸繊維大学大学院 工芸科学研究科 情報工学部門
Dept. of Information Science, Kyoto Institute of Technology

そして，そのメモリアクセス履歴をもとに，図 2 のループの各イタレーションをスレッドとして並列実行する場合に，どのような依存関係が存在するかを調査する．ただし，単一スレッド実行用のコードを用いてメモリアクセス履歴を採取するので，実際には，見かけ上の依存関係が大量に発生する．例えば，図 2 のループ本体で関数（メソッド）を呼び出す場合，あるイタレーションで呼び出した関数が使用するローカル変数領域と別のイタレーションで呼び出した関数が使用するローカル変数領域とが重なり，メモリアクセスに依存関係を生じてしまう．しかし，関数内で使用するローカル変数領域はその関数の実行を終了した時点で解放される性質のものであるため，それらは本質的な依存関係ではない．通常はスレッドごとにスタックフレーム領域を割り当て，また，453.povray の場合もそのようにしてスレッドを並列実行するものと想定して問題ないので，図 2 のループ本体から呼び出す関数については，スタックフレーム領域に対するメモリアクセス履歴はスレッド間の依存解析の対象とはしない（もちろん，図 2 のループが記述されている関数のスタックフレームよりボトム方向の領域については，依存解析の対象とする）．

以上のようにして採取したメモリアクセス履歴をもとに，スレッド間の依存関係を調査・分析する．基本的には，メモリロケーションごとにどのようなアクセスがどのような順序で行われるかによって依存解析を行うが，従来の並列化を目的とした場合のように，単に，フロー依存，出力依存，逆依存，の 3 種に分類するだけではスレッドレベルの並列化を目指すことは難しい．そこで，本稿では，投機実行や値予測などの技術の利用を視野に入れながら，さらにはセマンティックなレベルにまで及ぶ解析を行う．

3. 依存関係の分類

3.1 分類の概要

シミュレーションによるメモリアクセス履歴をもとにしてソースコードを分析し，スレッド間に依存関係を発生させる変数を分類した．まず，おおまかには，アクセス内容によるもの，他の変数との独立性に関するもの，プログラムの処理内容によるもの，の 3 種に分類した．

(a) アクセス内容による分類

各イタレーションにおいて実行されるアクセスの種類（書き込みか読み出し）とそのときの変数の値をもとに，さらに細かく以下の 4 種類に分類した．

- ライトスタート（write start）変数
- セイムリード（same read）変数
- 帰納変数
- ワンスライト（once write）変数

(b) 他の変数との独立性に関する分類

(a)に分類した変数を除いた残りの変数について，値の生成や条件分岐命令における条件判定に関して，同一のイタレーション内で他の変数から影響を受けるかどうか，または，他の変数へ影響を与えるかどうか，という観点でさらに以下の 2 種類に分類した．

- セルフアップデート（self update）変数
- ファイナル（final）変数

(c) プログラムの処理内容による分類

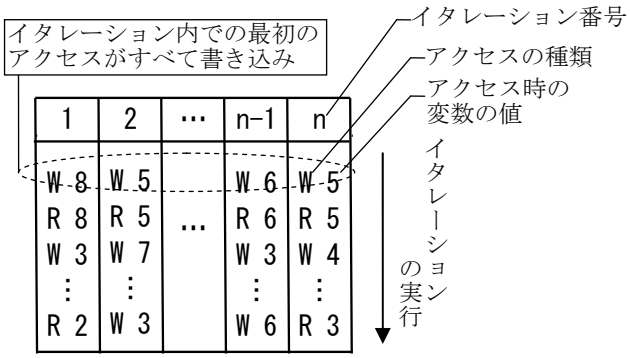


図 3 ライトスタート変数へのアクセス例

(a), (b)に分類した変数を除いた残りの変数について，プログラムの処理内容をもとに，さらに以下の 3 種類に分類した．

- エンブロイル（embroil）変数
- 動的再利用ポインタ変数
- キャッシュ（cache）変数

3.2 アクセス内容による分類

3.2.1 ライトスタート変数

ある変数にアクセスするすべてのイタレーションにおいて，最初のアクセスが書き込みである場合，この変数をライトスタート変数と呼ぶことにする．図 3 にライトスタート変数についての各イタレーションにおけるアクセスの様子を例示する．図 3 において W は書き込み，R は読み出しのアクセスを示す．また，各アクセスの横に書かれている数字はそのアクセス時の変数の値を示す．

ライトスタート変数にアクセスするイタレーションでは，少なくとも 1 回は書き込みを行うので，それらのイタレーション間に出力依存の関係が生じる．また，あるイタレーションでライトスタート変数の読み出しも行う場合は，そのイタレーションと後続イタレーションの間に逆依存の関係が生じる．

しかし，ライトスタート変数は，あるイタレーション中でのアクセスが書き込みで始まり，先行イタレーションにおいて生成したこの変数の値を使用しない．よって，ライトスタート変数については，イタレーション間に本質的な依存関係は存在しない．

3.2.2 セイムリード変数

ある変数にアクセスする各イタレーションにおいて，最初に読み出す値がすべてのイタレーションで等しい場合，この変数をセイムリード変数と呼ぶことにする．ただし，イタレーション内での最初のアクセスが書き込みであっても，その値がすべてのイタレーションで同一であれば，その変数もセイムリード変数に分類する．図 4 にセイムリード変数についての各イタレーションにおけるアクセスの様子を例示する．

セイムリード変数については，図 4 の例のように，各イタレーションで最初に読み出した値をそのイタレーションの最後で書き戻すと，先行イタレーションでの書き込みと後続イタレーションでの読み出しとの間にフロー依存の関

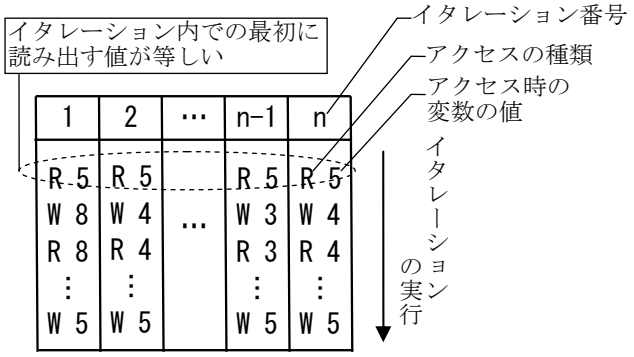


図 4　セイムリード変数へのアクセス例

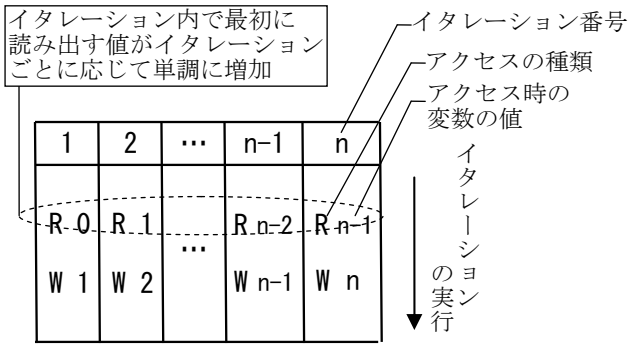


図 5　帰納変数へのアクセス例

係が生じる．また，先行イタレーションでの読み出しと後続イタレーションでの書き込みとの間で逆依存，先行イタレーションでの書き込みと後続イタレーションでの書き込みとの間で出力依存の関係がそれぞれ生じる場合がある．

しかし，セイムリード変数の場合は，各イタレーションで最初に読み出す値がすべて等しいので，その値があらかじめ分かるのであれば，各イタレーションを並列に実行できる可能性がある．このように，セイムリード変数はイタレーション間の依存関係による実行順序の制約を回避できる可能性がある．

3.2.3 帰納変数

各イタレーションにおいて，最初に読み出す値がイタレーションごとに一定値ずつ単調に増加または減少する変数を帰納変数[12]という．図 5 に帰納変数についての各イタレーションにおけるアクセスの様子を例示する．

帰納変数は，先行イタレーションにおける変数の値に一定の増分値を加算して後続イタレーションで用いる値を生成するので，先行イタレーションでの書き込みと後続イタレーションでの読み出しとの間にフロー依存の関係が生じる．

しかし，初期値とイタレーションごとの増分値があらかじめ分かるのであれば，各イタレーションで帰納変数の値を独立して生成することができるので，各イタレーションを並列に実行できる可能性がある．

3.2.4 ワンスライト変数

最初にアクセスされるイタレーションにおいてのみ書き込みが行われ，以降のイタレーションでは読み出しのみが

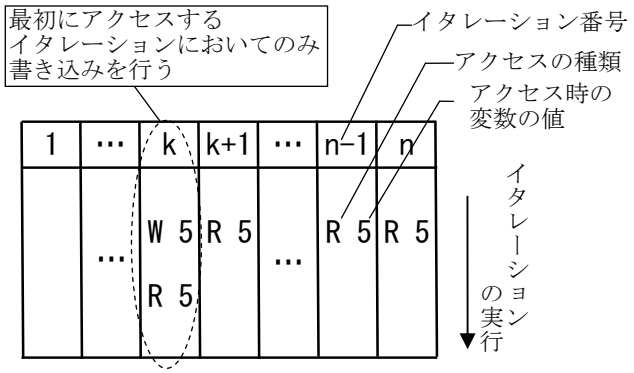


図 6　ワンスライト変数へのアクセス例

行われる変数をワンスライト変数と呼ぶことにする．図 6 にワンスライト変数についての各イタレーションにおけるアクセスの様子を例示する．

ワンスライト変数については，最初にアクセスするイタレーションとそれ以降のイタレーションとの間にフロー依存の関係が生じる．一般的には，このフロー依存の関係を取り除くことはできないが，ワンスライト変数に最初にアクセスするイタレーションを除けば，以降のイタレーションを並列に実行することが可能である．

3.3 他の変数との独立性に関する分類

3.3.1 セルフアップデート変数

同一のイタレーション内において，ある変数に書き込む値の生成に他の変数（ただし，関数内のローカルな変数を除く）の値を使用せず，また，その変数への書き込みアクセス自体が他の変数の値に基づく条件分岐の影響を受けない場合，その変数をセルフアップデート変数と呼ぶことにする．

図 7 にセルフアップデート変数の例を示す．図 7 では変数 A がセルフアップデート変数である．図 7(a)の関数 func()が図 7(b)に示す並列処理対象のループの本体で呼び出される場合，変数 A への各イタレーションでのアクセスは図 7(c)のようになる．変数 A の更新値は変数 A の値のみを用いて計算され，他の変数の値に影響されない．

図 7(c)のように，先行イタレーションで生成したセルフアップデート変数の値を後続イタレーションで読み出す場合，イタレーション間にフロー依存の関係が存在する．

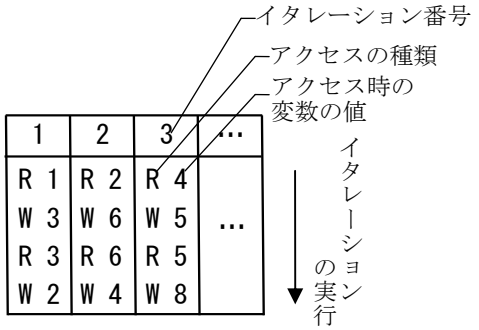
セルフアップデート変数の存在は，並列実行を行う上で大きな阻害要因となり得る．本稿で調査対象とした 453.povray では，乱数の生成に関連してセルフアップデート変数が現れる．レイトレーシングのアルゴリズム自体には並列性が備わっていても，逐次実行（単一スレッド実行）することを前提にプログラムが記述されているため，乱数の生成には並列性に乏しいアルゴリズムが使用されている．セルフアップデート変数によって生じる依存関係が避けがたい本質的なものであるか否かをプログラミング言語のレベルで判断することは困難であるが，ここでの使用目的が乱数生成であることから本質的な依存関係ではない．しかし，実際にこのような依存関係を取り除くには，アルゴリズムレベルでの対応が必要になる．

```
func( ){                    ← セルフアップデート変数
   static int A = 1;
   A = 2 * (A / 3) + 3 * (A %3);
   return(A);
}
```

(a)　セルフアップデート変数を用いる関数

```
for(i=1; i <= n; i++){
        …
   b = func( );
        …
   c = func( );
        …
}
```

(b)　並列処理対象のループ



(c)　セルフアップデート変数Aへのアクセスの様子

図 7　セルフアップデート変数の例

3.3.2 ファイナル変数

同一のイタレーション内において，他の変数（ただし，関数内のローカルな変数を除く）に書き込む値や出力値の生成に使用されず，また，それらの生成を制御する条件分岐の条件判定にも影響を与えない変数をファイナル変数と呼ぶことにする．

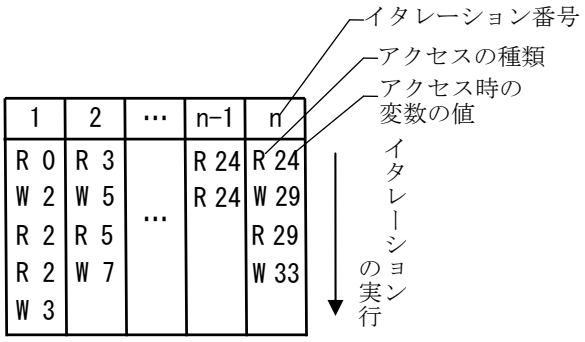
図 8 にファイナル変数の例を示す．図 8 では変数 Max がファイナル変数である．図 8(a)のコードは並列処理対象の各イタレーション内で不定回数実行され，変数 Max に対しては，イタレーション内において図 8(a)以外の形式ではアクセスされないものとする．この場合の各イタレーションにおける変数 Max へのアクセスの様子を図 8(b)に例示する．

ファイナル変数は，図 8 のように先行イタレーションで生成した値を後続イタレーションで読み出すので，イタレーション間でフロー依存の関係が生じる．

ファイナル変数は，イタレーション内で他の変数の値の生成に影響を与えることはなく，一般的には，並列処理対象のループの終了後に，その値が用いられる．ファイナル変数に起因するイタレーション間の依存関係は，並列実行する前提でプログラムを記述する場合には回避できる問題である．図 8 のように最大値を求める他に，最小値や総和を求める場合などが考えられる．自動ベクトル化コンパイラでも典型的なコードパターンに対してベクトル化を行っていたが，ここでのファイナル変数に関する問題はその一般化ととらえることができる．

```
if( value > Max){
   Max = value;
}
```
　　└── ファイナル変数

(a)　ファイナル変数を用いるプログラムの記述



(b)　ファイナル変数 Max へのアクセスの様子

図 8　ファイナル変数の例

3.4 プログラムの処理内容による分類

3.4.1 エンブロイル変数

イタレーション内での計算には本質的には無関係である（使用されない）にもかかわらず，構造体変数の一括代入として，アクセスされることにより，イタレーション間に依存関係が生じる変数をエンブロイル変数と呼ぶことにする．

エンブロイル変数には，一括代入する構造体変数のメンバである場合と，構造体変数の領域内に含まれるフラグメント（不使用領域）である場合の 2 通りがある．ソースプログラムで構造体変数の一括代入を行うと，オブジェクトプログラムのレベルでは計算に必要のないメンバやフラグメントのメモリ領域にもアクセスすることがあり得る．そして，そのアクセスが原因で，構造体メンバやフラグメントに関してイタレーション間で依存関係が生じる場合がある．

エンブロイル変数は，構造体変数の一括代入を行わずに，計算に必要な構造体変数のメンバだけをアクセスする場合は現れない．つまり，エンブロイル変数に関する依存関係を解消するためには，計算に必要な構造体変数のメンバだけをアクセスするようにソースレベルでの書き換え，またはコンパイラにおける配慮が必要となる．

3.4.2 動的再利用ポインタ変数

あるイタレーションにおいて動的に確保したメモリ領域をそのイタレーションの最後で解放することなく，後続イタレーションでも使用する場合がある．このとき，その領域を指すポインタ変数に関してイタレーション間に依存関係が生じる可能性があり，本稿では，このポインタ変数を動的再利用ポインタ変数と呼ぶことにする．

453.povray では，逐次処理においてメモリ領域の確保と使用するデータ構造に対する操作のオーバヘッドを減らす

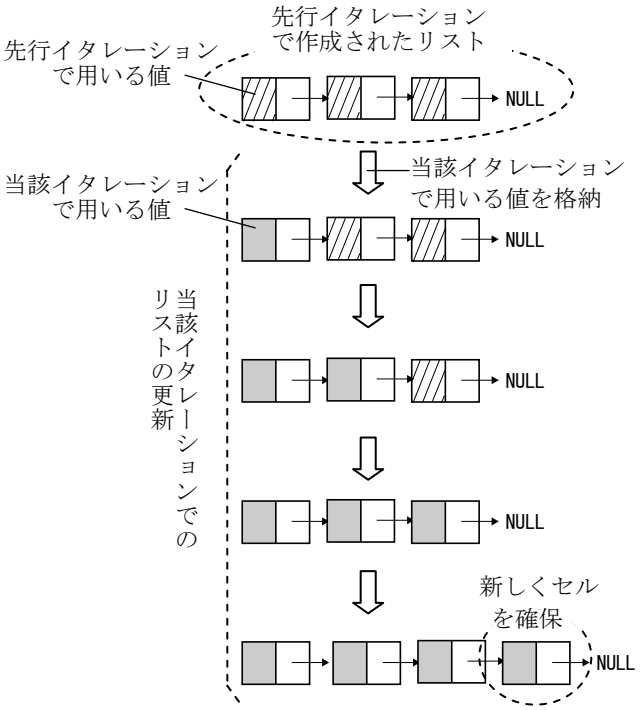


図 9　リストのメモリ領域の再利用

ために，あるイタレーションにおいて生成したリストや木などのデータ構造を，後続イタレーションにおいても用いている．図 9 に，あるイタレーションにおいて，先行イタレーションで生成したリストを用いる場合の動作について示す．

図 9 では，先行イタレーションにおいて生成されたリスト構造とそのメモリ領域を，当該イタレーションでそのまま使用し，このリストに当該イタレーションで用いる値を格納する．また，値を格納するリストのセルが足りなくなった場合は，新しくセルを確保し，このリストにつなげる．図 9 の場合，リストをたどる目的でリスト中のセルを指すポインタ変数の値を読み出すとき，そのセルのメモリ領域が先行イタレーションで確保されたものであるならば，先行イタレーションで書き込んだポインタ変数の値を当該イタレーションで読み出すことになる．したがって，イタレーション間にフロー依存の関係が生じる．

この動的再利用ポインタ変数は，各イタレーションごとに新たにリストを生成する場合には現れず，この場合のポインタ変数はライトスタート変数として出現するはずである．ただし，動的に割り当てられるメモリ領域の再利用については検討すべき課題が残されている．

3.4.3 キャッシュ変数

プログラムの実行時間を短縮する目的で，途中の計算結果などを一時的に保存しておく場合がある．このような目的で，先行イタレーションで計算した値をある変数に記憶しておき，後続イタレーションでその変数の値を用いる場合，その変数をキャッシュ変数と呼ぶことにする．

例えば，レイトレーシング法では投影面の各ピクセルの色を決定する過程で，物体のある点に光源から光が届いているかどうか，つまり，その点に届くはずの光源からの光線が他の物体に遮られて影になっていないかどうかを調べる必要がある．このために，光源からの光線を遮る物体を見つけるか，あるいは遮る物体がないことがわかるまで，光線と各物体との交差判定を繰り返す．この探索処理に関して，物体上の近傍の点に対する処理の間には，視点からたどる光の経路が類似するために，光線を遮る物体が同一である可能性が高いという特性がある．この視点間のコヒーレンスを利用して，453.povray では，前回の探索で光線を遮った物体を記憶しておき，まずその物体から光線を遮るかどうかを調べはじめることによって，探索に要する処理量を削減して高速化を図っている．前回の探索で光線を遮った物体を記憶しておくための変数については，先行イタレーションで書き込んだ値を後続イタレーションで読み出すので，イタレーション間に依存関係が生じる．

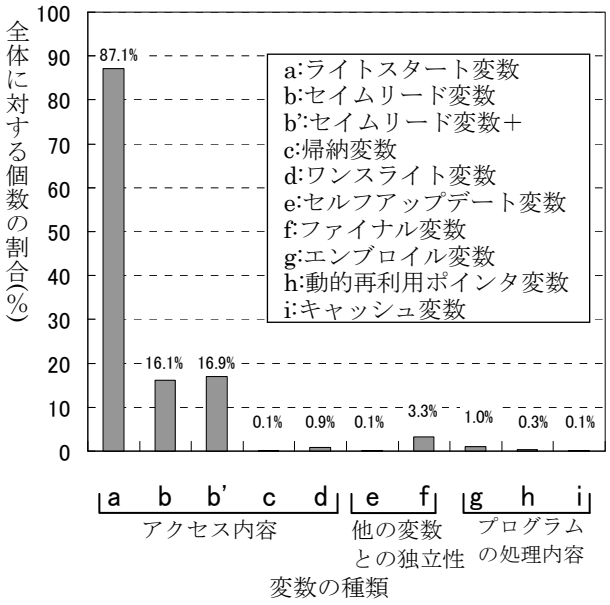


図 10　各変数の出現頻度

レイトレーシング法による画像生成プログラムにおいて，視線間のコヒーレンスを利用するか否か，またはどのように利用するかは，並列実行を考慮に入れながらアルゴリズムレベルで判断すべき事項である．しかし，このキャッシュ変数は，その使用目的から判断すると，プログラムの実行結果自体に影響を与えるのではなく，プログラムの実行時間に影響を与えるものである．この点に留意しながら有効な並列処理手法を見出すことができるか否かが，並列化を行う上での大きな鍵になるものと考えられる．

4. 分類ごとの出現頻度

453.povray において，イタレーション間にフロー依存，逆依存，出力依存のいずれかの依存関係を生じる変数の総数は 1024 個であった．これらを 3. で示した変数に分類し，その分類ごとの出現頻度を図 10 に示す．図 10 の横軸は変数の各分類を表わし，縦軸は出現頻度を表す．ここで，出現頻度は依存関係がある変数の総数に対する各変数の個数の割合を示す．

453.povray には，セイムリード変数には分類されないが，最初にアクセスするイタレーションを除いた残りのイタレ

ーションにおいて，セイムリード変数の特徴が表れる変数が存在した．図 10 では，このような変数とセイムリード変数の出現頻度を合わせてセイムリード変数+として示す．また，453.povray にはライトスタート変数とセイムリード変数の両方に分類される変数，セイムリード変数とワンスライト変数の両方に分類される変数，セイムリード変数+とワンスライト変数の両方に分類される変数が存在した．それぞれ依存関係がある変数の総数に対して，8.9%, 0.8%, 0.9%の割合で含まれていた．

図 10 より，ライトスタート変数の個数が最も多く，イタレーション間の依存関係の起因となる変数全体の 80%以上を占めている．ライトスタート変数はデータの値に関して本質的な依存関係を発生させるものではなく，単にイタレーション間で使用するデータの記憶域を共有することによって依存関係を生じるものである．セイムリード変数も本質的な依存関係の起因となる変数ではないが，ライトスタート変数とセイムリード変数の両者を合わせると全体の約 95%とほぼすべてを占める．今回の調査で並列処理対象としたループの各イタレーションを単に投機的に並列実行する場合を仮定すると，本質的ではない依存関係によって頻繁に投機実行の失敗を繰り返すことになる．しかし，ライトスタート変数とセイムリード変数による依存関係を取り除くことができれば，投機実行に失敗する回数は激減することになる．これは，並列性を有するアルゴリズムを用いていることを思い起こすと当然のことであるが，逆に，並列実行を意識しないで記述したプログラムがどのように並列化を阻害するかを端的に表わしていると考えられる．ただし，プログラムに与える入力データによっては，ライトスタート変数，セイムリード変数，ワンスライト変数の割合が変化する可能性があるが，ライトスタート変数とセイムリード変数とでほとんど全体を占める傾向は変わらないものと考えられる．

また，他の変数に分類されたものについても，全体に占める割合が少ないからという理由で，並列実行の可能性に与える影響が小さいとは限らない．例えば，あるライトスタート変数はごくわずかな数のイタレーションの間に依存関係を発生させるかもしれないのに対し，あるファイナル変数はすべてのイタレーションの間に依存関係を発生させるかもしれない．

5. むすび

遂次実行（単一スレッド実行）を念頭に記述されたプログラムから，スレッドレベルの並列性を抽出する際にどのような阻害要因が存在するかを，大規模なメモリアクセスのトレースを行って解析した．その結果，データ自体の本質的な依存関係ではなく，データの保存場所の使用に関する見かけ上の依存関係が多数存在することがわかった．

今回は，上記の目的から，アルゴリズムレベルで並列性を有するプログラムを対象として分析を行った．したがって，本稿で分類した依存関係の生成起因が必ずしも分類され尽くしたとは限らないが，ここで得られた結果は，他の多くのプログラムにも当てはまる点が多いものと考えられる．

今後は，並列性抽出の阻害要因に対する解決策を検討するとともに，他の一般的なプログラムも対象にその適用性・有効性を評価してゆく．

謝辞

本研究の一部は日本学術振興会科学研究費補助金（基盤研究（C）21500053，同 22500046 および若手研究（B）21700058）の補助による．

参考文献

[1] H. Hirata, K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase and T. Nishizawa, “An Elementary Processor Architecture with Simultaneous Instruction Issuing from Multiple Threads,” Proceedings of the 19th Annual International Symposium on Computer Architecture, pp.136-145 (1992).
[2] D. Tullsen, S. Eggers and H. Levy, “Simultaneous Multithreading: Maximizing On-Chip Parallelism,” Proceedings of the 22rd Annual International Symposium on Computer Architecture, pp.392-403 (1995).
[3] H. Zima, “Supercompilers for Parallel and Vector Computers,” Addison-Wesley (1991).
[4] M. S. Lam and R. P. Wilson, “Limits of Control Flow on Parallelism,” Proceedings of the 19th Annual International Symposium on Computer Architecture, pp.46-57 (1992).
[5] A. Nakajima, R. Kobayashi, H. Ando and T. Shimada, “Limits of Thread-Level Parallelism in Non-numerical Programs,” IPSJ Transactions on Advanced Computing Systems, Vol. 47, No. SIG 7(ACS 14), pp.12-20 (2006).
[6] G. S. Sohi, S. E. Breach and T. N. Vijaykumar, “Multiscalar Processor,” roceedings of the 22rd Annual nternational Symposium on Computer Architecture, pp.414-425 (1995).
[7] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson and K. Chang, “Tha Case for a Single-Chip Multiprocessor,” Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operationg Systems, pp.2-11 (1996).
[8] Standard Performance Evaluation Corporation, “SPEC CPU2006,” http://www.spec.org/cpu2006/
[9] 鷲島敬之, 西澤貞次, 浅原重夫, “並列図形処理,” コロナ社 (1991).
[10] Freescale Semiconductor, “Programming Environments Manual for 32-Bit Implementations of the PowerPC Architecture” (2001).
[11] 上堂薗浩光, 布目淳, 平田博章, 柴山潔, “プロセッサ開発支援システム GYPSI における命令シミュレーションライブラリ,” 電気関係学会関西支部連合大会, P.G230 (2007).
[12] A.V. Aho, R. Sethi, J.D. Ullman, “Compilers-Principle, Techniques, and Tools, ” Addison-Wesley (1986).