第７章　構造化プログラムの実行効率の分析

第７章　構造化プログラムの実行効率の分析

7.1　概　　要

一般に新しい言語を開発した場合，その言語機能の高機能化の代償として，処理系が生成するオブジェクトコードの実行効率が従来言語に比べて低下することが多い。そして，その度合が著しい場合は新言語普及の阻害要因となる。特に，本研究の対象であるＳＰＬは，応答性の厳しい制御用応用プログラムに適用することを目的としているため，オブジェクト効率の良否は重要な課題である。そこで，本章でその分析を行う。なお，ＳＰＬの稼動計算機が１６ビットマシンであることから，主記憶容量の制約が厳しく，プログラムサイズの増加は主記憶に常駐できないプログラムの増加を招く。そのための補助記憶装置との入出力処理の増加に伴い，実行速度が低下することから，本章では，所要メモリの増加についても分析対象とする。

従来の制御用 Fortran （ＰＣＬ）の代りにＳＰＬを用いることによるオブジェクト効率の低下は，主に構造化プログラミング機能に起因すると考えられる。ここでは，次の２つの機能について分析する。

(1)　段階的詳細化法やデータ抽象化技法に基づく階層化プログラミング機能。

(2)　goto 文を排し，制御構造を限定した構造化コーディング機能。

即ち，(1)はオブジェクト効率の低下がモジュール間にわたって生じると考えられるもの，(2)はモジュール内に限られるものである。

7.2　　階層化プログラミングの影響分析

7.2.1　オブジェクト効率低下要因

ＳＰＬでは，プログラムはモジュールの階層構造によって構成されるが，各々のモジュールはそれ自身で閉じたプログラムとして記述される。そして，このようなモジュール群は，コンパイル処理をされた後，下位モジュールがそれを引用している上位モジュールの中に埋め込まれることが多いため，オブジェクトプログラムは幾つかのソースモジュールを合成したものになっている。従って，このように，処理内容の局所性の高いモジュールを合成したオブジェクトは，

(1)　処理の冗長性の増大，

(2)　リンケージオーバーヘッドの発生，

(3)　作業エリアの増加

などを招き，オブジェクト効率が低下する。以下では，まずこれらの要因を分析した後，その対応策を検討する。

7.2.2　処理の冗長性

段階的詳細化技法における上位モジュールや，データ抽象化技法におけるデータ操作手続きの引用

側モジュールでは，引用するモジュールの機能だけ認識しておけば良く，その詳細な処理手順は知る必要がない。そのため，引用されたモジュールをインライン展開した時に冗長な処理が発生する場合がある。

例えば，ベルトコンベア上の特定の２区間の通過時間を求めるようなトラッキング処理を考える。区間の長さをＬＥＮＧＴＨ１，ＬＥＮＧＴＨ２とし，ｎ番目のベルトコンベアの速度は，VELOCITY OF NO（ｎ）という関数で求めるとすると，プログラムは，

```
ＴＩＭＥ１＝ＬＥＮＧＴＨ１／ＶＥＬＯＣＩＴＹ　ＯＦ　ＮＯ（Ｋ）　；
ＴＩＭＥ２＝ＬＥＮＧＴＨ２／ＶＥＬＯＣＩＴＹ　ＯＦ　ＮＯ（Ｋ）　；
```

と書けば良い。一方，この速度関数を次のように定義したとする。

```
function VELOCITY OF NO(N) : real opt(open)  ;
    read（３２，ＢＬＴＤＣＷ（Ｎ））ＷＯＲＫ　；
    return（ＷＯＲＫ＊３００）　；
end VELOCITY ;
```

この場合，インライン展開されたオブジェクトプログラムの該当部分は，

```
read（３２，ＢＬＴＤＣＷ（Ｋ））ＷＯＲＫ　；
ＱＱＱ００１＝ＷＯＲＫ＊３００；
ＴＩＭＥ１＝ＬＥＮＧＴＨ１／ＱＱＱ００１；
read（３２，ＢＬＴＤＣＷ（Ｋ））ＷＯＲＫ；
ＱＱＱ００２＝ＷＯＲＫ＊３００；
ＴＩＭＥ２＝ＬＥＮＧＴＨ２／ＱＱＱ００２；
```

となる。

この例では，オブジェクトの１，２行目と４，５行目が同じ処理をしており，冗長になっている。このような冗長な処理の生じる場合として，この例のような入出力処理や演算処理の他に，初期処理やエラーチェックなどが考えられる。

7.2.3　リンケージオーバーヘッド

階層化プログラミングでは，上位モジュールからの下位モジュール参照が多い。これらの下位モジュールのうち，外部手続きになるものは従来方式でも同様と考えられるので，そのリンケージオーバーヘッドは新方式によるオブジェクト効率低下要因ではない。

一方，インライン展開されるopen型手続きは新方式に特有のものである。このうち，値を返さないものは手続き本体が埋め込まれるので，リンケージオーバーヘッドは生じない。しかしながら，値を返す，いわゆる関数型の手続きは，コンパイラが割当てた作業エリアを介して値を返すようなオブジェクトになるため，この作業エリア，およびそのエリアに対する値の代入と参照処理が余分に必要となる。例えば，前項（7.2.2項）の速度関数の例において，作業エリアＱＱＱ００１およびＱＱＱ

００２がこれに当る。このオブジェクトは，速度関数を用いないで，直接，

ＴＩＭＥ１＝ＬＥＮＧＴＨ１／（ＷＯＲＫ＊３００）

と記述した場合に比べて，メモリ効率，処理効率共に低下している。

7.2.4 作業エリア

従来，一時的に必要となる作業エリアは，その使用目的とは無関係に共有することができた。しかしながら，階層化プログラミングにおいては，各々のモジュール内でしか使用しない作業エリアは，局所変数としてその中に宣言して用いるため，モジュール間での共有が困難である。このような例としては，演算結果の一時記憶エリア，入出力バッファエリア，ループ制御変数，実行制御のための状態変数などがある。

7.2.5 その対応策

以上に述べたような，階層化プログラミングによるオブジェクト効率低下要因に対して，ユーザおよびコンパイラの採りえる対応策について検討する。

まず，ユーザ側の対応策として，次のような項目が考えられる。

(1) 冗長なオブジェクトの生成を防ぐために，下位モジュールに対して，ＳＰＬのコンパイル時実行機能を用いてオブジェクトの最適化処理を記述できる。この１例は既に 4.6 節の図 4.3 に掲げているが，7.2.2 項の例では速度関数を次のように定義すれば良い。

```
function VELOCITY OF NO (N) : real opt(open) ;
   % if  FIRST
        then  read (32, BLTDCW (N))  WORK ;
              WORK  =  WORK  *  300 ;
              % FIRST  =  .false.  ;
        end ;
   return ( WORK )
end  VELOCITY  ;
```

(2) 冗長なオブジェクトの生成を防ぐために，(1)とは逆に上位モジュール記述時に，下位モジュールの処理内容を意識して，その引用方法を工夫する。例えば，先の速度関数の例では，

```
WORK  =  VELOCITY  OF  NO  (K) ;
TIME1  =  LENGTH1  /  WORK  ;
TIME2  =  LENGTH2  /  WORK  ;
```

とすれば良い。しかしながら，このような方法は，下位モジュールの処理詳細が決まっていない時は困難である他，ドキュメント性も悪くなる。

(3) 作業エリアを共有するためには，両者に共通な環境モジュールの中で共通変数として宣言した

り，メモリ属性のequivalent 指定を用いる方法がある。また，メモリ容量の大きいものは，引数渡しにする方法が考えられる。しかしながら，いずれの方式も，構造化プログラミングの主目的である理解容易性を低下させるため，好ましくない。

次に，コンパイラによる最適化処理として，以下の項目が考えられる。

(1) 関数型手続きの返す値に関するリンケージオーバーヘッドを防ぐためには，return 文の式を手続き引用部分に埋め込む方式が考えられる。これは，引数の受渡しの１方式であるcall by name と同様のことを返す値に関して行うもので，return by name とでも呼びうる方式である。この本式は常に可能なわけではないが，安全側の制約条件として，

(a) 式の項として参照され，かつその式が実引数でない，

(b) 参照された手続きがreturn 文を複数個持たない，

を満たす場合に限れば，最適化処理は容易である。これらの条件のうち，(a)は返す値のエリアが書き替えられないこと，また(b)は埋め込む式が一意に決まることを示しており，先の速度関数の例はこの条件を満たしている。

(2) 作業エリアを共有するような最適化のためには，各々の作業エリアが不要になる時点を正確に把握する必要がある。この処理を容易にするためには，安全側の判断として，

(a) 作業エリアを用いる手続きのインライン展開終了時，

を考えれば良い。ただし，(1)の方式を採用した場合は，インライン展開終了後も使用される可能性があるので，(a)の代りに，

(b) その手続きを参照している文のオブジェクト生成終了時，とすれば良い。

7.3 構造化コーディングの影響分析

7.3.1 従来方式との比較

構造化コーディングがオブジェクト効率に与える影響を分析するために，まず，従来の制御用Fortran の制御文を用いて記述されていた処理が，ＳＰＬではどのように記述されるかについて述べる。

(1) 単純goto 文

無条件分岐のための単純goto 文の処理は，ＳＰＬでは，

(a) exit 文

(b) if 文

(c) repeat 文

(d) 部分的処理の複写

(e) 制御変数の導入

のいずれかによって記述される。即ち，exit 文は，ある処理のまとまりを途中から脱出する時に用いる。if 文は，ある論理式の真偽に応じて異なる処理を行う時に用いていたgoto 文の代

りとなる。repeat 文は，ループを構成するための goto 文の代りとなる。

(d)は，既に分岐している処理の流れを合流させて，その後の処理を同じにするための goto 文の代りとなる。この goto 文は，exit 文やｉｆ文で代替できる場合も多いが，ＳＰＬでは分岐に対する合流はブロックの終りでしか行えないため，処理の一部を複写する必要のある場合が生じる。この複写すべき部分が多い時は，その代りに(e)の制御変数の導入によって代替することもできる。即ち，合流のための goto 文の所では，そのための制御変数に特定の値を設定して，自分の身近な所へ合流する。そして，それ以降で，かつ本来合流すべき所までの間の処理は，この制御変数を用いてスキップすれば良い。

(2) 計算型 goto 文

この構文は，文番号を k_j，整数型変数をｉとして，

goto (k_1, k_2, …), i

であるが，意味的には次の表現と等価である。

```
if ( i . eq .  1 )  goto  k1
if ( i . eq .  2 )  goto  k2
          ⎩
```

従って，この代替機能は，先に述べた単純 goto 文と後に述べる論理ｉｆ文の場合と同じになる。

(3) 割当て型 goto 文

この機能は，意味的には，assign 文の

assign k_j to i

の代りに，

i = k_j

また，

goto i

の代りに

```
if ( i . eq .  k1 )  goto  k1
if ( i . eq .  k2 )  goto  k2
          ⎩
```

と記述した場合と等価なので，計算型 goto 文と同じく，単純 goto 文および論理ｉｆ文の問題に帰着する。

(4) 論理ｉｆ文

これは，基本的にはＳＰＬでの else 節のないｉｆ文で代替できる。但し，論理式が真の時に実行する文が，本節で言及している制御文の時はその問題に帰着する。

(5) 算術ｉｆ文

この文の構文は，算術式をｅとして，

if (e) k_1, k_2, k_3

であり，意味的には次の表現と等価である。

```
if ( e .lt. 0 )  goto k1
if ( e .eq. 0 )  goto k2
if ( e .gt. 0 )  goto k3
```

これは，先の論理ｉｆ文の問題に帰着する。

(6) call 文

これは，外部サブルーチンの引用に用いる文であり，ＳＰＬの手続き引用と同じである。しかし，制御構造の観点では，この call 文の実引数には文番号が許され，引用されたサブルーチンから直接その文番号の所へ分岐する機能がある。この代替機能として，ＳＰＬでは，次の２つの方法がある。

(a) ブロック名を実引数とする。

(b) 制御変数の導入。

このうち，(a)は文番号による直接分岐と似ているが，exit 文と同じくブロックの脱出に限られる。(b)は，引用された手続き側で制御変数に特定の値を設定し，引用側に戻った後，その値に基づいて条件分岐をする方法であり，計算型 goto 文の問題に帰着する。

(7) その他

その他のＰＣＬの制御文のうち，ｄｏ文は repeat 文で，また continue 文は空文で代替できる。return 文と stop 文は同じである。

7.3.2 オブジェクト効率低下要因

以上に述べてきた代替機能では，オブジェクト効率の低下を招くものがあるが，大別すると次のようになる。

(a) 部分的処理の複写

(b) 制御変数の導入

(c) その他

このうち，(a)と(b)はフローチャート記述の段階で制御構造を変更するものである。その他は，コーディングの段階で異なる記述をするものである。以下では，これらの要因の分析と対応策の検討を行う。

(1) 部分的処理の複写

このような対応策によって，オブジェクトプログラムの実行速度は不要か，あるいは少し速くなるが，所要メモリ量は増加する。これに対するユーザ側の対応策としては，

(a) 処理内容の多いものは手続きとしてまとめる。

(b) 7.3.1(1)の(e)で述べた制御変数の導入。

が考えられる。一方，コンパイラの最適化処理としては，共通部分式の利用などが可能である。

(2) 制御変数の導入

本方式によるメモリ増加要因としては，

(a) 制御変数のストレージ

(b) 制御変数への値の代入処理

(c) 制御変数の値に基づく選択処理

があり，このうち，(b)と(c)は実行速度の低下要因でもある。これらの要因に対する対応策は見当らないが，いずれも効率低下の度合は小さいと思われる。

(3) その他

フローチャートの記述は従来と同じで，コーディングが異なるためにオブジェクト効率が低下するものとして，計算型goto文と算術ｉｆ文の代替機能がある。

まず，計算型goto文の場合は，従来のコンパイラは，分岐先のアドレステーブルを設け，制御変数の値でアドレス修飾して間接分岐をするようなオブジェクトを生成できる。これに対し，ＳＰＬを用いて分岐数と同数のｉｆ文を並べて処理するような記述では，制御変数の値の評価が分岐数と同じだけ行なわれる。この対策としては，ユーザはＳＰＬの多方向分岐用のｉｆ文を用いて，以下のように記述する。

```
if   i  is  1  then …
        is  2  then …
         ⟆
```

この場合は，コンパイラ側で従来と同様のオブジェクトコードを生成するような最適化処理が可能である。

次に，算術ｉｆ文の場合，従来のコンパイラは，１回の算術式の評価により３方向に分岐するオブジェクトを生成できる。これに対し，ＳＰＬではｉｆ文を２度用いて判定するため効率が劣る。これはコンパイラの共通部分式の最適化が有効であるが，もともと算術ｉｆ文は２方向分岐に用いられることが多く，効率低下は少ない。

7.4 実験と評価

7.4.1 評価の方法

オブジェクト効率が従来方式に比べてどの程度低下するかを調べるために，既存のＰＣＬプログラムをＳＰＬで書き直す実験を行った。この対象としては，ＰＣＬで記述された実システムの中から，配合水分制御システムの主プログラム（１９２行）を選んだ。そして，次の手順で実験評価を行った。

(1) ＰＣＬプログラムのフローチャート作成

(2) 対応するＳＰＬ用構造化フローチャート作成

(3) ＳＰＬプログラム記述

(4) 双方のコンパイル結果の比較

7.4.2 プログラムの変更点

ＰＣＬプログラムからの書き替えは，まずフローチャートレベルで行った後，コーディングしたが，

主な変更点を以下に示す。

(1) 制御変数の導入

プログラムの構造化のために，制御変数を2個導入した。これに関連した追加処理が7箇所に生じたが，その内訳は，初期値設定と判定処理が各々2箇所，値の設定が3箇所であった。

(2) 部分的処理の複写

プログラムの構造化のための部分的処理の複写が必要なものの中で，複写量の多いものは，(1)で述べたような制御変数の導入で対処した。一方，複写量の比較的少ないもので，そのまま複写したものが7箇所生じた。

(3) exit 文と begin ブロックの導入

ＳＰＬの選択文や反復文によって解消しないgoto 文が2個存在したため，begin ブロックを挿入して，goto 文をexit 文に変更するようにした。

(4) 外部手続きインターフェイスの変更

ＰＣＬプログラムでは，外部手続き（サブルーチン）引用時に，文番号を実引数にすることにより，引用された外部手続きからその文番号への直接分岐を行うものが2箇所あった。そこで，これらの実引数を文番号から制御変数に変更し，外部手続きから戻った後で判定分岐を行うようにした。

(5) open 型手続きの導入

ＰＣＬプログラムでは1つの主プログラムとして記述されていたが，ＳＰＬ用フローチャート作成時に処理の階層化を行い，main 型の手続きの他にopen 型の手続きを8個導入した。また，コーディングの段階で，2個の環境モジュールを導入し，上位の環境モジュールでは外部変数の宣言，また下位の環境モジュールでは，本プログラムの手続き間で共有する変数の宣言を行った。

7.4.3 オブジェクト効率の比較

まずはじめに，メモリ効率について検討する。元のＰＣＬプログラムのオブジェクトサイズは771Wであったが，前項で述べた変更点により，以下のようにメモリが増加した。

(1)	制御変数の導入	： ２６W
(2)	部分的処理の複写	： ２２W
(3)	exit 文と begin ブロックの導入	： ０W
(4)	外部手続きインターフェイスの変更	： １９W
(5)	open 型手続きの導入	： ０W

結局，全体では６７W，即ち8.7％の増加となった。

一方，実行効率については，実行のための外部環境作成が難しいため，実測ができず，その代りに机上で命令数の増加を見積った結果，6.0％の低下になった。その主な内訳は，先のメモリ増加要因のうち，制御変数の導入により2.8％，外部手続きインターフェイスの変更により3.2％である。

7.5 結　言

ＳＰＬを用いた構造化プログラムのオブジェクト効率を調べるために，従来方式との比較分析を行った。

まず初めに，段階的詳細化法やデータ抽象化法によって導かれるモジュールの階層構造によるオブジェクト効率低下要因を分析した。その結果，処理の冗長性の増大，モジュール間リンケージオーバーヘッドの発生，作業エリアの増加などの要因を明らかにすると共に，コンパイル時実行機能の活用などのユーザ側対応策，およびコンパイラの最適化処理による解決策を導いた。

次に，goto 文を排し，構造化コーディング用の制御文を用いて個々のモジュールの処理を記述することによるオブジェクト効率低下要因を分析した。その結果，従来言語の制御文に対するＳＰＬの代替機能には，部分的処理の複写や制御変数の導入などを必要とするものがあることを明らかにすると共に，これらの要因に対する幾つかの対応策を導いた。

最後に，プログラムの構造化によるオブジェクト効率低下の度合を定量的に把握するために，既存のＰＣＬプログラムをＳＰＬで書き直す実験を行った。その結果，制御変数の導入，部分的処理の複写，外部手続きインターフェイスの変更などにより，所要メモリ容量は 8.7 %増加した。また，実行時間は，机上で実行命令数を見積り，6.0 %の増加となった。これらの結果から，オブジェクト効率の低下がＳＰＬの実用化を妨げないことを確認した。

なお，オブジェクト効率低下要因への対応策として，本章では，ユーザ側で行えるものとコンパイラの最適化処理によるものを掲げた。しかしながら，ユーザ側対応策の多くは，プログラムの理解容易性を重視する構造化プログラミングの主旨に反するため，その適用は止むを得ない場合に限定すべきである。一方，コンパイラの最適化処理によって解決するものも多く，この問題は基本的にはコンパイラに任せるべきである。ただ，現在のＳＰＬコンパイラはＰＣＬプログラムをオブジェクトとして出力するプリプロセッサー方式を採用しているため，最適化処理が不十分であり，この問題は今後の課題である。