並列処理プログラムの構造

Structure of Parallel Programs

斎藤信男
Nobuo SAITO
筑波大学
University of Tsukuba

[1] はじめに

独立な制御の流れを持つプログラムが，同時に実行される環境は，従来でも，オペレーティング・システムなどで見られたが，最近，コンピュータ・コンプレックスなどの発達に伴い，その重要性が増大しつつある。将来は，安価なプロセサが大量に使用可能になると思われるが，それを有効に利用するためにも，並列処理プログラム（あるいは，非同期処理プログラム）をうまく作り，それを実現することが必要となる。

ここでは，まず，最も簡単な並列処理プログラムとして，従来の順次型プログラムの集合を考える。そこで，相互排除問題を記述し，並列処理プログラムの基本的に満すべき性質を議論する。

次に，並列処理プログラムの相互作用を書き易くするために，同期基本命令として，腕木システムを導入する。そして，この一般的な形を論ずる。

腕木システムを一般化してゆくと，抽象的な並列処理プログラムに到達する。その能率のよい実現は別問題として，そのような並列処理プログラムの問題点を議論する。

[2] 並列処理プログラムの基本的性質

非同期に処理される並列処理プログラムを，最も簡単に記述する方法は，従来の順次型プログラムを複数個用いることである。この場合，各順次型プログラムが共通変数を用いれば，相互作用が生ずる。

このような並列処理プログラムは，順次型プログラムと異なり，並列処理特有の性質を持っており，これらのプログラムが正しく動くためには，ある条件を満足していなければならない。これらの基本的条件を考えるために，Dijkstraが文献[1]で述べている相互排除問題の解を考えてみよう。

Dijkstraは，相互排除問題の解として，同期基本命令を用いず，共通変数への代入文，条件分岐文，go-to文だけを用いてプログラムを構成している。この正解に到達するために，いくつかのステップを示し，それぞれに欠陥があることを述べているが，それらの欠陥は，任意の同期問題に於ても除外しなければならないものである。Dijkstraのましたステップに従って，それらの性質を論じてみよう。なお，以下のプログラム（フロー・チャート）に於ては，変数は，2つのプロセスに共通のもの（全域的変数）と考える。また，CS1，CS2は，プロセス1およびプロセス2が，危険部分での処理をしている状態をましている。また，代入文，条件分岐文など，フロー・チャートの箱の中の操作は，非可分操作（indivisible operation）として実行されることを仮定しておく。

解1 （図1）

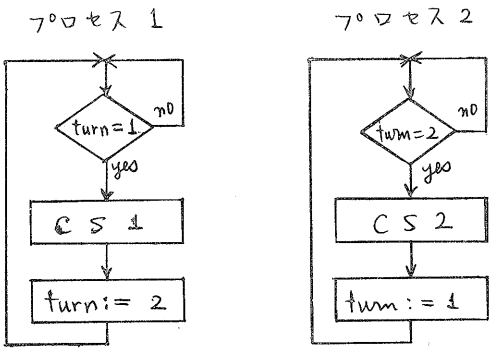
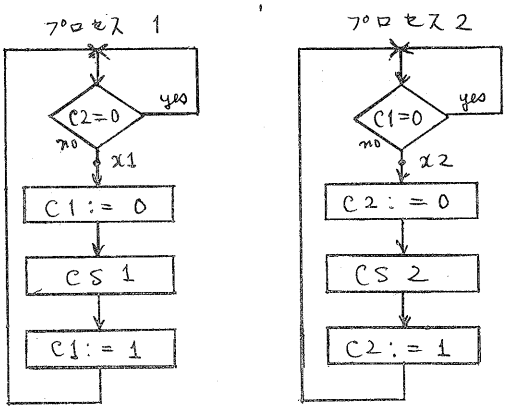


図1　解1　（初期値：turn＝1）

ここでは，turnという共通変数を1つだけ用いている。CS1，CS2に同時に入ることはないので，相互排除の命題を満足している。しかし，CS1およびCS2に入るのは，必ず交互にしか起らない。一般に，並列処理システムに於ては，異ったプロセスの動作は完全に独立していることが原則であり，危険部分に入る順序も全くランダムでなければならない。その意味で，この解は，正しい解とは言えない。

解2（図2）

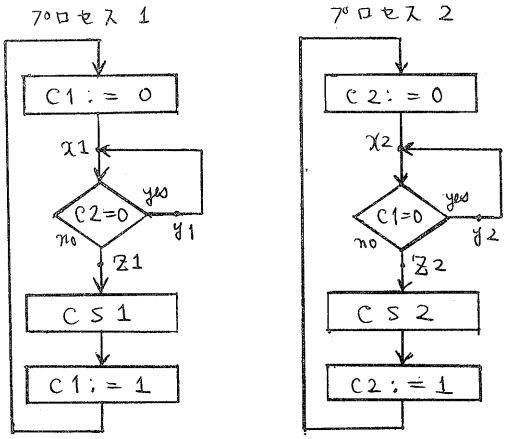


（初期値：C1＝C2＝1）

図2　解2

ここでは，C1とC2の2つの変数を用いているが，C1＝C2＝0のとき，あるタイミングによって，x1とx2の点に同時に到達してしまうことがあり得るので，CS1とCS2とに同時に入ってしまう可能性があり，相互排除という条件を必ずしも満足しない。

解3（図3）



（初期値：　C1＝C2＝0　）

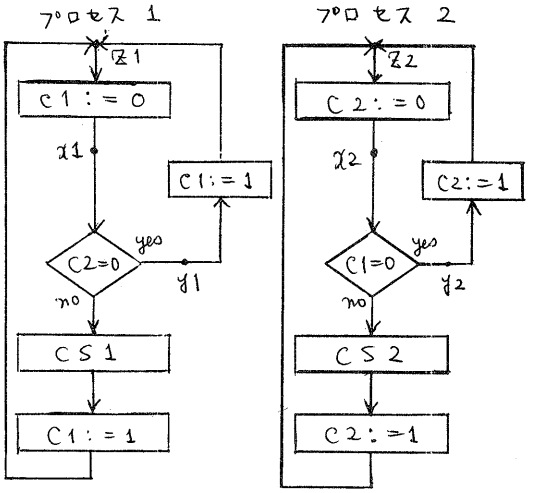
図3　　解3

ここでは，解2と異り，z1とz2の点に同時に到達してしまうことはあり得ないので，相互排除の条件は，満足される。一方，あるタイミングによって，x1とx2の点に同時に到達する可能性があるが，その状態に於ては，C1＝C2＝0であり，その直後の条件分岐命令は，両者とも"yes"に分岐し，結局，x1－y1およびx2－y2というループを，お互に永久にまわっているという状態になってしまう。このように，1ヵ所でお互に動きがとれなくなる状況を，デッドロック（deadlock）状態といい，並列処理プログラムでは絶対避けなければならない状態である。

解4　（図4）

解3に於ては，一旦x1，x2の点に同時に到達してしまうと，以後，C1，C2の値は変らないで，デッドロックのループに入ってしまう。解4では，y1，y2の点に同時に到達しても，C1:＝1　またはC2:＝1　の代入文を実行すれば，解3で生じたようなデッドロックの状態には陥入らない。しかし，これだけでは，未だ不充分であり，あるタイミングによっては，プロセス1は，x1－y1－z1のループを，プロセス2は，x2－y2－z2のループをお互にいつ迄も繰り返しているという状況が生ずる。

このような状態では，見かけ上は，何かプログラムが動作しているように見えるが，実際には，何も有益なことはしておらず，本来の目的である危険部分の処理は永久に行われないという結果になる。このような状況を，見かけ上のデッドロック（effective deadlock）といい，並列処理プログラムでは，やはり，避けなければならない状態である。

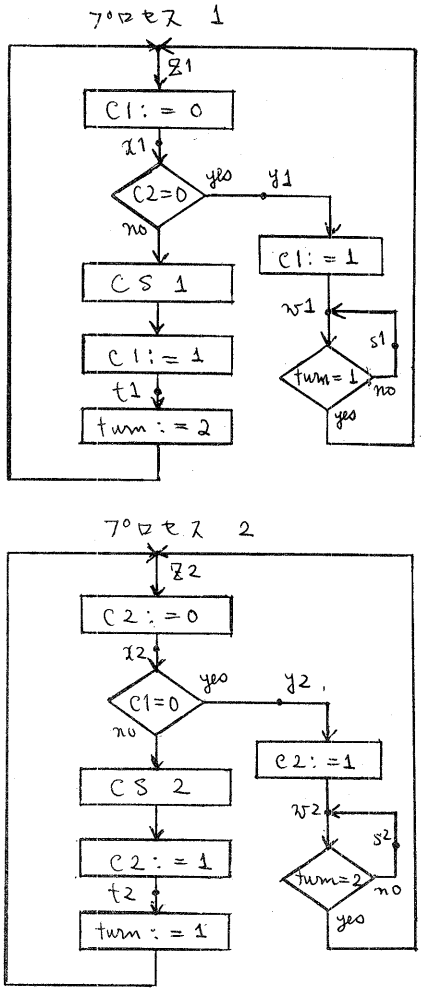


（初期値： c1 = c2 = 1）

図 4　　解 4

解 5　（図 5）　（Dekker のアルゴリズム）

解 4 で生じた見かけ上のデッドロックは，任意の時点に於て，両方のプロセスを全く平等に扱っていたことが一つの原因である。解 1 でのべたように，この原則は，本来守らなければならないものであるが，見かけ上のデッドロックを除去する際には，その原則を破って，ある時点に於て，どちらかのプロセスを優先的に扱えばよい。これは，解 1 にのべたように，何らかの順序関係に従って，両者の処理をすれば良いことになる。図 5 にのべた解は，w1－s1（w2－s2）に於ける変数 turn に関する条件分岐と，t1（t2）に於ける turn への代入文の 2 つの部分が，図 4 の解に追加されたものであるが，この追加の部分は，図 1 にのべた解 1 と同じ機能を果している。見かけ上のデッドロックを引き起すループに入ったならば，その時点での turn の値によって，どちらかのプロセスが優先的に処理される。図 5 にのべたものは，Dekker のアルゴリズムと等価なものであるが，それは，相互排除の条件を満し，かつ，デッドロック，見かけ上のデッドロックの状態に陥入る可能性を除去していることがわかる。したがって，これが，相互排除問題の正しい解である。



（初期値： c1 = c2 = 1； turn = 1）

図 5　　解 5

以上の議論からわかるように，並列処理プログラムが正しく動くということを検証するためには，次の条件が成立することを確かめなければならない。

(条件1)
与えられた問題に固有の性質を満たす。
(たとえば、相互排除)

(条件2)
各プロセスが、ランダムに実行される。

(条件3)
デッドロック状態が、決して生じない。

(条件4)
見かけ上のデッドロック状態が、決して生じない。

[3] 同期基本命令

相互排除問題を実現するような並列処理プログラムを記述するときに、共通変数に対する条件分岐と代入文だけを用いて行うと、2章に述べたように、かなり面倒なことになる。一般には、並列処理プログラム内の相互作用または相互通信を記述するために、同期基本命令(synchronization primitive)を用いる。この基本命令は、従来からいくつかの体系が提案されてきたが、最も一般的なものとしては、Dijkstraの提案した旗本システム(semaphore system)がある[1]。旗本システムを用いれば、大部分の同期の問題を記述することができる。また、いくつかの同期の問題(synchronization problem)を能率よく記述するために、旗本システムの拡張型が、いくつか提案されている。たとえば、PV Multiple[2], PV Chunk[3], 旗本アプリケーション[4]などである。これらの比較については、文献[5]などを参照されたい。

旗本システムを中心とする同期基本命令の共通の性質を考えて、その一般形を定義することができる[6]。

同期基本命令の一般形

(1) 同期変数(synchronization variable)
$\underset{\sim}{x} = (x_1, x_2, \cdots, x_n)$
各 x_i は、整数値をとる。

(2) 消費操作(consume operation)
P命令に対応する操作で、その詳細なフローを、図6にあす。ここで、$C(\underset{\sim}{x})$ は通過条件(pass condition)と呼び、また関数 $D: \underset{\sim}{x} \to \underset{\sim}{x}$ は、下降関数(down function)と呼び、一般に、$\underset{\sim}{x}$ に関する半順減少関数である。一般に、通過条件に対する判定分岐と、下降関数を用いた代入操作(図6の破線で囲んだ部分)は、非可分操作として実現しなければならない。

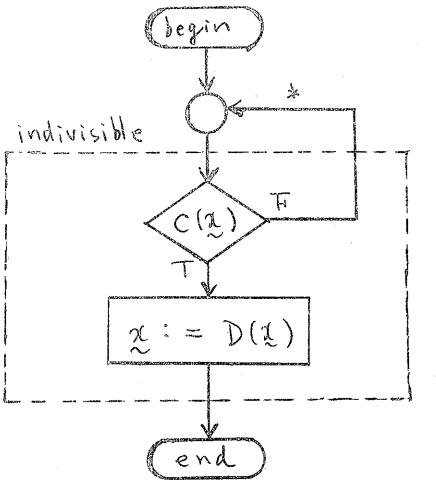


図6 consume operation のフロー

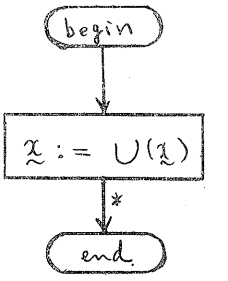


図7 produce operation のフロー

(3) 生産操作(produce operation)
V命令に対応する操作で、その詳細なフローを、図7にあす。ここで、関数 $U: \underset{\sim}{x} \to \underset{\sim}{x}$ は、上昇関数(up function)と呼び、一般に、$\underset{\sim}{x}$ に関する半順増加関数である。

なお、図6, 図7の定義は、ビジー待ち形式

(busy form of waiting) を用いている。同期基本命令の意味を説明するときには、この方が単純であるのでわかり易い。ただし、実際に実現するときは、下のレベルの同期基本命令を用いて、処理装置を解放した方が、能率が良い。

Dijkstra の提案した P, V 命令、あるいは、上述の一般形は、消費操作または P 命令においては、通過条件の判定を行うのに対し、生産操作または V 命令は、いつでも通過可能である。その意味で、両者の semantics は、非対称である。そこで、生産操作を一般化して、図に示すような定義をすることができる[7]。これは、消費と生産について、次に示すような対称的な原則で処理をしていることによる。

原則 I

「ある制限値以下しか存在しないものは、消費できない。」

原則 II

「ある制限値以上存在するものは、生産できない。」

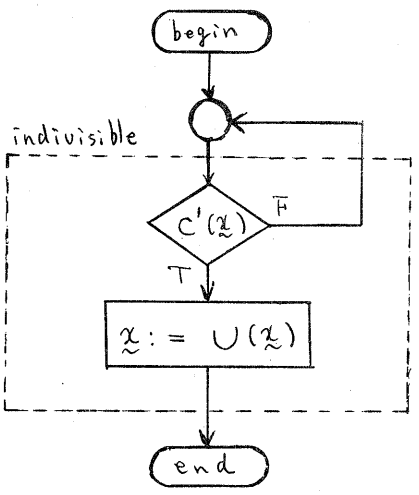


図 8　一般化した produce operation のフロー

[4] 並列処理プログラムの一般的モデル

同期基本命令を一般化してみると、ある通過条件を判定し、それが真であれば、上昇または下降関数を用いた代入操作を行っている。このような形式を一般化して、プログラムの任意の切片にかし、それを実行できるかどうかを決める通過条件を付加する。こうすると、次のような並列処理プログラムのモデルを定義できる。

モデル 1

並列処理プログラム P は、順次型プログラムの集合として定義される。

$$P = \{P_1, P_2, \cdots\cdots, P_n\}$$

ここに P_i は

$$P_i = (C_i, S_i)$$ で与えられる。

ただし

C_i : 通過条件（一般的なブール式）

S_i : プログラム本体（一般的な順次的プログラム）

この並列処理プログラムは、次のように処理する。

処理方式 1

(1) 通過条件 C_i が真であるようなプログラム P_i を見つける。

(2) それに対するプログラム本体 S_i を実行する。

これと似かよった考え方は、[4] の脱水アプリケーション、[8] の guarded command、[9] の sequence free computation などで用いている。

上述した処理方式では、ある章の同期基本命令のところで述べたように、正しい相互作用を行うことは不可能である。すなわち、通過条件の判定と、通過条件の値を変更する可能性のある代入操作とは、非可分操作として行わなければならない。これを実現するために、次のような変更を加える。

モデル 2

並列処理プログラム P は、次のように定義される。

$$P = \{P_1, P_2, \cdots, P_n\}$$

ここに、P_iは

$P_i = (C_i, S_i^m, S_i^d)$ でよこられる。

ただし、

C_i：通過条件

S_i^m：非可分に実行すべきセグメント

S_i^d：可分に実行してよいセグメント

このとき、並列処理プログラムは、次のような方式で処理される。

処理方式 2

(1) 通過条件 C_i が真となっているプログラム P_i を一つ見つける。

(2) 他の通過条件のscanは中断したまま、S_i^m の非可分操作を実行する。

(3) C_i のscanを再始し、それと共に、S_i^d を実行する。

このモデルを用いれば、同期基本命令を実現することができる。更に、順次型プログラムで従来から使われていた go-to 命令、繰り返し命令、判定分岐命令なども実現することができる。

モデルaで非可分に処理するプログラム・セグメントを考えたが、プログラム・セグメントは、全て非可分とすることもできる。そして、各プログラム P_i は、プログラム・セグメントの集合として考える。

モデルb

並列処理プログラム P は、次のように定義される。

$P = \{P_1, P_2, \cdots\cdots, P_n\}$

ここに、P_i は

$P_i = \{(C_i^1, \pi_i^1), (C_i^2, \pi_i^2), \cdots\cdots, (C_i^{k_i}, \pi_i^{k_i})\}$

でよこられる。

ただし、

C_i^k：通過条件

π_i^k：非可分に実行されるセグメント

この場合、各プログラム・セグメントには、それぞれ独立の通過条件をもたせてよい。また、P_i 中の1つのセグメントが終了すると、次に、P_i の中で行なわるべきセグメントが指定されるものとする。そして、P_i の通過条件は、そのセグメントに付いている通過条件と見做す。

この並列処理プログラムは、次のように処理する。

処理方式 3

(1) 通過条件 C_i が真となっているプログラム P_i を見つける。

(2) 他の通過条件のscanは中止したまま、P_i のcurrentのセグメント π_i^k を実行する。

(3) π_i^k の実行により、P_i のnextのセグメントから nextの通過条件が決まる。再び、(1)から繰り返す。

以上のような並列処理プログラムを具体的に実現するときは、特に、次の2点に関する配慮が必要である。

(I) 通過条件を、何点の論理式 (Boolean expression) とするか。

扱うべき変数を、制御変数(グローバルな変数)と、作業変数(必ずしも、ローカル変数でなくてもよい)とに分け、通過条件は、制御変数のみを用いて定義する。また、制御変数の値を参照、変更するのは、特別な関数として、システムが管理する。

(II) 通過条件を能率よく走査すること。

モデルaやbでは、非可分操作の存在が重要であるが、そのために、原則として、同時に、高々1個のセグメントしか実行できないことになり、能率が悪い。(せっかく、並列処理プログラムとしたのに、その並列性がもかせない。)

しかし、あるセグメントを実行しているときに、その中の命令によって影響を受けない(値の変わらない)ような通過条件を持つセグメントなら、並列的に実行しても差し支えないはずである。

[5] おわりに

並列処理プログラムが満すべき基本的性質について論じた。これは，並列処理ということから生じてくる特性であり，この性質を考慮して，正しいプログラムを作成しなければならない。

並列処理プログラムの相互作用を記述するための同期基本命令の一般形を述べたが，このような基本命令を用いずに，Hoareのモニタのような，もっと高度な基本命令をユーザは用いるべきであるという議論もある。

同期基本命令の一般化として，並列処理プログラムのモデルを提示した。このモデルの解析，意味の定義，例の記述など，まだ追及すべき問題として残っている。

参考文献

[1] E. W. Dijkstra: Co-operating Sequential Processes, Programming Languages (ed. F. Genuys), Academic Press, New York, pp. 43–112 (1968)

[2] S. S. Patil: Limitations and Capabilities of Dijkstra's Semaphore Primitives among Processes, CSG Memo 57, Project MAC, M.I.T., (1971)

[3] H. Vantilborgh and A. van Lambsweerde: On an Extension of Dijkstra's Semaphore Primitives, Information Processing Letters, Vol. 1, No. 5, pp. 181–186 (1972)

[4] P.L. Wodon: Still Another Tool for Synchronizing Co-operating Processes, Carnegie-Mellon Univ. (1172)

[5] 斎藤: OSの基礎理論(1), 情報処理, Vol. 15, No. 11, pp. 887–899 (1974)

[6] 斎藤: 同期基本命令の実現について, 情報処理, Vol. 15, No. 11, pp. 841–849, (1974)

[7] 斎藤: 腕木システムの一般化, 情報処理, Vol. 16, No. 11, pp. 1024–1027 (1975)

[8] E. W. Dijkstra: Guarded Commands, Nondeterminacy and Formal Derivation of Programs, CACM, Vol. 18, No. 8, pp. 453–457 (1975)

[9] 森: private communication