勉強会資料
— 並列処理言語 —

津邑 公暁

平成7年 5月 16日

1　導入

ハードウェア技術の発達により計算機の速度は目覚しい向上を遂げてきたが，それにも限界がある。にも拘らず計算機が処理する問題は大規模化しているため，並列処理の必要性が高まっている。また大量生産による汎用処理装置の低価格化により，超大型計算機を作るより複数の処理装置による並列処理システムの方がコストパフォーマンスがよくなる傾向にある。こういう点からも，並列処理言語の必要性は高まっている。

またソフトウェアの大規模化に伴い，その作成・保守・修正が困難になってきている。これを解決するにはプログラムのモジュール化，データの抽象化，情報隠蔽，オブジェクト指向プログラミング等が提案されているが，これらを実現するには並列処理言語が適している。

並列処理言語には二つの流れがある。一方は従来の逐次処理言語に並列処理機能を付加するものであり，もう一方はデータ・フロー言語のように新しい観点に立つものである。

並列処理システムにおいて並列に実行される，複数の処理間の情報交換を如何に記述するかが並列処理言語の基本的課題となっている。

並列処理言語の特徴は，並列動作が記述できるのは勿論のこと，論理構造に基づくモジュール分割が可能であることである。逐次処理言語ではモジュール分割が処理の流れに依存してしまい，一つの論理的機能を一つのモジュールに対応させることは難しい。この点並列処理言語では制御の流れの影響を受けずに，データや手続きの抽象化が可能である。

2　並列処理言語の機能

並列処理言語に必要な機能は，並列動作が記述できること。並列に動作するプロセス間の相互作用を記述するために必要となる，プロセス間通信の手段。そして非決定性である。以下それぞれについて説明する。

2.1　並列性

並列性を明示する方法には大きく分けて二つある。一つは各プロセスを一つのモジュールとして宣言し，並列分岐によって記述する方法。もう一方は一つのモジュール内で並列実行可能な部分を指定する方法である。

図 1: プロセス記述の分類

また，プロセスの起動方法によっても分類することができる。プロセスを動的に起動できる場合，すなわち起動されるプロセスの数が入力データによって変化し得る場合と，静的にしか起動できない場合，すなわち起動されるプロセスの数が固定されている場合とである。

- **cobegin/coend**

  cobegin S1; S2; ...; Sn coend という形式をしている。起動されるプロセスの数は固定されている。記述方法が簡単であるので，細かなレベルの並列性を記述するのに適する。

- **fork/join**

  fork で複数のプロセスを起動し，join で終了する。fork 命令は繰り返し実行することができるため，プロセスの起動を動的に行える。

以上二つの例は，一つのモジュール内で並列分岐を記述する方法である。この方法は言語機能が単純で済む上，効率のよいコード生成が行える。その反面，並列に動作するプロセスの全てから変数が見えてしまうことから，変数のアクセス制御が困難となり，モジュール化もできないという欠点がある。

これに対し，各プロセスを一つのモジュールとして定義する方法なら，プログラムの階層的分割が可能で，変数の管理も容易である。また制御の流れにとらわれず論理構造に基づくモジュール分割が行える。しかし並列動作部を全てモジュール化する必要があるため，細かな並列動作の記述には不向きである。

プロセスを一つのモジュールとして記述する場合，その起動方法にも静的な場合と動的な場合がある。プロセスの定義を含むモジュールが実行開始した時点で，自動的にそのプロセスが起動されるのが静的な場合。自動的にプロセスが起動されることはなく，そのプロセス定義を参照可能なモジュール内で起動命令を陽に実行するのが動的な場合である。静的な場合は，起動されるプロセス数は当然そのモジュール内で定義されているプロセス数に等しい。

プロセスの入れ子については，プロセスをモジュールとして定義した場合は内部モジュールが親モジュール内で定義されてた名前を参照できるため，同期や排他制御が問題となる。よってプロセスをモジュールとして記述する言語は大部分がプロセスの入れ子を禁止している。

2.2 プロセス間相互作用

並列に動作するプロセスが協力して処理を進めるには，プロセス間で情報交換を行う必要がある。これをプロセス間通信という。プロセス間通信は，受け手が情報を受け取るまで送り手が待たされる同期式と，そうでない非同期式に分類できる。

同期式通信では，送信側は情報が受信されたことを確認でき，またバッファも不要であるが，待ちの間送信側は処理を行えない。同期式通信を実現する方法としては，情報受理のチェックを繰り返し行い続ける busy waiting(繁忙待機) と，情報受理が完了するまで実行を中断する方法とがある。busy waiting はなんら生産的な仕事を行わずに CPU 時間を浪費してしまう。

非同期式通信は高度な並列性を実現することができるが，バッファ管理の必要があり，また情報の受理を確認できない。

一般に一方向通信を基本とする場合には非同期式，双方向通信を基本とする場合には同期式が向いている。

2.2.1 同期と排他制御

プロセス間通信を実現しようとするとき，同期や排他制御が問題となる。

プロセス間で同期をとるということは，ある事象が起きるのを待つことである。逆にいえば, 待たれている (すなわち起こることが期待されている) 事象を引き起こすプロセスが存在し，その事象が起きたときそのプロセスは他のプロセスにそのことを知らせる必要がある。プロセス間の同期の手段は，並列言語の基本機能としてシステムルーチンの形で提供することが多い。

逐次再使用可能な資源に対しては常に一つのプロセスしか使用できないよう管理する必要がある。これが排他制御である。ここで，あるプロセスが排他制御を必要とする資源を使用中に，他のプロセスがその資源を使用しようとすれば，そのプロセスは資源が解放されるのを待たねばならない。つまり排他制御を実現するには同期手段が必要である。

同期や排他制御の記述を誤ると，待ち状態になったプロセスが永久にその状態から抜け出せなくなる危険性がある。これを deadlock という。

2.2.2 ペトリネット

ペトリネットは並列動作を記述するための形式的モデルである。制御の流れを抽象化したもので，並列プログラムの性質を明らかにするのに用いられる。

ペトリネット N を $< T, P, A, B^0 >$ で表される有向グラフと定義する。

$N = < T, P, A, B^0 >$

$T=\{t_1, t_2, ...t_m\}$:有限個の transition の集合

$P=\{p_1, p_2, ...p_n\}$:有限個の place の集合

$A=\{a_1, a_2, ...a_e\}$:有限個の有向枝の集合

$B^0 \subset P$:token の初期配置

place は丸，transition は棒，token は黒丸で表す。transition は事象に，place は条件に対応し，ある place に token が存在することは，その place に対応する条件が成立していることを示す。

図 2: transition の発火

並列動作をペトリネットで記述することにより，そのシステムが deadlock に陥る危険性の判定等が可能になる。以下に具体的な問題例とそのペトリネットによる記述を記す。

- producer/consumer 問題

  producer は消費されるべき情報を作り，consumer がその情報を処理する。producer は consumer の進行状況とは無関係に新しい情報を作り出すことができるが，consumer は producer が情報を作り出すまで待たねばならない。

  producer と consumer の間には無限長のバッファが存在すると仮定されている。

図 3: producer/consumer 問題

- 排他制御問題 1 つの逐次再使用可能な資源を N 個のプロセスが共有するとき，同時には 1 個のプロセスしかその資源を使用できないように管理する問題。N=2 の場合を図 4に示す。

図 4: 排他制御の問題

- reader/writer 問題

  次の 2 種類のプロセスが 1 つの資源を使用する。

  writer : 資源を排他的に使用
  reader : 他の reader と同時使用可能
  writer とは同時使用不可

  reader が 2 つ，writer が 1 つの場合を図 5に示す。この例では writer が資源要求をして待たされているときに，新しい reader が資源要求をすると，writer を差し置いて reader の使用が許可される。この場合を reader 優先という。

  これに対し一度 writer が資源要求すると，それ以後の新しい reader の要求が許可される前に writer の要求が満たされるという条件をつ

図 5: 読み書き問題 (reader 優先)

ける。この場合で，reader と writer が各 1 個の例を図 6に示す。

図 6: 読み書き問題 (writer 優先)

- 喫煙家の問題

  煙草を吸うために，きざみ煙草を紙に巻き，マッチで火をつけなくてはならない状況を考える。3 人の喫煙家がそれぞれきざみ煙草，巻紙，マッチのうちの 1 つずつを持ってテーブルに座っている。各喫煙家は自分の材料を他人に与えることは許されない。テーブルの中央に 3 つの材料のうち任意の 2 つだけが置かれる。そこで残りの 1 つの材料を持っている喫煙家だけが煙草を吸うことができる。これをペトリネットで表すと図 7のようになる。

  今までの問題と異なり，この問題は 2 つの資源で競合が生じる。そのため 2 つの材料を 2 人が 1 つずつ取った場合は，deadlock に陥ってしまう。

- Dining Philosopher 問題

  5 人の哲学者が図 8のようにテーブルに座り，食事と思索を交互に行う。各哲学者の間にはフォークが 1 本ずつあり，テーブルの中央にスパゲッティが用意されている。哲学者は自分の両脇にあるフォークを持たないとスパゲッティを食べることができない。

  各哲学者が食事と思索それぞれに費やす時間が予め分からないため，前もって食事の順番

を決めてしまうと効率が悪い。よってちょうど食事を始めようとしている哲学者達の間だけで，競合を解決する必要がある。

図 7: 喫煙家の問題

図 8: 哲学者の配置

図 9: Dining Philosopher 問題

2.3 非決定性

複数の実行系列の可能性が存在する場合それを唯一に決める必要はない。むしろ決定性のプログラムにしてしまうことは他の可能性を否定してしまうことであり，望ましくない。また，非決定的な表現を可能にすることにより記述が簡単になる。

非決定性とは実行可能な命令系列が複数存在することを意味する。これには二つの解釈が可能である。一つは，個々の命令系列が出す結果の中で，与えられた条件を満たすものを全体の答えとする場合。この場合は無制限のバックトラックが必要である。もう一つは，個々の命令系列から生じる結果の中でどれを全体の答えとしてもよい場合である。この場合は全ての命令系列が同じ結果を生じることが多い。

非決定性の考えを進めたものに guarded command がある。次に実行すべき文の選択を制御す

るために各文の前に論理式を置き，その論理式が真のときだけ後ろの文が実行される。一般の条件文と違うところは，それぞれの条件が互いに排他的である必要がないという点である。複数の論理式が真になる場合は，その中の一つが非決定的に選ばれる。guarded command は次のような形式をしている。

$$\mathrm{B_1 \rightarrow S_1, B_2 \rightarrow S_2, ..., B_n \rightarrow S_n}$$

非決定的記述を一つのプロセス内だけでなく他プロセスとの関係に用いることができると，相手の応対に応じて自分の態度を変えるような記述ができるようになり，情報交換による deadlock の危険は減少する。

2.4 例外処理

例外処理とは例外的な事柄，つまりエラーや緊急時の情報交換が生じたときの対応を記述する部分を表す。

通常の情報交換は，情報を交換する二つのプロセスの双方がそのためのコードを実行して初めて行われる。しかしある種の情報交換は，受け手プロセスがそのためのコードを実行していなくとも行われる必要がある。但しある種の情報を常に受け取れるように通常の処理に混在して記述すると，プログラムが煩雑になる。よってこれは通常の処理と分離し，例外処理として記述する。

3 プロセス間通信

個々のプロセス間で行われる情報交換をプロセス間通信と呼ぶ。実現方法としては主に以下の 4 つがある。

- 変数を直接共有する。
- 共有変数を管理するモニタなどを介する。
- メッセージを交換する。
- 相手プロセス内の手続きを呼び出す。

以下，それぞれの方法について説明する。

3.1 共有変数

これがプロセス間通信を実現する最も簡単な方法である。共有変数に情報を読み書きすることによって情報のやり取りが可能になる。

排他制御問題をソフトウェア的に解決しようとする方法には，

- 単純な排他制御にも，複雑な記述が必要
- busy waiting

という大きな欠点がある。これらを避けるために提案された方法の主なものに，セマフォと臨界領域がある。以下これらについて説明する。

3.1.1 セマフォ

非負の整数値をとるセマフォS に対し，P と V の二つの操作がある。

P(S): $\mathrm{S > 0}$ ならば $\mathrm{S := S - 1}$
$\mathrm{S = 0}$ ならば，待ち状態に入る
V(S): 待ち状態にあるプロセスが存在すれば，そのプロセスを実行状態に移す。
存在しなければ，$\mathrm{S := S + 1}$

このとき，セマフォ値は使用されていない資源の数を表す。ここで，P，V 操作はそれぞれ不可分に実行されねばならない。つまり，セマフォという共有変数を排他的にアクセスする必要がある。

この方法により，P 操作を実行しようとするプロセスがセマフォの値を繰り返し調べる (busy waiting) 必要がなくなる。

このセマフォの拡張の一つとして，セマフォ値が負の値をとれるようにしたものがある。セマフォが負の場合，それは待ち状態にあるプロセスの数を表す。

また，複数の資源で同時に競合が起こる場合に対応するため，一度に複数のセマフォを扱えるように拡張したものもあるが，待ち行列の管理が難しくなってしまう。

セマフォを用いると同期と排他制御の両方を簡単に記述できるが，デッドロックに陥り易いという欠点がある。

3.1.2 臨界領域

共有変数に対して一度に1つのプロセスしかアクセスできないように制限すれば，同期や排他制御に関する問題は解決できる。これを実現するためのものとして臨界領域 (Critical Section) がある。臨界領域とは次のような文として定義される。

region v **do** S

これは共有変数vと文Sを対応づけるものであり，共有変数の使用は対応する臨界領域の内部に限られるので，コンパイラがそれをチェックすることが可能である

臨界領域について以下のような仮定を設ける。

- あるプロセスが臨界領域に入ろうとすると，有限時間内に入ることができる。
- 同一の共有変数に対しては，一度に1つのプロセスしか臨界領域に入ることができない。
- 臨界領域内のプロセスは有限時間内にそこから出なければならない。

異なった共有変数に対応する臨界領域は同時に実行しても構わないし，入れ子にすることもできる。しかし入れ子にした場合deadlockになる場合がある。その例を以下に示す。

```
cobegin
  A: region v1 do region v2 do...  ;
  B: region v2 do region v1 do...  ;
coend
```

Aが領域v1に入ると同時にBが領域v2に入ってしまうと，AもBも待ち行列に入ったまま抜け出すことができない。これは3番目の仮定に反する。

臨界領域の概念を拡張したものに，条件つき臨界領域がある。これは任意の条件が成立するまでプロセスを待ち状態にさせるもので，次のような形式である。

region v do begin... **await** C; ...end

この場合，共有変数vが条件Cを成立させるまでプロセスを待たせる。awaitは臨界領域の内部でしか使用できない。awaitによって待ち状態になっている間は，共有変数vに対する排他制御は解除され，他のプロセスが臨界領域に入る。awaitによって待ち状態になったプロセスは条件が成立し次第，臨界領域に入るべく他のプロセスと競合する。

3.2 モニタ

資源の定義とその資源への操作とを一ヶ所に集めることによって，資源利用の際に内部の詳細を知る必要がなくなる。

```
モニタ名:monitor;
  begin
    状態変数の宣言
    procedure 手続き名
      begin
        手続き本体
      end;
    他の手続きの定義
    状態変数の初期化コード
  end;
```

図10: モニタ定義

モニタは資源の状態を保持する状態変数と，資源への操作を実現するグローバルな手続きからなる。状態変数はローカルであり，手続きを介してのみアクセス可能である。

モニタの入れ子は禁止されている。共有変数に比べると，アクセス方法が限定される分エラー原因は減少するが，モニタは全てのモジュールからアクセス可能である点がエラーの原因になり易い。

状態変数へのアクセスは排他的に行われなければならないため，手続き実行は排他制御されるのが普通だが，実際には同時実行可能な手続きの組み合せも存在する。そこでモニタの概念を拡張し，モニタ内の手続きの同期や排他制御を記述する方法として，次のようなものがある。

3.2.1 condition 変数

condition変数はモニタ内でのみ宣言される変数であり，モニタ内の手続き実行を同期させるために使用される。

condition変数に対してsignalとwaitという二つの操作が定義される。

wait を実行した手続きは wait 状態になる。一方 signal を実行したとき，wait 状態にある手続きが存在しなければ，signal を実行した手続きの処理を続行する。wait 状態にある手続きが存在すれば，signal を実行した手続きは一時的に suspend 状態となり，wait 状態にあった手続きを一つ再起動する。suspend 状態になった手続きは，モニタ内に実行すべき手続きがなくなったときに起動される。手続きが二つの場合の様子を，図 11 に示す。これにより同一モニタ内に複数の手続きが存在可能であるが，同時に実行できるのは一つの手続きだけである。

図 11: condition 変数による同期

3.2.2 パス式

各手続きの並列性に関する制約条件をモニタ内の一ヶ所にまとめて記述する方法の一つで，全体の理解，記述が容易である。

パス式は各手続きの実行順序に関する制約を記述するものであり，同期命令を書く必要がない。記述方法は以下の通り。

path <パスの並び> **end**

パスの並びには次のようなものがある。

- p; q; r

  手続き p,q,r の順に逐次実行しなければならないことを表す。

- p, q, r

  p,q,r は互いに並列実行することが不可能であることを表す。

- p; {q; r}

  先ず p を実行し，次に q;r という系列を複数個並列に実行してもよいことを表す。

パス式は実行順序の表現に優れている。また手続きの動作記述と，同期や排他制御に関する制約の記述を分離した点で有用だである。しかし，状態変数に依存するような並列実行可能性の記述ができない。

3.2.3 並列指定

手続き間の同期と排他制御を分離するという考え方で，同期操作は個々の手続きの中に記述し，排他制御 (並列実行の可能性) は別に記述する。

同期に関する記述は既述の condition 変数を用いれば可能である。排他制御は，parallel 句で並列実行可能な手続きの組み合せを記述することによって実現する。parallel 句は次のような形式である。

parallel 手続き$_1$ **with** 手続き$_2$,
⋮
手続き$_{2n-1}$ **with** 手続き$_{2n}$;

parallel 句で指定されない手続きの組は排他制御される。また，ある手続きが他の手続きを呼び出す場合，前の手続きの排他制御はそのまま残るのが普通であるが，前の手続きが呼び出しのスケジューリングに用いられている場合などは排他制御を解除できると効率がよくなる。それを実現するのが exit 句である。exit 句は次のような形式をしている。

exit 手続き$_1$ **to** 手続き$_2$,
⋮
手続き$_{2n-1}$ **to** 手続き$_{2n}$;

to より左側がモニタ内，右側がモニタ外の手続きである。左側の内部手続きが右側の外部手続きを呼び出すとモニタの排他制御は解除され，外部手続きから戻ると再び排他制御がなされる。

この方法は排他制御の記述が静的であるため，パス式よりも理解が容易であるが，以下のような欠点がある。

- 状態変数の値に依存するような並列可能性の指定ができない。
- condition 変数により動的に同期問題を記述しているため，エラーが起き易く，deadlock の原因になる。

- 並列実行の可能性が対称でない (可換でない) 場合の記述ができない。

3.3　メッセージ通信

共有変数を使わない情報交換の方法にメッセージ通信がある。これは知らせたい内容を直接やり取りするものであり，送信側と受信側がそれぞれ送信命令と受信命令を実行してはじめてデータのやり取りがなされる。受信命令が出されるまで送信側が待つ同期式と，受信側に関係なく送信側はメッセージを送信できる非同期式とがある。

非同期式の方が多くの並列処理が可能となるためよく用いられるが，メッセージが順に受信される保証がなく，途中のメッセージが最後まで受信されないで残る可能性もある。

メッセージ通信のための機能の一つに send/receive がある。これは以下のような形式で表される。

send ＜式の並び＞ **to** ＜受信者＞
receive ＜変数の並び＞ **from** ＜送信者＞

通信路を指定する時期には 2 種類ある。一方は静的なもので，コンパイル時に通信路が確定しており，通信路の指定には定数が使われる。もう一方は動的な方法で，コンパイル時には確定できず実行時に通信路を選択する。

静的な指定法は指定が簡単で実行効率も高いが，実行時に選択ができないためプログラムの柔軟性に欠ける。動的な指定法では通信路自身をメッセージとして送ることなども可能になるが，効率的な実現が難しい。

通信路として何を指定するかによっても 2 種類に分けることができる。相手プロセスを直接指定する方法と，通信路としてのポートを指定する方法である。

相手プロセスを直接指定する方法は 1 対 1 通信には適しているが，1 対多や多対多の通信には適さない。メッセージを複数のプロセスのうちどのプロセスが受信してもよい場合，あるいは受信する際どのプロセスから送信されたメッセージでもよい場合などは，記述が困難あるいは不可能となる。

これに対しポートを用いる方法は，1 対多や多対多の通信を基本としている。1 つのポートに対して send/receive を行うプロセスがともに一つである場合，これは 1 対 1 通信になるが，一般にはともに複数存在する。

図 12: ポートによるメッセージ通信

3.4　遠隔手続き呼出し

メッセージ通信が一方向通信を基本としたものであるのに対し，遠隔手続き呼出しは双方向通信を基本としている。

プロセスの定義には変数宣言，動作記述といった従来の宣言の他に，遠隔手続きの宣言を含む。プロセスは自分のローカル変数にしかアクセスできず，共有変数は存在しない。

情報交換を行おうとする場合は，相手プロセス内の遠隔手続きを call し，呼び出した手続きが終了するまで待たされる。データのやり取りは引数転送で行われる。

呼ばれた遠隔手続きがいつ実行されるかという点で，2 つの方法がある。プロセスは先ず通常の処理を実行し，それが終了するか待ち状態に入るまで実行を続ける。処理が中断すると他プロセスからの外部要求 (遠隔手続きの呼出し) によって別の処理が始まる。次にこの処理が中断すると，別の処理が始まるか前の処理が復活する。以下これが永遠に続き，通常の処理が終了した後もプロセスは存在し続け，外部要求を受け付ける。この方法は分散プロセスで採用されている。

これに対し Ada では，遠隔手続きを受け付けるための文として accept 文を用意している。

accept ＜手続き名＞ (＜仮引数＞)
do ＜手続き本体＞ end

これを実行すると，指定された手続きが呼び出されるまで待たされる。そして入力引数と，文のスコープ内の変数を用いて手続き本体を実行する。

手続きが終了すると，出力引数を呼出し側のプロセスに返し，次の文に制御を移す。この方法をランデヴと呼ぶ。

前者では，いつどのような要求を受け付けるかという制御を行えない。つまり呼び出される側は完全に受身になってしまうが，後者ではいつどの手続きが呼び出されるかを指定できるため明確な記述が可能である。

4　データフローモデル

従来の逐次処理言語のように制御の流れを中心にプログラムを記述するのではなく，データの流れを中心に記述するという考え方から生まれた言語をデータフロー言語という。

データフロー言語の実行にはデータフロー計算機という， von Neumann 型計算機とは基本的に異なるマシンが必要である。

4.1　データフローグラフ

データの流れに基づいた計算方法をグラフ化したものをデータフローグラフという。これは従来の言語のフローチャートに対応するものであり，制御の流れではなくデータの流れを表す。

例えば $a+b\times c+d+e\times f$ という計算を考える。従来の方法では次のような順序で計算する。

(1) $X1 = b \times c$

(2) $X2 = e \times f$

(3) $X3 = a + X1$

(4) $X4 = X3 + d$

(5) $Answer = X4 + X2$

この式を解析木で表すと図 13のようになる。ここで節点 A はすぐに計算できるが，節点 B の計算は節点 A の計算がが終了した後でないと行えない。このようにこの図は計算の依存関係を表している。演算装置を多数用意しておき，実行可能になった演算から順に実行して行けば最大限の並列性が実現できる。これがデータフロー計算である。

データフローグラフは処理間の依存関係を表した有向グラフであり，節点で処理，枝で情報の記憶が行われる。情報にはデータと制御信号があり，それぞれ中黒，中抜きで矢印や token が表される。

節点には token を分岐させる link と，入力に対し処理を行う actor がある。以下いくつかの actor について説明する。

図 13: $a+b\times c+d+e\times f$の解析木

図 14: リンク

図 15: リンクの発火

- T-gate

  制御入力が真の場合にのみ入力データをそのまま出力する。制御入力が偽の場合は何も出力されない。

- F-gate

  T-gate の逆で，発火条件は制御入力が偽であることである。

- merge

  制御入力と一致する枝の入力を出力する。

図 16: 基本 actor の種類

データフローグラフの例として，Z=n!を計算するグラフを図 17に示す。図で示された状態から動作すれば，最終的に n!の値が下から出力される。このグラフは n の値を 1 ずつ現じる部分と，その値を掛け算する部分，ループを制御する部分の 3 つに分かれている。

データフローグラフは各演算子の実行に関して，最大限の並列性が実現できる。しかし，利用者がプログラムとしてデータフローグラフをそのまま入力することは不可能である。またグラフ表現自身はモジュール化に向いていないので，人間が表現する手段としても不向きである。そこで，人間がプログラム言語で表現したものをコンパイラ等が自動的にグラフ化し，計算機で実行する方法がとられる。

図 17: Z=n!を求めるグラフ

4.2 データフロー言語

データフロー計算を表現するための言語である。制御の流れを記述するのではなく，命令間の依存関係の記述を最小限にする。

データフロー言語を用いてソフトウェアを記述するには，以下の点が重要である。

- 副作用からの解放

副作用[1]があると並列性の検出が困難になる。

- 影響の局所性

  データ間の依存性が局所的なこと。

- プログラムの実行結果はデータの依存関係だけで決まる

- 単一代入規則

  一つの変数には一回の代入しか許されない。

- 繰り返しのための特別な技法

  前項「単一代入規則」のために繰り返しが不可能となるため，特別な記法が必要である。

- 手続きに履歴依存性がない

  呼び出しが次回の呼び出しに影響を与えない。

このような点を考慮したデータフロー言語には，Ida，VAL，Lucid などがある。

4.2.1　単一代入規則

同一の変数に対する代入文を一度しか許さない規則。代入文で左辺の変数は右辺に現れる変数に依存する。しかし，通常は同一変数に何度も代入が行われるため，いつ代入された値に依存するかが問題となる。これを決定しようとすると制御の流れを追跡しなければならないが，制御の流れは入力データによって変化するのでコンパイル時に決定することは不可能となる。これを解決するために単一代入規則が採用される。

単一代入規則では代入文は，変数の定義文とみなすことができる。例えば”S:=X+Y;” という文があれば，”S=X+Y”という関係は常に成り立つ。単一代入規則を導入することにより，データ依存関係が明らかになるだけでなくプログラムの検証も容易になる。

4.2.2　繰り返しの記法

単一代入規則を満たすためには，繰り返しに特別な記法を用いる必要がある。このためデータフロー言語では特定の形のループしか作らせないよう，ループのための手段を制限している。一般にループは以下の 4 つで構成される。

- ループ変数の初期設定
- ループ終了条件
- ループが終了したときに返すべき値の指定
- ループ中でのループ変数の再定義

こうすればループ変数の値は次々と変化するが，その変化は新しいループに入るときに起きる。各ループ内では単一代入規則は守られている。そして新しいループ変数か古いループ変数かを明らかにするため，VAL では iter を用い，Id や Lucid では new を用いる。VAL のループ記述の例を図 18 に示す。

```
for I, J := 1, 0;
  do if I = N
    then J
    else iter I, J := I+1, J+1;
  end
end
```

図 18: 1 から N までの和を求めるプログラム

4.2.3　履歴依存性

これまでの議論は，従来の言語では普通に記述できる問題である実時間処理—-例えば整数を入力する度にそれまでの和を順に出力するような問題—-には不向きである。

この問題は二つの方法で解決できる。一つは履歴依存性のある関数，すなわち過去の入力を憶えている関数を提供するものである。上の例では，それまでの入力の和を記憶しておき，新しいデータを入力する度にその和に加えて行けばよい。

もう一つの方法はストリームという概念を導入し，それに対して作用する関数を作るものである。ストリームは配列と同じように扱われるが，実時間でデータが供給される。和を求める問題を Id を用いて記述した例を図 19に示す。

ここで SUM と IN_DATA がストリームである。IN_DATA には次々と新しい値が入り，その値を到着した順に消費していく。SUM は計算結果を順に出力するためのストリームである。ストリームは従来型の並列処理言語におけるポートと同じ役割

---

[1] 関数の評価等において，関数値を返す以外に引数やグローバル変数の値を変更すること

```
(initial SUM ← 0
 for each T in IN_DATA do
     new SUM ← SUM + T
 return all SUM )
```

図 19: ストリームの例

をしている。但し情報交換は 1 対 1 で行われる。データ送信側はストリーム変数に次々とデータを代入し, これらの値が代入された順に受信側に渡される。データフロー言語ではストリームの概念を用いることにより一連のデータ交換を隠してしまう。

5 並列処理言語の例

並列処理言語のいくつかの具体的な例について概観する。

5.1 並列 Pascal

- プロセス

  process 内や monitor 内で process や monitor を宣言することができないため，プログラムを構造化することができない。

- 情報交換

  情報の交換は monitor を介してのみ行われる。monitor は共有変数を用いて情報交換を実現しているが，一時的にデータを送受信するだけでその後は何ら交換をしないような場合も変数を共有している。また情報交換の相手を特定することができない。

5.2 Modula

- プロセス

  プログラムを一つの module として表現する。module の役割はオブジェクトのスコープに壁を設けることであり，これによってプログラムのモジュール化が可能である。

  並列処理は process を用いて記述され，実行時に動的に起動される。しかし process 自身はレベル 0(最も浅いブロック) でしか宣言できないため，入れ子にすることは不可能であり構造化の面で問題がある。

- 情報交換

  proces 間の情報交換は interface module を介してのみ行われる。interface module は，生成が静的に行われるため固定数しか存在せず，情報交換が制限される。

5.3 CSP

- プロセス

  モジュールという概念が存在しないため，プロセス内外の間で変数のスコープに壁ができず，変数管理が難しい。また生成されるプロセスの数は固定であるので，process の数が動的に変化するような記述は不可能である。

- 情報交換

  プロセス間の相互作用を記述する手段としてメッセージ交換を用意し，非決定的なメッセージ交換を可能にしている。よってデータの一時的やり取りには適しているが，プロセス間でオブジェクトを共有するような場合の記述が困難である。

  相互作用は入出力命令を用いて記述するが，この際相手プロセス名を指定する。よって不特定プロセスを相手にするサーバプロセスを記述することはできない。

- 非決定性

  guarded command を拡張して利用できる。具体的には guard 部に論理式に加えて入力文を書くことができる。入力文が guard に含まれている場合は，論理式が真で且つ入力文の相手プロセスの準備ができていれば guard は真とみなされる。これにより必ずしも相手の応答を待つ必要がなく，可能なメッセージ交換から実行できる。

5.4 分散プロセス

- プロセス

  リアルタイム処理を目的とした言語であるから process の数が固定しており，自動的に実

行を開始する。言語を簡単にするため1つの機能でさまざまなことが記述できるようにしているが，その分エラー対処が困難である。またモジュールとしてprocessしか許されておらず，process間の入れ子は禁止されているため，プログラムの階層構造をモジュール構造に反映することができない。

- 情報交換

  情報交換は，相手process内のcommon procedureに対して外部要求を出すことによって実現する。呼び出す側は相手を指定できるのに対し，呼び出される側は指定できない。

- 非決定性

  guarded commandとguarded regionが用意されている。しかし自分の内部だけにおける非決定性であり，CSPのように他プロセスとの情報交換の可能性を，条件として書くことはできない。

5.5 Ada

- タスク

  タスクの生成/起動は動的にも静的にも行えるが，起動時に引数を渡すことができない。

- 情報交換

  情報交換のためのランデヴ(*c.f.* 3.4節)は分散プロセスの場合に比べて呼び出す側と呼び出される側の対称性は増加し，呼び出される側でも呼び出される時点を決定できるようになったが，相手を指定することはできない。

  また情報交換のもう一つの手段として共有変数が存在するが，プログラムの信頼性・保守性から考えると不要である。

- 非決定性

  選択待ち，条件つきエントリ呼出し，時限つきエントリ呼出しという3種類のselect文が提供され，情報交換も含めた対外的非決定性が記述可能である。

- 例外処理

  異常事態や例外的事態が発生したときに対応するが，タスクレベルで見ると自分の中で対処できなくなったときそのタスクは異常終了し，他のタスクに例外に関する情報が伝わらない。

6 並列論理型言語 KL1

6.1 プログラム構造

KL1は以下の形式で表される。

$$H_1 :- G_{11}, ..., G_{1m_1} | B_{11}, ..., B_{1n_1}$$
$$\vdots$$
$$H_l :- G_{l1}, ..., G_{lm_l} | B_{l1}, ..., B_{ln_l}$$

各行は節と呼ばれる。H, G, Bは述語であり，それぞれクローズヘッド，ガードゴール，ボディゴールと呼ばれる。

あるゴールHが与えられると，HとH_iとの同一化，及びガード部$G_{i1}, ..., G_{im_i}$の実行を並列に行うことができる。ここでガード部が成功したクローズのうち一つが選択され，そのボディ部$B_{k1}, ..., B_{kn_k}$が実行される。述語の同定は，述語名と引数の数とで行われる。

述語を”述語名/引数の数”で表すとする。図20にスタックプロセスのプログラムを示す。このプログラムで例えば初期ゴール`stack(L,R,[])`が与えられたとき，stack/3に同定されるのは`[1][2][3][4][5]`である。このうち例えば`[2]`を選択すると，次期ゴールは図20よりpop/3とstack/3となる。以下同様に同定が行われる。

6.2 データ構造

KL1のデータ領域は，セルと呼ばれる最小単位で管理される。データ型には単一のセルで値を直接格納するatom型やinteger型。複数のセルから成り，別のデータに対する参照ポインタを格納するfunctor型やlist型。他のセルへの参照ポインタであるreference型などがある。KL1の変数はこの参照ポインタで表現され，値に関しては特定の型を持たない。初期状態では未具体化であり値を持たないが，同一化により対象へのポインタが格納される。具体値との同一化(これを具体化と呼ぶ)が行われると具体値へのポインタが格納さ

```
[1] stack([push(D)|I],O,S)
      :- stack(I,O,[D|S]).
[2] stack([pop|I],O,S)
      :- O=[A|NO], pop(A,S,NS),
         stack(??,??,??)
[3] stack([pop(N)|I],O,S)
      :- O=[L|NO], pop(N,L,S,NS),
         stack(I,NO,NS).
[4] stack([reverse(N)|I],O,S)
      :- O=[L|NO], rev:rev(R,L),
         pop(N,R,S,NS), stack(I,NO,NS).
[5] stack([],O,_) :- O=[].
[6] pop(A,[X|S],NS) :- A=answer(X)
                       NS=S.
[7] pop(A,[],NS) :- A=empty, NS=[].
[8] pop(N,L,S,OS)
      :- N>0
         | L=[X|NL], pop(X,S,NS),
           NN:=N-1, pop(NN,NL,NS,OS).
[9] pop(0,L,S,OS) :- L=[], OS=S.
```

図 20: スタックプロセスのプログラム

れ，以後この変数を参照すると具体値を読み出すことができる。一度具体化された変数は，他の値に書き換えることはできない。

6.3 KL1 処理系 KLIC

KLIC はICOT で開発が進められている KL1 の処理系である[2]。これは KL1 プログラムを一旦 C プログラムに変換し，オブジェクトコード生成は C コンパイラに行わせる方法をとっている。よって移植性の高い処理系となっている。

参考文献

[1] 宮村 勲. 並列処理言語. マグロウヒル, 1986.

[2] 大野 和彦ほか. 静的解析による並列論理型言語の実行最適化. 1994.

[3] 大野 和彦, 伊川 雅彦ほか. 並列論理型言語 KL1 の最適化手法. 1994.

[4] 津田 孝夫ほか. 現代オペレーティングシステムの基礎. オーム社, 1988.

---

[2] 1994 年現在。