ICOT Technical Memorandum: TM-0782

TM-0782

並列プログラミング

上田和紀

July, 1989

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03) 456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

並列プログラミング

上田　和紀

新世代コンピュータ技術開発機構

1．はじめに

本チュートリアルの目的は，並列プログラミングの基本的な考え方を紹介することである．

ここで考える並列プログラミングとは，一言で言えば、MIMD型の低～中並列汎用計算機（特殊な場合として逐次計算機を含む）上での記号処理プログラミングである．このような計算機は，高性能のマイクロプロセッサの普及によって，容易に，しかも安く作れるようになってきた．そこで，その性能を活かすようなプログラミング技術を開発し，普及させることが大変重要になってきたのである．より特殊な計算機，たとえば数値計算用並列計算機やコネクション・マシン（SIMD型高並列計算機）のためのプログラミングも興味深いが，ここでは取り上げない．

また，ＡＩその他の分野で現れる並列アルゴリズムも興味深いが，各論にまで立ち入る余裕はない．並列アルゴリズムの話は並列プログラミングの話の先に位置づけられるものであり，また並列計算機の計算量のモデルに依存するところが大きいからである．ただここで，ひとつだけ注意をしておこう．それは，汎用並列計算機上の並列プログラミングは，プログラムないしはアルゴリズムの時間効率を改善することを重要な目標とはしているが，時間複雑度(time complexity) を改善することを目標としてはいないということである（なぜか？）．ＡＩにおける並列アルゴリズムの開発は，まだまだこれからの分野であり，具体的な問題をもっている研究者の協力が非常に期待されている分野である．

並列プログラミングを考える最大の動機は，並列計算機の性能を活かすところにあるが，そのために多大の苦労をしいられるのではないかと心配する人は多い．本チュートリアルの主要な目的は，アルゴリズムはともかくとして，プログラミング方法論としての並列プログラミングは別に難しいものでなく，しかも興味深くかつ重要なプログラミング・パラダイムであることを示すことである．

説明に用いる並列プログラミング言語としては，ＧＨＣ[Fuchi87][Ueda87][Ueda88a]を採用したい．ＧＨＣは，論理プログラミングにおける並列性の研究から1984年末に筆者が提案した言語であり，ICOTが開発している並列推論マシンの核言語（ＫＬ１）の基本となっている言語である．ＧＨＣを使うのは，徹底的に並列であり，また非常に単純だからである．ＧＨＣは並列論理型言語とよばれることが多いが，ここでの話に関する限り，論理とのかかわりを考える必要はない．

注目に値する並列プログラミング言語には，他にもQlisp[Goldman88]やMultilisp[Halstead85][Halstead86] などの並列Lisp，ABCL/1[Shibayama88b]などの並列オブジェクト指向言語[Yonezawa87]，Occam [Onai86] やLinda[Carriero89] などの並列手続き型言語などがあって，目移りするかもしれない．だが，これらの言語を本質的によく理解するためには，ＧＨＣのような単純で原始的な言語をまず理解しておくことは有用であろう．チュートリアルでは，これらの言語についても折にふれて言及する予定である．

なお，この原稿は，[Ueda88c] をもとに作成したものであることを記しておく．

2．パラダイムとしての並列プログラミング

パラダイムとしての並列プログラミングは，どのようなものを考えればよいのだろうか．以下に，筆者な

りの考えをまとめてみよう．

① 通常プロセスと呼ばれる，並列に実行される構成要素と，それらの間の相互作用を記述することによって，プログラムを構成するもの．かりにFortran で書いた数値計算プログラムや，Prologで書いた探索プログラムが，結果として並列計算機の上で並列に実行できるものであったとしても，それだけでは並列プログラミングをしたことにはならない．何と何を並列に動かすかを，プログラミングの上で意識し，それをプログラム上に反映させることが，プログラミング・パラダイムとしては重要である．

② “普通”のプログラムを対象にして，積極的に行なうもの．“普通の”とは，逐次プログラミングでも書ける，という程度の意味である．これまでは，並列プログラミングというと，オペレーティング・システムのように，プロセスの考え方なしにはプログラミングが非常に困難になるような場面で，必要に迫られて行なうものであった．が，そのような限定された場面でしか並列プログラミングを考えないのでは，技術としても発展しないし，並列ソフトウェアの蓄積も図れない．並列計算機の時代を迎えようとしているのだから，もっと積極的に並列プログラミングに取り組んでいかなければならない．

③ Programming in the large，つまりプログラムをその部品（プロセス）から構成する方法論を与えるもの．従来，並列プログラミングというと，興味の中心が，programming in the small，つまり言語に備わっている基本操作を用いて並列プログラムの断片を記述する方法に集まることが多かった．これももちろん重要だが，それだけ考えていたのでは，トイ・プログラミングでない，本格的なプログラミングのためのパラダイムとしての並列プログラミングが発展しない．

④ 望むらくは，並列計算機の性能（つまり並列性）を生かすための方法論を与えるもの．プログラミングにおける並列性と，そのプログラムの実行における並列性は，実は独立であって[Ueda86]，並列プログラミング言語で書いたからといって，並列計算機の上で効率よく実行できるとは限らない．本質的に並列処理に向かない問題ならば，これはやむを得ないことである．だが，並列計算機の性能を活かす方法を考えるのに，思考の道具として並列プログラミング言語を持つことは大切で，その上でよりよい並列プログラムの研究開発を進めてゆくのが本道であろう．また，それが，並列プログラミングに期待される大きな役割であることも確かである．

パラダイムとしての並列プログラミングは，オブジェクト指向[IPSJ88][Shibayama88a]と似たところを持っている．つまりこれらは，ある程度言語に独立な概念であって，プロセスなりオブジェクトなりを記述できる言語さえあれば，あとはむしろ実践の方が問題にされる．パラダイムの根幹を支える安定した理論体系の整備が待たれる，という点でも似ている．技術的にみても，並列プログラミングとオブジェクト指向は非常に近い．プロセスとは，並列に動作し，互いに通信しあうオブジェクトに他ならないからである．

3．並列プログラミングはむずかしくない

並列プログラミングを勉強するのに，一体何から始めるのがよいだろうか？

十年ほど前まで，この分野の代表的文献といえば，DijkstraのCo-operating Sequential Processes という論文[Dijkstra68]であった．メモリを共有する手続き型プロセス間の同期の問題を扱った古典的な論文である．だが，この論文は，「並列プログラミングというものはとにかくむずかしいものである」という意識を多くの人々に植えつけてしまったのではないかと思う．並列プログラミングを研究として志す人にとって，今でも一度は読むべき文献であることに変わりはないが，初心者にはもっと抽象度の高いものから与えなければ，良くて消化不良，普通は拒絶反応を起こしてしまう．

一方，そのころ出版されたHoare のCommunicating Sequential Processes（ＣＳＰ）に関する論文[Hoare78] は，Dijkstraの論文に比べるとはるかに簡明に，並列プログラミングの面白さを示したものだっ

た．ＣＳＰのわかりやすさは，次のような点によるものであろう：

① なじみやすい機能によって，並列性が実現されていたこと．簡単にいえば，普通の手続き型の逐次プログラムが外界との間で入出力をしていたのに対し，ＣＳＰでは，そのような逐次プログラムが複数個並列に走り，互いの間で入出力をするようになっただけである．
② 言語が非常に単純であること．言語にはそれぞれ使命があって，たとえばＡｄａやCommon Lisp が複雑だから良くないなどと，一概に言えるものではない．だが，手続き型言語やLisp（これももはや手続き型言語だ，という説が有力だが）のような成熟市場ならともかく，並列プログラミングのようにわけのわからない（とされている）ものが対象ならば，単純明快な言語の存在意義は大きい．

並列プログラミングへの第一歩がこれ位簡単になるならば，プログラミング教育の非常に初期から，並列プログラミングを教えることができる．Pascalなどで逐次プログラミングを学んで，逐次計算機上でのスピード感覚を身につけるのも重要だが，それだけしか勉強しないと，フォン・ノイマン計算機を前提にした思考に強くしばられてしまい，それがすべてであると思いこんでしまう危険がある．並列プログラミングを初期から学ぶことは，オブジェクト指向プログラミングへのよい入門にもなるであろう．２節でもふれたように，メッセージのやりとりによって交信するプロセスは，オブジェクトに他ならないのだから．

本稿で紹介する言語ＧＨＣ(Guarded Horn Clauses)は，いろいろな意味でＣＳＰの影響をうけて設計された並列プログラミング言語である．詳細は次節以下で明らかにするが，通信しあうプロセスを用いてプログラムを構成するという点や，単純であるという特徴は引き継ぎ，さらにそれを徹底しようとしている．

並列プログラミングへの導入言語として，ＣＳＰや，その実用版であるOccam [Onai86]も悪くない選択だと思うが，ＧＨＣからはいるのも，次のような意味で興味深い：

- ＣＳＰとちがって，各プロセスの内部にも逐次実行の概念がないという意味で，徹底的に並列である．つまりＧＨＣでは，本質的に必要でない逐次性を一切使わずに，並列プログラムが書ける．このことを知るのは重要である．
- ＣＳＰの変数には，異なる値を何回でも代入できる．つまり破壊的代入ができる．一方ＧＨＣの変数は単一代入変数であり，値を一回代入したら変更できない．この単一代入変数の採用で，プロセス間通信や同期のメカニズムが簡明になっている．
- 機能が原始的で，他言語にみられるいろいろな機能を統一的に説明する土台を与えてくれる．
- 機能が原始的ではあるが，ＧＨＣは手続き型のアセンブリ言語とちがってプロセスを用いた抽象化の機能をはじめから備えている．

４．ＧＨＣが与える並列計算の枠組

本節では，ＧＨＣという言語が提供する並列計算の枠組（あるいは並列計算モデル）を標語的に示し，それに解説を加えてゆくことにしよう．ここでは特に，“ものの見方”に重点をおいて話をすすめることにしたい．

４．１．ＧＨＣにおける計算とは，外界との情報のやりとり（通信）である

いきなり，計算とは通信である，などという聞き捨てならぬ標語から始まった．ここで計算というのは，ＧＨＣ（でなくてもよいのだが）で書いたプログラムの実行を，外界から眺めたものを指している．多くの読者は，“計算”ときくと，算術式やラムダ式の評価，あるいは代入文，goto文のような文(statement)の実行をイメージするであろう．計算というものを内側から，つまり計算機の中にはいりこんでとらえれば，

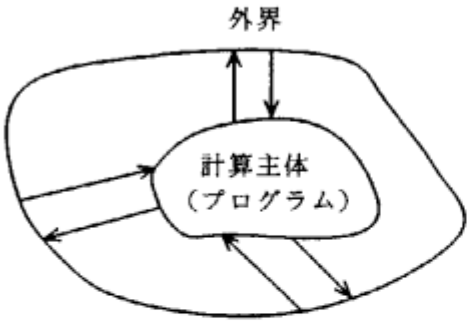

図1　"計算"のイメージ

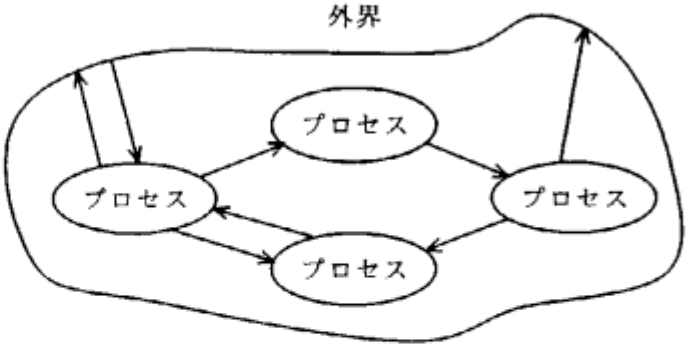

図2　並列プログラムの構成

このようなメカニズム的なものが見えてこよう．だが，そのような計算機の内側のことを捨象してしまうと，結局与えられた問題やデータを処理する主体A（プログラム）と，Aに問題やデータを与えて，計算結果を受けとる主体B（外界）との間の通信――ふつう入出力とよばれる――が残る（図1）．並列プログラミングやオブジェクト指向パラダイムの特徴は，このような情報の授受に注目する点である．

外界との通信は，それが計算の最初や最後だけでなく，計算の途中にも起こりうる．このことは，会話的なプログラムを考えるときに重要となる．

4．2．計算を行なう主体は，互いに，および外界と通信しあうプロセスの集まりである

プログラムが通信機能をもつならば，通信の相手は，別に外界でなく，並列に動いている別のプログラムであってもかまわないはずである．とすると，複数個の，並列に動作し，互いに通信しあうプログラムを全体として1個のプログラムとみなすこともできる．逆にいうと，1個のプログラムを，複数個の並列に動くプログラムを部品として構成することができる（図2）．

このような，他と並列に動き，通信機能をもつプログラムのことを，一般にプロセスと呼ぶ．なぜプロセスと呼ぶかというと，関数や手続きと異なり，実行の結果でなく，実行の過程（プロセス）における情報の授受が興味の対象となるからである．

4．3．プロセスは，停止するとは限らない

プロセスは，その実行過程，つまり外からみれば通信の履歴が主な興味の対象であり，いずれ停止するか否かということは，必ずしも重要ではない．たとえば，バンキング・システムを考えてみよう．銀行マンはともかく，現金支払機を操作する一般預金者にとっては，有限時間内にお金が出てくるかどうかが当面の関心事で，システムの停止性など，どうでもよいのである．

4．4．プロセスは，開いた系をモデル化する

これは，プロセスが外界と通信できる，ということをいいかえたものである．開いた系(open systems)というのは，閉じた系，つまり外界との相互作用なしに計算を行なう系の対立概念である．いままでに作られた多くのプログラミング言語では，外界との通信機能はad hocな形でしか与えられてこなかった．特にLispやPrologなどの（もともと）非手続き的（だったはずの）言語で，この問題は顕著であった．この意味で，開いた系の記述への適合性は，重要な特徴となりうる．

4．5．情報とは変数と値との結びつき（結合）のことである

そろそろ，ＧＨＣに特有の話やプログラム例もまじえながら，話を具体化してゆこう．

プロセスがやりとりする情報とは，一体どのようなものであろうか？

たとえば，５という自然数を考えてみる．この５が情報であるかというと，これは単なる自然数であって，情報とはいえない．だが，「Ａ君の一学期の国語の成績は５である」というのは立派な情報である．つまり，何か知りたいものがあって，その値（ここでは，知りたかったものは値として記号的に表現できるものとする）がわかるということが重要なのである．

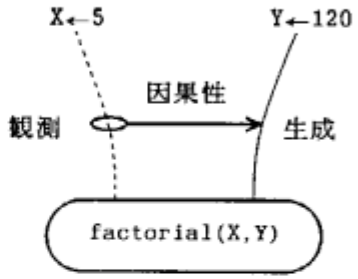

図３　プロセスfactorial(X,Y)の機能

ＧＨＣでは，何かわからないもの（未知数）を表わすのに変数を用いる．Ａ君の一学期の国語の成績を表わす変数をＫとしよう（本稿では，変数は大文字で始める）．すると，上述の情報は，Ｋと５との結びつき（結合(binding) といい，K←5 などと書く）としてモデル化することができる．変数の値に関する情報が得られることを，変数が具体化(instantiate) するという．

ＧＨＣの変数は，いったん値がきまると，その結びつきが変更されたり取り消されたりすることは未来永劫ない．変更されることがないという点で，ＧＨＣの変数は手続き型言語の変数と異なる．ではPrologのような論理型言語の変数と同じかというと，Prologの変数の値はバックトラックによって取り消されることがあるのに対し，ＧＨＣの変数はその心配がない．国語の成績が５だとわかったら，それを知ったプロセス（Ａ君）は，安心してその情報を使う（たとえばプレゼントをねだる）ことができるのである．

変数の値に関する情報は，一度にまとめて来るとは限らない．たとえば，「Ｘは100 未満の素数の列（リスト）である」という情報 X←[2,3,5,...,89,97] は，より小さな情報――たとえば，「Ｘは２を先頭とするリストである」という情報と，「残りが[3,5,...,89,97] である」という情報――に分割されてくるかもしれない．“残り”に関する情報も，同様に分割できる．ＧＨＣでは（というか論理型言語一般の特徴だが），「Ｘは２を先頭とするリストである」というような不完全情報（あるいは部分情報）を，新たな変数（X1とする）を用いて X←[2 | X1] とモデル化することができる．ここで[2 | X1] は，２を先頭要素とし，残りがX1であるようなリストを表わす．X1が，あとでX1←[3 | X2]，X2←[5 | X3] のようにきまってゆく部分である．

この例におけるＸの値のように，要素の値が頭の方から徐々にきまってゆくようなリストのことを，我々はストリームと言い習わしている．ストリームについては５．１節その他でまた詳しく述べる．

4．6．プロセスは，結合の観測と生成を行なう

Ｘの階乗の値Ｘ！を計算するプロセスを考えてみよう．ＧＨＣのプロセスが外界に結果を伝える唯一の方法は，変数と値との結合を生成することである．そこで，階乗計算のプロセスは，Ｘと，結果を返すための変数（Ｙとする）を引数として，factorial(X,Y)というような形で書くことにする．引数は，外部との通信の窓口だと考えればよい．

プロセスfactorial(X,Y)の仕事は，Ｘの値を観測し，わかったらその階乗値を計算してＹに返すことである．たとえばＸが５であることがわかったら，Ｙを120 に具体化する（図３）．factorial(X,Y)を起動したとき，Ｘの値はすでにわかっていても，まだわかっていなくてもよい．まだわかっていなければ，factorial(X,Y)は，Ｘの値がわかるまで待っている．これがＧＨＣにおける同期(synchronization) のメカ

ニズムである．

さて，プロセスというからには，階乗の値を１個計算しておわり，ではいかにも寂しい．いろいろな自然数が次々にやって来て，それらの階乗を求めることを考えてみよう．

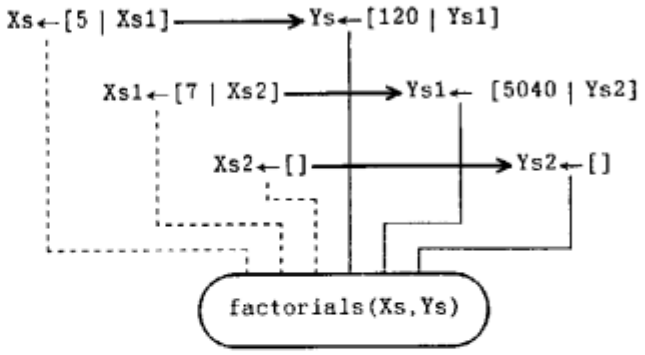

図４　プロセスfactorials(Xs,Ys) の機能

次々にやってくる自然数 n_1 ,n_2 ･･･ を表わすのに，X← n_1 ， X← n_2 といった結合を次々に作るわけにはいかない．ＧＨＣの変数の値は，変更がきかないからである．このような場合には，ストリームを利用する．自然数のストリームを観測し，それらの階乗値のストリームを生成するようなプロセスを考えるのである．このプロセスを，factorials(Xs,Ys) としよう（図４），factorials(Xs,Ys) を起動しておいて，Xs←[5 | Xs1]という結合をXsに与えると，このプロセスはYs←[120 | Ys1]という結合を生成する．次に7!を計算したくなったら，Xsではなくて，Xs1 に対して Xs1←[7 | Xs2]という結合を与える．すると，factorials(Xs,Ys) は Ys1← [5040 | Ys2]という結合を生成する．ここでもう階乗計算をやめたくなったら，Xs2←[] (□は空リスト）という結合を与えると，Ys2←[]という結合が生成して，factorials(Xs,Ys) の実行が終了する．

終わってみれば，結局factorialsはXs←[5,7] という情報に対してYs←[120,5040]という情報を返したことになるが，重要なことは，Xsの値（入力）が全部きまらないうちに，Ysの値（出力）の一部をきめることができた，という点である．この性質は，ストリームのパイプライン処理（５．３節）に活かすことができる．

４．７．プロセスは，プロセスを用いて定義する

前節のfactorial やfactorialsの定義は，ＧＨＣではどう書くのだろうか？

ＣＳＰをはじめとして，手続き型の並列言語の多くは，プロセスの内部のくり返し構造を，ループによって記述していたが，ＧＨＣではくり返しは再帰(recursion) によって実現する．つまりプロセスはプロセスを用いて定義する．たとえばfactorial の定義は次のようになる：

```
①  factorial(X,Y) :- X>0  |  X1:=X-1, factorial(X1,Y1), Y:=X*Y1.
②  factorial(0,Y) :- true |  Y=1.
```

①，②のそれぞれをガードつきホーン節(guarded Horn clause)，あるいは単に節という．このガードつきホーン節の集合によってプログラムを書くところが，ＧＨＣという名前の由来である．

個々の節は，次のように読む：

① factorial(X,Y)の形のプロセスがあって，Ｘが正整数であることがわかったら，新たに二つの補助変数（X1，Y1とする）を用意して，もとのプロセスをX1:=X-1, factorial(X1,Y1), Y:=X*Y1の三つのプロセスにおきかえてよい．

② factorial(X,Y)の形のプロセスがあって，Ｘが０であることがわかったら，それをプロセスY=1 でおきかえてよい．

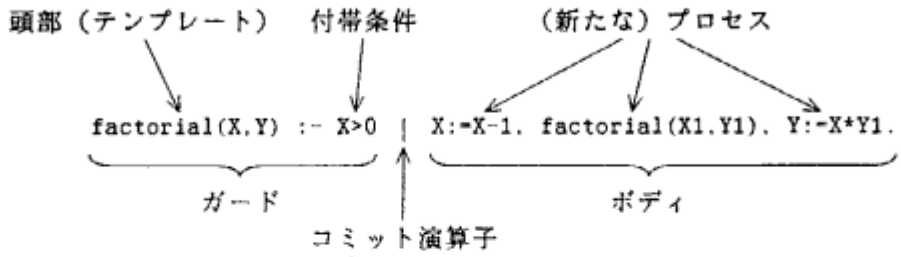

図5　節（プロセスの書換え規則）の構成

つまり，各節は，プロセスの書換え規則であり，縦棒の左側に，その規則が適用できる条件，右側に書換え後のプロセスが書いてある．縦棒の左側をガード，右側をボディと呼び，縦棒をコミット演算子と呼ぶ（図5），ガードの中の記号 ":-" の左側を頭部と呼ぶ．頭部は，どういう形のプロセスを書き換えるかを表わし，残りのガード（":-" と "|" の間）は，その書換えのための付帯条件を示す．②のtrueは，付帯条件が空であることを示す．

プロセスの書換えがおきても，もとのプロセスが概念的に消えてなくなるわけではない．むしろ，プロセスの書換えとは，もとのプロセスが，自らの任務を遂行するために，必要なサブプロセス（下請け）を生成することだと考えた方がよいであろう．プロセスが書換え規則によって定義してあるということは，そのプロセスを使う側には関係のないことである．

節の頭部と，プロセスとの間では，パターン・マッチングが行なわれる．たとえばfactorial(M,N)というプロセスは，①の頭部factorial(X,Y)とマッチし，M>0 であることがわかったら規則①を適用して，X1:=M-1, factorial(X1,Y1), N:=M*Y1におきかえてよい．一方，factorial(M,N)は，②の頭部factorial(0,Y)とは，Mの値が0であることがわかるまではマッチできない．つまり，パターン・マッチングのときは，書き換えようとするプロセスの方が，頭部よりも一般的な形をしていてはならない．

②のY=1 というプロセスは，Yを1に具体化するもので，"=" の定義は，言語に作りつけになっている．①のX1:=X-1 は，Xの値がわかるまで待ってX-1 の値を計算し，X1をその値に具体化する．":=" の定義は，概念的にはＧＨＣで定義できるので，言語に作りつけになってはいないが，処理系側で用意してある．"=" と ":=" のちがいは，":=" が右辺を算術式と思って評価するのに対し，"=" は評価しない点である．たとえばX1=5-1を実行すると，X1は4でなく，5-1 という形のもの（項と呼んでいる）に具体化する．":=" が，概念的にはＧＨＣで定義できるといったのは，

```
X:=5-1 :- true | X=4.
```

のような形の，式から値を求める規則が，多数列挙してあると思えばよい，という意味である．

次に，factorialsの定義は，次のようになる：

```
③  factorials([X | Xs1],Ys) :- true | factorial(X,Y), Ys=[Y | Ys1], factorials(Xs1,Ys1).
④  factorials([],       Ys) :- true | Ys=[].
```

③のガードは，"factorials(Xs,Ys) の形のプロセスがあって，Xsが[X | Xs1]の形，つまり長さ1以上のリストであることがわかったら" と読む．

①節や③節のボディのプロセスには順序がなく，並列に実行できる．これが，ＧＨＣの並列性の源泉である．そうは言っても，同期の規則があるから，複数のプロセスが本当に並列に実行できるとは限らない．①では，プロセスfactorial(X1,Y1)は，X1:=X-1 がX1の値をきめるまで書換えができないし，Y:=X*Y1 は，factorial(X1,Y1)がY1の値をきめるまでは掛算のしようがない．

しかし，③では事情がちがう．たとえば，プロセスfactorials(Xs,Ys) は，Xs←[150 | Xs1]を観測すると，

```
factorial(150,Y), Ys=[Y | Ys1], factorials(Xs1,Ys1)
```

の三つのプロセスにおきかわる．これらはそれぞれ，150!の計算，出力リスト構造の作成，および次の入力の処理であって，並列に処理できる．実際，Xs1 が[3 | Xs2]だとわかったら，150!の計算と並行して factorials(Xs1,Ys1) が

```
factorial(3,Y1), Ys1=[Y1 | Ys2], factorials(Xs2,Ys2)
```

に分裂し，3!が150!よりも先に求まってしまうかも知れない．

４．８．通信は，プロセス間の共有変数を用いて行なう

ＧＨＣにおけるプロセス間通信は，あるプロセスが生成した結合を，別のプロセスが観測することで成立する．したがって，通信は，両プロセスの共有変数，つまり双方からアクセスできる変数を介して行なわれることになる．

しかしながら，このＧＨＣの共有変数通信は，手続き型並列言語における大域変数を用いた共有変数通信とは全く異なる．まず，１対１や１対多の通信についていうと，ＧＨＣでは，変数の値を積極的に不定にしておくことができるので，「値がわかるまで待つ」という自然な形で同期を実現することができる．手続き型の並列プログラミングでは，共有変数通信とメッセージ通信が対立する概念として論じられることが多いが，ＧＨＣの共有変数通信は，同期と通信が一体となっているという点で，メッセージ通信でもある．実際，結合の生成はメッセージ送信，観測はメッセージ受信と解釈できる．

一方，手続き型並列言語では，多数のプロセスが変数を共有し，排他的にそれにアクセスしてその内容を書きかえる，ということができるが，ＧＨＣの変数はそういう目的には使えない．そのような共有資源を実現するには，ＧＨＣではプロセスを用いる．プロセスによる共有資源の実現については，５．２節で述べる．

ＧＨＣのプロセス間通信を，メッセージ通信としてみたときの特徴は，メッセージの列がストリームというデータ構造として明示的に存在していることである．Ada のような手続き型並列言語や，オブジェクト指向言語では，メッセージの列の概念が言語に組み込まれていて，それに対する専用の操作（メッセージ送受信等）が用意されていた．ＧＨＣでは，メッセージ列は，単なるリスト構造だと思って扱うことができる．これは，概念整理の観点から興味深いし，プログラムの性質を形式的に議論するときに有利であると期待できる．

４．９．外界も，プロセスとしてモデル化される

トイ・プログラムを別にすれば，プログラムは通常，自分で外界と通信しなければならない．表計算言語など，特定用途の問題向き言語では必ずしもそうは言えないが，汎用言語では，入出力がプログラムで制御できることが必須である．ＧＨＣは，この意味で汎用言語である．

ＧＨＣでは，入出力とプロセス間通信を，同一のメカニズムで実現できる．各処理系は，外界，具体的にいうと周辺機器をモデル化したプロセスを提供し，ユーザプロセスはそれとプロセス間通信をするのである．

例をあげよう．Prologの上に筆者が作成したＧＨＣ処理系[Fuchi87]には，outstream(X)という組込みのプロセスがある．これは出力端末をモデル化したもので，出力指令のストリームＸの中の各指令を順に処理する．出力指令は，write(hello)やnlなど，Prologでは副作用によって出力を行なうゴールであったものを用いる．たとえば，"hello"と表示して改行を行なうには，Prologではwrite(hello)とnlという"ゴール"をこの順に実行していた．ＧＨＣでは，outstream の引数Ｘを[write(hello),nl | X1]に具体化すればよい．

４．１０．通信は，非同期的である

ＧＨＣのプロセス間通信では，送信側，つまり結合を生成する側のプロセスは，その結合を受信側プロセスが観測するのを待たずに，処理をすすめることができる．つまり，ＧＨＣのプロセス間通信は非同期的である．

さらにＧＨＣでは，ABCL/1 [Shibayama88b] とちがって，送った二つの情報が，送った順番に受信側に届き，処理されるという保証も，計算モデルの上では存在しない．もっと正確にいうと，そもそもＧＨＣでは，独立の二つの情報を送るのに，時間的順序を指定することができない．たとえば，以下に定義するプロセスp(X,Y)は，X←5 という情報と Y←6 という情報を送信するが，それを行なうゴールX=5 とゴールY=6 がどのような順序で実行されるかはわからない：

```
p(X,Y) :- true | X=5, q(Y).
q(Y)   :- true | Y=6.
```

しかしながら，通信にストリームを使えば，その中では要素は順序づけられているから，受信側で順序をとりちがえることはない．つまり，複数個のメッセージを，順序を保って送りたいときは，１本のストリームを使って送ればよい：

```
p(Xs) :- true  | Xs=[5,6].
```

ABCL/1では，送信順序の保存が計算モデルの問題であるのに対し，ＧＨＣではプログラミングの問題となっているわけである．

４．１１．プロセスのふるまいは，非決定的でありうる

切符の予約システムのように，複数台の端末から入力される要求を処理するシステムを考えてみよう．話を簡単にするために，端末は２台しかないとする．１台の端末からの要求は逐次的に発生するから，その要求列はストリームで表現するのが自然である．一方，システムは，そのようにしてできる２本のストリームの中の要求に，適当な順序をつけて，一本化して処理しなければならない．

この順序づけが，両ストリームの内容だけから，一意にきめられるならば，システムは決定的であるという．だが，切符の予約システムの場合は，各要求を，その内容ではなくて"時間的順序"にしたがって処理する必要がある．この時間的順序なるものがまた曲者で，２台の端末からほとんど同時に要求が出た場合のような，微妙なケースまで考えると，客観的に"正しい"順序などというものは，ありえないことがわかる．つまり，ほぼ同時に二つの要求がきたら，どんなに公平なシステムでも，自分の裁量でどちらかを選んで先に処理せざるを得ない．これは，このシステムの挙動が，入力だけからは本質的に定められない，つまり非決定的でなければならないことを示す．このようなシステムを記述することは，ＧＨＣの重要な応用の一つであるので，ＧＨＣは非決定的処理の機能を備えている．具体的には，節，つまりプロセスの書き換え規則のなかに，適用可能なものが複数個あるとき，そのどれを適用するかを，プロセス自身の判断に委ねている．

読者の中には，上の非決定性の問題は，各要求にタイムスタンプをつける，つまり各要求に，その送信時刻をつけて送ることで解決できると思う人がいるかもしれない．たとえば，秒単位のタイムスタンプをつければ，秒単位の精度で，要求を送信順に処理できそうにも思える．しかしそうは問屋がおろさない．ＧＨＣでは，２台の端末（をモデル化したプロセス）とシステム（をモデル化したプロセス）の間の通信には遅れが伴ってもよいことになっている．すると，11時22分33秒のタイムスタンプをもつ要求ｒが端末Ａから来ても，ただちにそれを処理するわけにはいかないのである．端末Ｂからの要求を５秒待ってみて，11時22分32秒以前の要求がこなければ，ｒを処理する，というたぐいの解決法は，抽象計算モデルの世界では，一般性も説得力ももたない．

5．もう少し具体的なパラダイム

前節では，ＧＨＣの提供する並列プログラミングの枠組について，計算モデルの基礎に重点をおいて解説した．本節では，そのような基本メカニズムを用いて，具体的にどのようなプログラミング上の概念が実現できるかを示す．

5．1．ストリームと双方向通信

前節でもふれたが，ストリームとは頭の方から徐々に生成され，消費されるリスト構造のことである．ＧＨＣでは，ストリーム型のようなものが，言語に作りつけになっているわけではない．ＧＨＣのストリームは，静的には単なるリストであり，上述のような作り方，使い方をするリストを特にストリームと呼び習わしているにすぎない．

ストリームの概念は，Miranda [Kato88]，ＫＲＣ[Fuchi86] をはじめとする関数型言語にもみられる．関数型言語の場合，ストリームを頭の方から生成するためには，ふつう遅延評価というメカニズムを用いる．一方ＧＨＣでは，単一代入変数を利用して，リストの残りをとりあえず不定にして先頭の要素を先にきめることでストリームを実現している．

ストリームに関して重要なことは，生成プロセスが止まらない場合は，極限において無限長のリストができるということである．このことの影響が，一つにはプログラムの理論的扱いに現れる．一般に，プログラム（の断片）の性質についての議論は，停止性と，止まったときの結果の２段構えで行なうことが多かった．だが並列プログラムの場合は，止まらないプログラムから得られる結果も議論できなければならず（４．３節参照），そのような議論においては，ストリームの長さに関する帰納法が一般には使えないことになる（ためしに，二つのプログラムの生成する無限ストリームが同じであるかどうかを論ずるにはどうしたらよいかを考えてみてほしい）．これに対する一つの代案は，ストリームの観測精度に関する帰納法を使う方法である．つまり，二つの無限ストリームが同じであるということを，“頭から要素を比較していったとき，有限回の比較では違いが発見されないこと”と定義し，比較の回数に関する帰納法を用いるのである．このような論法で，止まらないプログラムに関して現在考えられているさまざまな性質をすべて扱えるわけではないが，多くの“素直”な性質は，この方法で扱うことができる．

４．１０節に述べたように，ストリームによるプロセス間通信は，完全な非同期通信である．ストリームの生産者（送信プロセス）は，消費者（受信プロセス）に無関係に要素の値をきめてゆくことができ，消費者もまた，生産者を追い抜かないという制限の下で，マイペースでその値を使っていってよい．つまり，ストリームは非有界バッファ(unbounded buffer)の役割を果たしている．

ＧＨＣのストリームの重要な特徴は，生産者から消費者への単方向通信に使えるばかりでなく，返信を伴うような双方向通信も一本のストリームでできる点である．つまり，メッセージの送信者が“返事”がほしい場合，送信メッセージに変数を含めておき，受信者にその変数を具体化してもらうのである．

例をあげよう．端末のキーボードからデータを入力したい場合，前述のProlog上のＧＨＣ処理系では，

instream(X) という組込プロセスを利用する．Xを[read(T)|X1] に具体化すると，instreamは端末からデータを１個読み込み，その値をTと結合する．read(T) のように，変数を含むメッセージを，未完成メッセージ(incomplete message)という．未完成メッセージに対して返答を出し，メッセージを"完成"させるのは，受信側の役割である．

instream(X) には，実は出力メッセージを送ることもできる．たとえば，Xを[write('more?'),nl,read(T) |X1] に具体化すれば，"more? " というプロンプトが出て，改行したあとで読込みがおきる．instreamにwrite('more?')のような出力指令を送れるようにしたのは，出力指令と入力指令を，同じストリームの中に並べて順序づけることにより，ディスプレイにプロンプトを出すタイミングと，データを読み込むタイミングの前後関係を保証できるようにするためである．４．９節で説明したoutstream が，出力端末をモデル化したものであったのに対し，instreamは，入出力端末をモデル化したものであるといえる．

未完成メッセージは，プロセス間の双方向通信の記述をきわめて容易にする．双方向通信が容易に書けることは，オブジェクト指向のプログラミングや会話的プログラムの記述に大変重要である．双方向通信は，もちろん単方向通信のストリームを２本使ってもできるが，プログラムが繁雑になるのは否めない．

５．２．履歴のあるオブジェクトの表現

未完成メッセージを用いると，履歴をもつオブジェクトを，容易に記述できる．オブジェクトをプロセスとして表現し，そのプロセスにつながるストリームに，指令の列を与えるのである．たとえば，スタックをプロセスとして定義すると，次のようになる：

```
stack(Xs) :- true | stk(Xs,[]).
stk([push(T)|Xs1],Ls      ) :- true | stk(Xs1,[T |Ls]).
stk([pop(T) |Xs1],[L |Ls1]) :- true | T=L, stk(Xs1,Ls1).
stk([pop(T) |Xs1],[]      ) :- true | T=error, stk(Xs1,[]).
stk([],           Ls      ) :- true | true.
```

引数Xsは，スタックに対する指令列を表わすストリームである．実際にスタックを使う場合は，stack(Xs) というプロセスを起動する．起動しておいて，Xs←[push(5),push(4),pop(X),pop(Y),pop(Z)]という結合を与えると，X=4，Y=5，Z=error という答が返ってきて，スタック・プロセスは終了する．stack を異なる引数で２回起動すれば，二つの独立なスタック・プロセスができる．

一般に，関数型言語や論理型言語など，変数への破壊的代入を許さない言語で，履歴（状態）をもつオブジェクトを表現することは困難だと考えられているが，ＧＨＣでは，書換え規則で定義したプロセスと双方向通信を用いることによって，その問題を解決している．

上のスタックを，複数のプロセスが排他的にアクセスする共有資源とするときは，それらのプロセスからの指令列を併合(merge) して，一本化しなければならない．ストリームの併合は，基本的には，次のように定義される併合プロセスを用いて行なう：

```
merge([A|Xs1],Ys,     Zs) :- true | Zs=[A|Zs1], merge(Xs1,Ys,Zs1).
merge(Xs,     [A|Ys1],Zs) :- true | Zs=[A|Zs1], merge(Xs,Ys1,Zs1).
merge([],     Ys,     Zs) :- true | Zs=Ys.
merge(Xs,     [],     Zs) :- true | Zs=Xs.
```

併合プロセスを用いて，指令列を一本化するのは，ＧＨＣにおける多対１の通信の基本テクニックである．

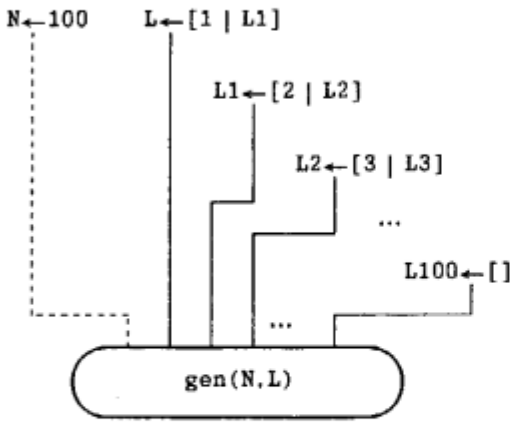

図6　自然数列の生成（データ駆動版）

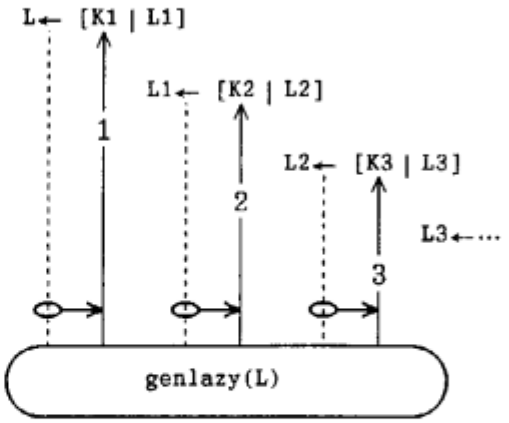

図7　自然数列の生成（要求駆動版）

ABCL/1をはじめとして，多対1のメッセージ通信を許す多くの言語では，指令列の一本化はimplicitであるが，ＧＨＣは原始的であるため，そのような一本化操作が介在している以上，そのことをexplicitに書かなければならない．これも一つの見識だと思うのだが，いかがであろうか．

そうはいっても，多対1の通信を多用するアプリケーションでは，併合プロセスをいちいち書くのは負担であろう．また，その処理効率も重要である．そこで，ストリームの併合については，プログラミングの経済性と実行効率の両面から，研究がすすめられている．

5．3．データ駆動計算と要求駆動計算

データ駆動計算(data-driven computation) とは，関数の値やプロセスの出力などを計算しようとするときに，その計算に必要なデータが揃っていれば，出力を他の関数やプロセスが必要としているかどうかにかかわらずに計算をすすめる計算法をいう．

一方，要求駆動計算(demand-driven computation) とは，関数の値やプロセスの出力などを積極的に計算せず，それらを利用する関数やプロセスから要求が来て初めて計算する計算法をいう．

関数型言語では，この2種（もっと細かく分類することもあるが）の計算法の違いは，引数の評価のメカニズム――call by value，call by need 等――の違いとして，計算モデル，つまり言語の操作的意味論の中で論じられてきた．これに対してＧＨＣでは，データ駆動と要求駆動の違いは，同じ（ＧＨＣの）計算モデルの下での，プログラミング技法の違いとして実現している．

例をあげよう．自然数の列[1,2,....,N] を積極的に生成するプロセスの定義は，次のように書ける：

```
gen(N,L) :- true | g(0,N,L).
g(I,N,L) :- I<N  | I1:=I+1, L=[I1 | L1], g(I1,N,L1).
g(N,N,L) :- true | L=[].
```

プロセスgen(N,L)は，Ｎの値さえわかれば，ストリームＬを自律的に生成してゆく（図6）．これに対して，次の定義を考えてみよう：

```
primes(Max,Ps) :- true | gen(1,Max,Ns), sift(Ns,Ps).

gen(N,Max,Ns) :- N< Max | N1:=N+1, Ns=[N1|Ns1], gen(N1,Max,Ns1).
gen(N,Max,Ns) :- N>=Max | Ns=[].

sift([P|Xs],Zs) :- true | Zs=[P|Zs1], filter(P,Xs,Ys), sift(Ys,Zs1).
sift([],    Zs) :- true | Zs=[].

filter(P,[X|Xs],Ys) :- X mod P=:=0 |             filter(P,Xs,Ys).
filter(P,[X|Xs],Ys) :- X mod P=\=0 | Ys=[X|Ys1], filter(P,Xs,Ys1).
filter(P,[],    Ys) :- true        | Ys=[].
```

図８　素数列生成プログラム（データ駆動版）

```
primes(Ps) :- true | gen(1,Ns), sift(Ns,Ps).

gen(M,[N|Ns1]) :- true | M1:=M+1, N=M1, gen(M1,Ns1).
gen(M,[]     ) :- true | true.

sift(Xs,[Z|Zs1]) :- true |
        Xs=[Z|Xs1], filter(Z,Xs1,Ys), sift(Ys,Zs1).
sift(Xs,[]     ) :- true | Xs=[].

filter(P,Xs,[Y|Ys1]) :- true | Xs=[X|Xs1], filter2(P,X,Xs1,Y,Ys1).
filter(P,Xs,[]     ) :- true | Xs=[].
filter2(P,X,Xs1,Y,Ys1) :- X mod P=:=0 |      filter(P,Xs1,[Y|Ys1]).
filter2(P,X,Xs1,Y,Ys1) :- X mod P=\=0 | Y=X, filter(P,Xs1,Ys1    ).
```

図９　素数列生成プログラム（要求駆動版）

```
genlazy(L) :- true | gl(0,L).
gl(I,[J |L1]) :- true | I1:=I+1, J=I1, gl(I1,L1).
```

プロセスgenlazy(L)は，Ｌの値を自律的に生成できない．genlazy(L)は，ストリームＬに関して消費者になっているのである．ではどうやってgenlazy(L)を駆動するかというと，Ｌを外部から，相異なる変数のストリーム[K1,K2,K3,... に具体化してやる．すると，genlazy はK1←1，K2←2，K3←3，... という結合を生成するのである（図７）．

お気づきかもしれないが，gen ではストリームを単方向通信に使っていたのに対し，genlazy では双方向通信に使っている．genlazy への送信データK1，K2，... は，未完成メッセージの特殊な場合で，genlazy に対して，次の値を計算するように要求していると考えることができる．ＧＨＣの要求駆動計算では，計算結果を要求するプロセスがストリームの骨格（リスト構造）を作成し，計算をするプロセスがその中味を埋めてゆく．１本のストリームが，二つのプロセスの協同作業で作られるわけである．

さて，上の自然数列生成プロセスを用いた例として，素数列を生成するプログラムを示そう．まずデータ駆動版は，図８のようになる．プロセスprimes(Max,Ps)は，直ちに二つのプロセス，つまり自然数列生成プロセスgen(1,Max,Ns) と，素数以外をふるい落とすプロセスsift(Ns,Ps) に分裂する．後者は，素数Ｐから始まり，Ｐ未満の素数の倍数がすでにふるい落とされている数列Nsから，素数列Psを作りだすものであり，素数が１個みつかるごとに，その倍数をふるい落とすfilterプロセスを生成している．filterプロセスはストリームによって一列につながっていて，パイプライン処理によって，gen の生成する自然数列から，各素数の倍数を並列にふるい落としてゆく．filterの定義の中で，"X mod P=:=0 " は，"ＸがＰで割り切れる

```
test :- true | instream(IOs), ask(IOs,Ps), primes(Ps).

ask(IOs,Ps) :- true | IOs=[read(M)|IOs1], ask2(IOs1,Ps,M).
ask2(IOs1,Ps,M) :- M > 0 | Ps=[P|Ps1], ask3(IOs1,Ps1,M,P).
ask2(IOs1,Ps,M) :- M=:=0 | ask(IOs1,Ps).
ask2(IOs1,Ps,M) :- M < 0 | IOs1=[], Ps=[].
ask3(IOs1,Ps1,M,P) :- true |
        IOs1=[write(P),nl|IOs2], M1:=M-1, ask2(IOs2,Ps1,M1).
```

図１０　素数列生成プログラム（要求駆動版）の駆動プログラム

```
| ?- ghc test.
|: 5.
2
3
5
7
11
|: 0.
|: 7.
13
17
19
23
29
31
37
|: -1.
500 msec.
```

図１１　素数列生成プログラム（要求駆動版）の実行

ことがわかる”，“X mod P =\=0” は，“XがPで割り切れないことがわかる”，とそれぞれ読む．

次に，要求駆動版は図９のようになる．プログラムの基本的構造はデータ駆動版と同じだが，gen，sift，filter の各定義で，ストリームの骨格の流れる向きが逆になっている．たとえば，filter(P,Xs,Ys) というプロセスがあったとすると，Ysから，“Xsの要素でPの倍数でないものが欲しい”という要求がくる．すると，filter(P,Xs,Ys) は，ストリームXsの“次”の値を，Pの倍数でないものがみつかるまで，前段のプロセスに要求しつづける．

図１０は，この要求駆動版のプロセスprimes(Ps)を駆動するプログラムの一例である．トップレベルのプロセスtestを呼ぶと，すぐに三つのプロセスができるが，主導権をもっているのはプロセスask(IOs,Ps) である．ask(IOs,Ps) は，端末から整数Mを読み込み，M≧０ならば次のM個の素数を要求，表示し，また次の個数を端末に聞きにいく．M＜０ならば，IOs とPsを閉じることによりinstream(IOs) とprimes(Ps)を終了させ，自らも終了する．図１１に実行例を示す．

この素数生成プログラムの二つの版を，Miranda で書いたもの[Kato88]と比べてみると，ＧＨＣで書いたものの方がかなり長い．これは，ＧＨＣの提供している機能の方が，より原始的だからである．たとえばＧＨＣ版では，要求駆動計算の要求も，プログラム上でexplicitに扱っている．Miranda の素数プログラムは疑いもなく高級言語のプログラムであるが，ＧＨＣの素数プログラムは，それに比べればかなりアセンブラ的であるともいえる．

なお，上述の要求駆動計算と類似の技法を用いると，有界バッファ(bounded buffer)通信もＧＨＣで記述できる[Fuchi87] ことを付記しておこう．

５．４．モジュラリティと差分プログラミング

プログラムとストリームを用いたプログラミングを，programming in the largeの観点から少し眺めてみよう．

前節の素数生成プログラム（データ駆動版）を修正して，双子素数（隣りあう奇数の対で，そのどちらもが素数であるもの）の列を生成するプログラムを作ることになったとする．このとき，primesの定義には全く手を入れなくてよい．素数列を入力として，双子素数列を出力とするプロセスを外付けすればよいのである（図１２）．

このように，少し異なる結果がほしい場合は，その差に相当する変換作業をプロセスとして定義して外付けすればよい．これが，プロセス指向の並列プログラミングにおける差分プログラミング(differential programming)の方法である．差分プログラミングができるということは，プログラムのモジュラリティ，汎用性，再利用性にとって重要である．

```
twins(Max,Ts) :- true | primes(Max,Ps), twin(Ps,Ts).

twin([P1,P2|Ps2],Ts) :- P2=\=P1+2 | twin([P2|Ps2],Ts).
twin([P1,P2|Ps2],Ts) :- P2=:=P1+2 | Ts=[(P1,P2)|Ts1],twin([P2|Ps2],Ts1).
twin([_],        Ts) :- true      | Ts=[].
twin([],         Ts) :- true      | Ts=[].
```

図１２　双子素数生成プログラム

```
emptyset([has(N,R)|Is])   :- true | R=false,           emptyset(Is).
emptyset([insert(N)|Is]) :- true | element(Is,Os,N), emptyset(Os).
emptyset([])              :- true | true.

element([has(M,R)|Is],Os,N)   :- M < N | R=false,             element(Is,Os, N).
element([has(M,R)|Is],Os,N)   :- M=:=N | R=true,              element(Is,Os, N).
element([has(M,R)|Is],Os,N)   :- M > N | Os=[has(M,R)|Os1],   element(Is,Os1,N).
element([insert(M)|Is],Os,N) :- M < N | Os=[insert(N)|Os1], element(Is,Os1,M).
element([insert(M)|Is],Os,N) :- M=:=N |                      element(Is,Os, N).
element([insert(M)|Is],Os,N) :- M > N | Os=[insert(M)|Os1], element(Is,Os1,N).
element([],            Os,N) :- true  | Os=[].
```

図１３　プロセスによる集合の表現

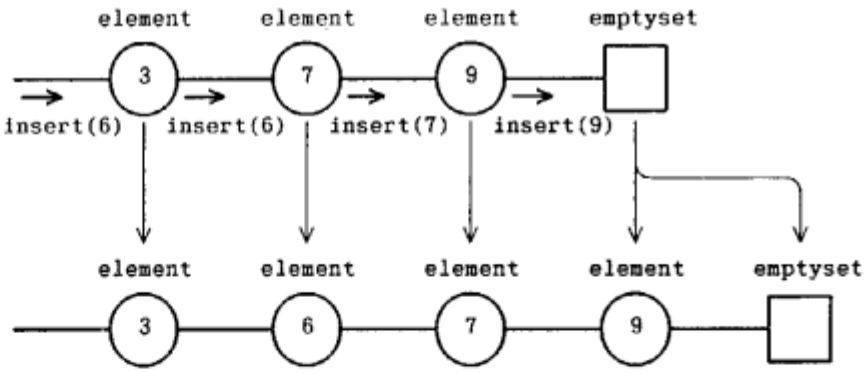

図１４　集合への要素の追加

ＧＨＣでプログラムを書くときは，その内容をなるべく多くの，意味のある部品（プロセス）に分けて定義し，ストリームを用いたネットワーキングによって全体を組みあげると，プログラムがわかりやすくなる小さなプロセスを作りすぎるとプロセス間通信がふえ効率が低下すると心配する人もあろうが，それは処理系作成技法やプログラム変換技法で解決すべき性質の問題であり，プログラミング・スタイルを犠牲にして解決すべきではない．複数個の小さなプロセスをプログラム変換で融合する技法は，[Ueda88b] を参照してほしい．

５．５．プロセスによるデータ表現

プロセスは他と並列に動作し，他と通信し，また履歴依存性（状態）を表現することができる．すると，比較的小容量のデータをもつ多数個のプロセスをストリームによって構造化すれば，並列アクセスの可能性をもった，大きなデータ構造ないしはデータベースを構築できる可能性がある．

たとえば，ＣＳＰの論文[Hoare78] には，整数の集合を，プロセスの配列によって表現する例が出ている．このプログラムをＧＨＣに翻案してみよう（図１３）．

このプログラムでは，集合は，図１４のように，プロセスelement とプロセスemptysetからなる一次元的なプロセス構造で表現する．プロセスelement(Is,Os,N)は，ちょうど１個の整数Ｎを管理しており，Isは前段の，Osは後段のプロセスにつながっている．これらのプロセスは，Ｎの値の小さい順に整列してあり，そのうしろに末端のプロセスemptysetがつながる（図１４）．この集合プロセスを利用するときは，最初にemptyset(Is)を起動し，Isにinsertやhas メッセージの列を送ればよい．

各プロセスは，メッセージinsert(N) とhas(N)に対して適切な行動をとる．集合の要素数がふえる場合の処理，つまり新たなelement プロセスの作成は，emptysetの第２節で行なっている．element の第４節は，"自分が持っている整数Ｎより小さな整数Ｍの挿入要求がきたとき，Ｎを手放して次のプロセスに送り，自分は新たにＭを持つ" という規則を表わしている．

この例題は，たかが集合を管理するのにプロセスとストリーム通信を多用しており，とても実用的には見えないかも知れない．だが，そう言って一笑に付してしまうのは早い．この技法には，次のような重要な示唆が含まれている：

① データ構造は，プログラムと同様，プロセス網によって構成できること
② プロセス網として表現されたデータ構造は，その検索，修正に並列性を活かすことができる．上の集合の例では，各メッセージがパイプライン的に処理できる．

この技法を現実のものにするには，プロセスとストリームを非常に効率的に実現しなければならないが，それに成功すれば，この技法もパラダイムとして確立してゆくと期待される[Morita89]．

６．歴史的背景

前節までに紹介した考え方や技法は，もちろん完全にＧＨＣ独自のものというわけではない．多くは，ＧＨＣに至るまでの並列プログラミング研究の成果を反映したものである．本節では，それらの成果のうち，ＧＨＣに深く影響を与えたものを紹介しつつ，歴史をふり返ってみよう．

冒頭で，ＧＨＣは，論理プログラミングにおける並列性の研究から生まれた言語であると述べた．論理プログラミングは，Kowalskiの論文[Kowalski74]によって，パラダイムとして呈示され，その後めざましく発展した分野である．この論文はすでに，論理プログラムの並列実行の可能性を示唆しており，実際その可能性が，ＧＨＣにおける徹底した並列性の源になっている．またＧＨＣは，同一化(unification) や融合原理(resolution principle)[Robinson65]など，基本計算メカニズムの多くを論理プログラミングからうけついでいる．

一方，互いに通信しあうプロセスを部品として並列プログラムを作るという考え方は，KahnとMacQueenの論文[Kahn77]およびHoare の論文[Hoare78] で，言語とともに示された．KahnとMacQueenの言語は，POP-2をベースに作られたものである．非同期通信とプロセスの動的生成を許しているが，プロセスのふるまいは（意図的に）決定的である．一方Hoare の言語は，Dijkstraのguarded commands [Dijkstra75] をベースにしている．こちらはプロセスの動的生成，再帰呼出し，非同期通信のような，動的メモリ管理を要する機能を排しているが，プロセスのふるまいは非決定的でありうる．以上の三つの文献は，いずれもきわめて有名なもので，10年以上たった今でも味わい深く，教えられることが多い．

論理プログラミングの上でプロセス指向のプログラミングを考察した研究としては，van Emden とde Lucena Filhoのもの[Emden82] や，Clark とGregory のRelational Language [Clark81] が初期のものである．前者は，KahnとMacQueenの計算モデルが論理プログラミングの上で実現できることを示し，論理プログラムにプロセス的解釈を与えた．一方Relational Language は，KahnとMacQueenが示した非同期通信や動的プロセス網の機能と，ＣＳＰのような非決定性とを合わせ持つ言語を具体的に示したもので，その後に設計

されたConcurrent Prolog，PARLOG，ＧＨＣの基本的枠組を与えたという意味で重要である．

Shapiro の設計したConcurrent Prolog [Shapiro86][Shapiro87]は，read-only annotationという同期メカニズムを用いて，未完成メッセージによる双方向通信の手法を開発し，数多くのプログラム例でその有効性を示した．履歴依存オブジェクトが，自然に表現できることを示した意義は大きい．一方Relational Languageを設計したClark とGregory は，その後継言語PARLOG [Clark86][Gregory87] で，未完成メッセージの技法が，入出力モードという，より簡単な同期機構をもった言語で実現できることを示した．ＧＨＣは，ICOTが開発する並列推論マシンの核言語を，Concurrent Prolog をたたき台として検討する過程で生まれたもので，同期メカニズムをはじめとする諸概念を整理して言語を単純化したことを最大の特徴とする．枚田のＯｃ[Hirata87]は，ＧＨＣをさらに単純化したもので，本稿に示した各技法は，実はＯｃにも適用できる．FosterとTaylorは，これらの言語の研究成果を手際よくとりいれた商用ベースの実用言語Strand (STReam AND parallelism の略）を設計した[Foster89]．

並列Lispの研究は，並列論理プログラミングの研究とほとんど独立にすすめられてきたように見える．だが，それにもかかわらず，両者で発明された機構には似たものもある．たとえばMultilisp [Halstead85] のfutureというコンストラクトは，Lispに並列評価メカニズムと“値がわかるまで待つ”という形の同期メカニズムを導入したものとみることができる．

このほかにも，ふれておくべき関連研究は多いが，それについては上に掲げた文献や，論理プログラミングや並列プログラミングに関係する会議論文集，論文誌を手がかりにしてあたってほしい．なお，並列プログラミングを扱った教科書としては，網羅的な[Filman86]や，理論的な[Chandy88]などがある．

7．並列プログラミングと効率

せっかく並列プログラムを書くからには，書いたプログラムが並列計算機の上で効率よく走ってほしいものである．

ＧＨＣの計算は，プロセスの書き換えによってすすむ．これを並列に実行する最も普通の方法は，異なるプロセスの書き換えを，別々のプロセッサで同時にすすめる方法である（本稿は，汎用マルチプロセッサシステムによる並列処理を想像しながら書いている）．しかし，ＧＨＣのプロセスは互いに通信しあうから，その手間も考えなければならない．二つのプロセスを，一つのプロセッサの中で擬似並列処理すれば，物理的並列性がないかわりに，プロセス間通信にかかる手間も小さい．一方，それらを，二つのプロセッサで並列実行させようとすると，両者が独立ならば計算時間が短縮できるが，プロセス間通信は，擬似並列処理の場合よりはるかに高くつく．さらに，その二つの仕事が十分大きくないと，複数のプロセッサに仕事を分散させる手間（これも通信の手間のうちである）に見合わない．

通信の手間には，latency （おくれ）とthroughput（単位時間あたりの転送量）とがあるが，共有メモリによらないプロセッサ間通信の場合，特にlatency の問題が深刻である．つまり，小さなデータを送ることは，一般に非常に不利である．

プログラムがどれ位並列実行に適するかは，以下のように定性的に議論できる．まずプロセス間の関係を，次の三つに分類する：

① 無通信，または疎な通信：　たとえば，クイックソートで，“大きなデータ”の列と“小さなデータ”の列をそれぞれソートするプロセス間の関係
② 密な単方向通信：　たとえば，データ駆動型の素数生成プログラム（５．３節）における，filterプロセスどうしの関係
③ 密な双方向通信：　たとえば，要求駆動型の素数生成プログラムにおける，filterプロセスどうしの関係

①→②→③の順に，プロセス間の関係は密になり，並列実行は困難になる．②では，通信データをまとめて送れるので，プロセッサ間通信を，throughputを重点に設計すれば，並列実行は有効である．しかし，③では，通信データが小刻みになり，また複数のプロセスが互いに相手の情報を待ち合わせることになるので，並列処理はむしろ有害ですらある．

結局，プログラムが並列処理に向くかどうかは，それが局所性の高く，しかも十分に仕事量のあるプロセスに分割できるかにかかっている．並列プログラム，並列アルゴリズムを設計するときは，このことに十分注意しなければならない．また，高い並列度が得られたとしても，そのために計算複雑度において逐次アルゴリズムに劣る並列アルゴリズムを採用したのでは，大きな問題を並列計算機で解く意味がなくなってしまう，ということにも注意する必要がある．つまり，並列度の大小は、それだけではプログラムのよさを評価する基準になりえない．

効率の観点からもうひとつ重要なことは，投機的計算(speculative computation)[Halstead86] の考え方をうまく活用することである．一般に，無駄な計算をせずに処理系の並列性を活かすことができれば，それに越したことはない．だが，並列性を活かすには，「必要最小限の計算しかしない」という態度を改めなければならないことがよくある．要求駆動型の素数生成プログラムもこれに該当する（ではどういう解決法がありうるか？）．ＡＩにおける探索問題においても同じことがいえる．たとえば，アルファ・ベータ探索を行なうのに，探索木の枝どうしが頻繁に評価関数の交信をすれば，無駄な探索は少なくなるが，通信量がふえ，また並列性が損なわれることになる．逆に，評価関数の交信をたまにしか行なわないと，無駄な計算をしていることに長い間気付かない，ということが起きうる．投機的計算とは，役に立つことを期待して行なうが，無駄になる可能性もある計算のことである．この概念は，従来の逐次計算機においても低いレベルの処理ではしばしば利用されてきた（実例を考えてみよ）が，通常のプログラミングで意識されることはそれほど多くなかった．だが，並列処理，とくにＡＩにおける並列処理では，これをうまく導入し，かつうまく制御することが効率上大変重要となるであろう．投機的計算は，その活用法のみならず言語設計の点からも，今後の研究にかかる期待が大きい．

8．おわりに

並列プログラミングの基本について，ＧＨＣを用いて解説してきた．本稿で示した並列プログラミングの考え方は，“ＧＨＣ的なものの見方”をかなり反映したものだが，もとよりこれが並列プログラミングのすべてであるというわけではない．興味のある方は，他の言語を用いた並列プログラミングとの比較検討を試みてほしい．

並列プログラミングに親しむためには，プログラムを実際に走らせてみるのが一番よい．最も手軽なのは，前述のProlog上の処理系で，[Fuchi87] に，ソースプログラムを含めた詳細が公開されている．最新版(DEC-10 Prolog，C-Prolog，Quintus Prolog，SICStus Prologで動く）の入手については，筆者（〒108 港区三田１丁目4-28，三田国際ビル21階，新世代コンピュータ技術開発機構，(03)456-2514）まで問い合わせていただきたい．より本格的な処理系としては，Ｃで書いたPDSS (PIMOS Development Support System) とよばれる処理系をICOTで開発した[Hirano89]，UNIX 4.2BSD （および4.3BSD）上で動く．こちらも，非営利目的に限って頒布している．ICOT（ (03)456-3191 ，担当：天野）まで御連絡いただきたい．

比較的小さなプロセスを構成要素としてプログラムを作るという考え方は，手続き型プログラミングの世界でも，lightweight process あるいはスレッド(thread)という名前と共に普及しつつある．ＧＨＣは，このlightweight process の考え方を徹底させたものといってもよい．ただし，徹底させた分だけ，プロセスのインプリメンテーションも徹底的に効率化する必要がある．ICOTでは，このプロセスを構成要素として，知識ベースを構築することも検討している．

並列プログラミングを発展させるためには，プログラミング・パラダイムや言語の開発以外にも，しなければならないことがたくさんある．使える並列処理系の開発，ちゃんとしたプログラミング環境やデバッグ環境の整備，役に立つ理論の構築など，重要なものが目白押しである．また，プログラミング・パラダイムも，"このような問題にはこのような技法を使う"というレベルまで具体化してゆく必要がある．多くの人に，この新しく大きな分野に取り組んでいただきたいと思う．

参考文献

[Carriero89] Carriero, N. and Gelernter, D., Linda in Context. Comm. ACM, Vol.32, No.4 (1989), pp.444-458.

[Chandy88] Chandy, K.M. and Misra, J., Parallel Program Design. Addison-Wesley, 1988.

[Clark81] Clark, K.L. and Gregory, S., A Relational Language for Parallel Programming. In Proc. ACM Conf. on Functional Programming Languages and Computer Architecture, ACM, 1981, pp.171-178.

[Clark86] Clark, K.L. and Gregory, S., PARLOG: Parallel Programming in Logic. ACM Trans. Prog. Lang. Syst., Vol.8, No.1 (1986), pp.1-49.

[Dijkstra68] Dijkstra, E.W., Co-operating Sequential Processes. In Programming Languages, Genuys, F. (ed.), Academic Press, 1968, pp.43-112.

[Dijkstra75] Dijkstra, E.W., Guarded Commands, Nondeterminacy and Formal Derivation of Programs. Comm. ACM, Vol.18, No.8 (1975), pp.453-457.

[Emden82] Emden, M.H.van and Lucena Filho, G.J.de, Predicate Logic as a Language for Parallel Programming. In Logic Programming, Clark, K.L. and Tarnlund, S.-A.(eds.), Academic Press, 1982, pp.189-198.

[Filman86] Filman, R.E. and Friedman, D.P., 雨宮，尾内，高橋訳，協調型計算システム：分散型ソフトウェアの技法と道具立て，マグロウヒルブック，1986.

[Foster89] Foster, I. and Taylor, S., Strand: a Practical Parallel Programming Language. To be presented at the 1989 North American Conf. on Logic Programming, Cleveland, 1989.

[Fuchi86] 淵一博，黒川利明編著，新世代プログラミング，共立出版，1986.

[Fuchi87] 淵一博監修，古川康一，溝口文雄共編，並列論理型言語ＧＨＣとその応用，知識情報処理シリーズ第６巻，共立出版，1987.

[Goldman88] Goldman, R. and Gabriel, R.P., Qlisp: Experience and New Directions. In Proc. ACM/SIGPLAN PPEARLS, Sigplan Notices, Vol.23, No.9 (1988), pp.111-123.

[Gregory87] Gregory, S., Parallel Logic Programming in PARLOG. Addison-Wesley, 1987.

[Halstead85] Halstead, R.H.Jr., Multilisp: A Language for Concurrent Symbolic Computation. ACM Trans. Prog. Lang. Syst., Vol.7, No.4 (1985), pp.501-538.

[Halstead86] Halstead, R.H.Jr., Parallel Symbolic Computing. IEEE Computer, Vol.19, No.8 (1986), pp.35-43.

[Hirano89] 平野喜芳，中越靖行，西崎慎一郎，宮崎芳枝，宮崎敏彦，近山隆，汎用計算機上のＫＬ１処理系－－ＰＤＳＳ－－，In Proc. Logic Programming Conference '89, ICOT, 1989.

[Hirata87] 枚田正宏，並列記号処理言語Ｏｃとその自己記述，コンピュータソフトウェア，Vol.4, No.3 (1987), pp.41-64.

[Hoare78] Hoare, C.A.R., Communicating Sequential Processes. Comm. ACM, Vol.21, No.8 (1978), pp.666-677.

[IPSJ88] 大特集：オブジェクト指向プログラミング．情報処理，Vol.29，No.4 (1988)．

[Kahn77] Kahn, G. and MacQueen, D.B., Coroutines and Networks of Parallel Processes. In Proc. IFIP'77, North-Holland, 1977, pp.993-998.

[Kato88]加藤和彦，Miranda．bit，Vol.20，No.8 (1988年８月号)．pp.35-45．

[Kowalski74] Kowalski, R., Predicate Logic as Programming Language. In Proc. IFIP '74, North-Holland, 1974, pp.569-574.

[Morita89] 森田正雄，上田和紀，ＧＨＣプログラムの最適化．In Proc. Logic Programming Conference '89，ICOT，1989．

[Onai86] 尾内理紀夫，Occam とトランスピュータ，共立出版，1986．

[Robinson65] Robinson, J.A., A Machine-Oriented Logic Based on Resolution Principle. J. ACM, Vol.12, No.1 (1965), pp.23-41.

[Shapiro86] Shapiro, E.Y., Concurrent Prolog: A Progress Report. Computer, Vol.19, No.8 (1986), pp.44-58.

[Shapiro87] Shapiro, E.Y. (ed.), Concurrent Prolog: Collected Papers, Vol.1-2. The MIT Press, 1987.

[Shibayama88a] 柴山悦哉，オブジェクト指向．bit，Vol.20，No.6 (1988 年６月号)．pp.42-54．

[Shibayama88b] 柴山悦哉，米澤明憲，並列オブジェクト指向言語ABCL/1（連載）．bit，Vol.20，No.7-9 (1988 年７－９月号)．

[Ueda86] 上田和紀，並列プログラミング言語．情報処理，Vol.27，No.9 (1986)，pp.995-1004．

[Ueda87] 上田和紀，並列プログラミング言語ＧＨＣの設計思想．bit，Vol.19，No.12 (1987年11月号)，pp.4-14．

[Ueda88a] Ueda, K., Guarded Horn Clauses: A Parallel Logic Programming Language with the Concept of a Guard. In Programming of Future Generation Computers, Nivat, M. and Fuchi, K. (eds.), North-Holland, 1988, pp.441-456.

[Ueda88b] Ueda, K. and Furukawa, K., Transformation Rules for GHC Programs. In Proc. Int. Conf. on FGCS'88, ICOT, 1988, pp.582-591.

[Ueda88c] 上田和紀，並列プログラミングとＧＨＣ．bit，Vol.20，No.10 (1988年10月号)，pp.83-98．

[Yonezawa87] Yonezawa, A. and Tokoro, M. (eds.), Object-Oriented Concurrent Programming. MIT Press, 1987.