インターフェイスの街角– 並列プログラミング

増井 俊之

並列プログラミングが流行らない理由

〝並列プログラミング〞は、ひと昔前には計算機科学の研究テーマとして人気がありましたが、最近はあまり話題にならないように思います。プロセスやスレッドを使うのが当り前になったのが一因かもしれませんが、それよりもむしろ、

- そもそも、並列プログラミングを必要とするプログラムは少ない
- 並列プログラミングは普通の人には難しい
- 並列プログラミングにより効率が上がったり、プログラムが分かりやすくなることはあまりない
- 並列プログラミングを自然に適用できる問題領域が少ない

といったあたりが〝流行らない〞理由ではないでしょうか。

ひろく使われているフィルタなど、大部分のプログラムはシングルタスクで動きますし、複数のタスクを使用するものも、たいていは fork() などによってプロセスを起動して疎な通信をおこなっています。一般的なアプリケーション・プログラムでは、並列プログラミングはほとんど使われていないのが実情でしょう。

UNIX 上では並列プログラミングはあまりポピュラーではありませんが、プロセスやスレッドよりも小さな単位で気軽に並列処理をおこなえれば便利な場面は多いはずです。たとえば、複数のデバイスからの入力を待つ場合は、select() のようなシステムコールを使ってポーリングをおこなう手法が一般的です。しかし、効率などを考えなければ、入力装置それぞれにプロセスを割り当てて処理させたほうがプログラムがすっきりするかもしれません。また、いろいろな機能をサブルーチンとして呼び出すよりもコルーチンとして並行動作させたほうが、制御の流れが分かりやすくなることもあります。

これらの手法があまり使われていないのは、UNIX や C 言語で手軽に並列プログラミングをおこなう方法がないからだと思います。

インターフェイスのプログラミングの問題点

並列処理機能をもたない手続き型言語を用いてユーザー・インターフェイスのプログラムを開発する場合、次のような点が問題になります。

制御の主体の問題

プログラムのどの部分が全体の実行の流れを制御するかは、きわめて重要です。

C や Java、Lisp などポピュラーな計算機言語では、主要な言語機能のほとんどすべては内部計算のためのものであり、入出力装置とのあいだのやりとりについては、文字ストリームを基本とした単純なものしか考慮されていません。入出力装置の主体が文字端末やキーボードであった時代には、計算機の負荷の大部分は内部計算であったため、それでも問題はありませんでした。しかし、グラフィック画面や複数の複雑な入力装置が使われる現在では、入出力装置の制御のほうが難しくなっています。このため、制御の主体を入出力装置側に移動させ、本来の計算がサブルーチンなどのかたちで入出力装置側から〝コールバック関数〞として呼び出される形式のインターフェイス・ツール

キットがひろく使われています。

単純なアプリケーションであれば、このようにインターフェイスを主体としたプログラミングでも問題はありません。しかし、制御構造が複雑で、かつ制御の移動がランタイムに決定されるようなアプリケーションには、インターフェイスによる制御を主体としたプログラミングは不向きです。たとえば、ユーザーと対話をおこないながら検索を絞り込んでいくようなシステムの場合には、システムの状態に応じて対話形式も変えていく必要があります。ところがツールキットを使うと、ユーザーの同じ操作からはいつも同じ関数が呼ばれてしまうため、システムの状態変数や条件分岐を大量に用意しなければなりません。

一般に逐次型のプログラミング言語では、自分が主体となっている場合とサブルーチンとして呼ばれる場合とでは、アルゴリズムを大幅に変更する必要があります。このため、端末インターフェイスを扱うアプリケーションとウィンドウ・システム上のアプリケーションでは、まったく異なる制御構造を利用しなければならないといったことがしばしば起こります。

一般的なアプリケーションにおける制御の移動に関しては、通常の逐次処理言語によるアルゴリズムのノウハウが蓄積されているので、それをそのまま使えばいいでしょう。しかし、採用するインターフェイスによってアプリケーション・プログラムの書き方を変えなければならないのではひどく不便です。

複数の入力を処理する問題

従来の言語では、一度に入出力ができるのは 1 つの装置に対してだけであり、複数の入出力装置を扱うのは困難でした。現在使われているほとんどのツールキットでは、すべての入力を〝イベント〟として同じデータ形式にして扱う方法がとられています。しかし、あらゆる入力装置や状態変化などを単一の形式にして扱うのは無理がありますし、プログラム自体も無用に複雑になってしまいます。

…

これらの問題点は一般にはそれほど重視されておらず、むしろ当り前と思われていることが多いようです。しかし、単純で柔軟な並列プログラミングさえできれば、あっさり解決してしまうでしょう。

並列プログラミングの利点

以下に記すように、並列プログラミングはユーザー・インターフェイスのプログラムを作成する場合にきわめて効果的です。

- アプリケーションとインターフェイスの分離
  プログラムの入出力部と本体が分離できると、いろいろな点で便利になります。まず、同じプログラムに対して異なるインターフェイスが使えます。たとえば、辞書検索アプリケーションでは、検索サーバー部と単語入力/結果表示部に分離して実装できます。こうしておけば、同じ検索サーバーをコマンドライン・インターフェイスからも GUI ツールキットからも使えます。また、入出力モジュールの代わりにテストモジュールを使い、自動的に本体アプリケーションのテストをおこなったり、デモモジュールを用いてデモ画面を表示させたりすることができます。
  多くのツールキットでは、入出力部とそれ以外の部分は分離できないのが普通です。そのため、特殊な機構を用いて、ポインティング・デバイスやキーボードのイベントを自動生成し、GUI のテストなどをおこなう仕組みになっているものが多いようです。しかし、はじめから入出力が分離されていれば、特殊な機構を用意しなくてもテストや自動デモなどを自由におこなえるはずです。
- 複数の入出力装置の使用
  たいていの GUI システムは、キーボードやマウスなどの複数の入力装置に対応していますし、テキスト枠やチェックボックスといった GUI 部品も同時に使えます。しかし、これらの処理は 1 つのプロセスで実現されていますから、あらゆる入力を〝イベント〟として一本化して受け取り、その後に必要に応じてディスパッチするようになっているのが普通です。このため、キーボード入力はテキスト枠でしか使わないと決まっている場合でも、すべての入力を同じ形式のイベントとしてまとめて扱わなければなりません。また、新たな入力装置を導入した場合も同じ形式に変換する必要があります。すべての入力やモジュールが独立して並列に動作するシステムであれば、このような工夫は不要になります。
- モジュールの分離

独立した機能の実行は、独立したプロセスでおこなうほうが便利です。現在、大部分のアプリケーションでは印刷機能は別のプロセスで実行されるようになっているので、データの印刷中にも編集作業を続行することができます。ところが、画像処理のように時間のかかる処理を実行しているときは、それ以外の操作がおこなえないアプリケーションも少なくありません。独立した機能は、すべて別のモジュールとして並列動作するようにしておけば、無意味に待たされたりもせず、キャンセル処理なども簡単になります。

●プログラム構造が簡単になる

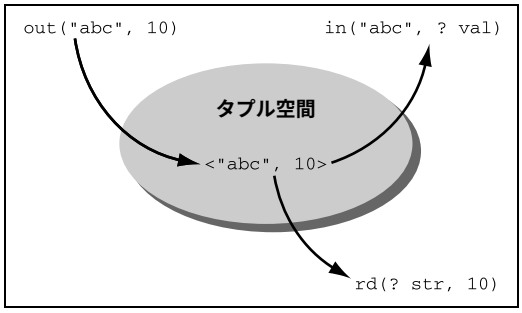
複雑なアルゴリズムで動く部分が複数ある場合には、これらをまとめて1つのプログラムとして動かすより、並列プログラムとして扱うほうが簡単です。

以上のように、ユーザー・インターフェイスのプログラムを作成する場合、並列プログラミングが有用であることは明らかです。協調する小さな規模のプロセスを気軽にたくさん使えれば、プログラムの構造がすっきりしますし、テストなども楽になるでしょう。それにもかかわらず、現実には並列プログラミングはほとんど利用されていません。その理由は、さきほども述べたとおり、UNIXやC言語では並列プログラミングが手軽におこなえないため、利用するのが面倒だという点につきるのではないでしょうか。

最近は各種のアプリケーション・サーバーの利用が流行っていますが、これらを正しく使うためのプロトコルやライブラリは複雑です。辞書サーバーのように比較的単純なものを使う場合も、正しい手順でコネクションを張り、決められたプロトコルで機能を順番に呼び出す必要があります。これを実現するプログラミングはそう簡単ではなく、おもしろいものでもありません。サーバーを作成するのはさらに面倒です。

サーバー、クライアントの区別なく、粒度の小さいプロセスを手軽に作ってプロセス間通信やデータ共有ができるのなら、多くの問題が解決するはずです。このような目的のために、Lindaという並列プログラミング機構が提案されています。

図1　Lindaの基本操作



Linda

Lindaは、複数のプロセスで共有される空間を用いてプロセス間通信やデータ共有をサポートする分散並列プログラム記述言語です[1]。プロセスが共有する空間はタプル空間(Tuple Space)と呼ばれ、タプル空間内のデータ(タプル)を使って通信やデータ共有をおこないます。このようにLindaのモデルはきわめて単純ですが、柔軟で強力なプロセス間通信を容易に記述することができます。

概要

Lindaではout、in、rdの3つの基本操作によってプロセス間通信をおこないます(図1)。

out：新しいデータ・オブジェクト(タプル)を生成し、プロセス間で共有されているタプル空間に置きます。たとえば、

```
out("a string", 15.01, 17, "another string");
out(0,1);
```

とすると、タプル空間内に、

```
<"a string", 15.01, 17, "another string">
<0, 1>
```

というタプルが生成されます。

in：指定した形式にマッチするタプルがタプル空間内にあるかどうかを調べ、みつかった場合は指定した変数にタプルの値を代入し、タプルをタプル空間から削除します。たとえば、上の例のタプルが存在する状態で、

```
in("a string", ? f, ? i, "another string")
```

を実行すると、inの引数のパターンが上記のタプルとマッチするので、fに15.01が、iに17がそれぞれ代

図 2　遠隔手続き呼出し

```
require 'linda.rb'

val = gets.to_i

TS.out (["double proc", val, $$])
pat, result = TS.in(["double proc result", nil, $$])
puts "Result is " + result.to_s
```

図 3　遠隔手続きサーバー

```
require 'linda.rb'

while true do
  pat, arg, proc = TS.in(["double proc", nil, nil])
  TS.out(["double proc result", arg * 2, proc])
end
```

入されたあと、タプルがタプル空間から削除されます。in のパターンにマッチするタプルが複数ある場合は、そのうちの 1 つが非決定的に選択されます。

rd：rd は in と同じ処理をおこないますが、タプルをタプル空間から削除しません。

一般的なメッセージ通信の場合とは異なり、タプルはただのデータであり、メッセージの送信先や受信先などの指定もおこなわれません。任意の数のプロセスが 1 つのメッセージ（タプル）を読めますし、送り手はどの受け手がいつメッセージを受け取るかを気にする必要もありません。つまり、送り手と受け手が相手を直接知らなくても通信することができるわけです。

Linda の機能はこのように単純ですが、これだけでプロセス間通信、同期、データ共有など、並列プログラミングに必要なあらゆる機能が実現できます[1]。

Linda の使用例

ここまでは C の構文を用いて説明してきましたが[2]、以降では Ruby で Linda を実装した `Rinda´を使い、実際に動くプログラムの例をいくつか紹介しましょう。

Rinda は、関 将俊氏が開発した分散オブジェクト指向プログラミング環境 dRuby [3] で Linda を実装したものです。dRuby の配布パッケージ[3]に含まれており、dRuby をインストールすればすぐに Rinda が利用できます。

Rinda でさきほどの out と in を呼び出すには、Ruby の配列を以下のように使います（誌面の都合上、⇒ で折り返しています。以下同様）。

```
out(["a string", 15.01, 17, "another string"])
out([0, 1])
pat1, f, i, pat2 = in(["a string", nil, nil, ⇒
"another string"])
```

この Rinda を例に、Linda のプロセス間通信について説明します。

遠隔手続き呼出し

図 2 は、別プロセスの処理を計算サーバーに依頼して結果を受け取るプログラムです。自分の出した要求に対する結果を確実に受け取るため、自分のプロセス番号を out でサーバーに伝え、その番号をもつタプルを in で受け取るようにしています。

計算サーバーのプログラムを図 3 に示します。クライアントからの要求をつねに in で監視し、パターンにマッチするタプルが出現したら、その値を取得して計算した結果を返すという動作を無限に繰り返します。

dRuby と Rinda の初期設定をおこなうために、下記の linda.rb を使っています。

```
require 'drb/drb'
require 'drb/rinda'

uri = ARGV.shift || raise("usage: #{$0} ⇒
 <server_uri>")
```

1 Linda のいろいろな利用法については、Linda を例として使っている並列プログラミングの教科書 [2] に詳しく解説されています。

2 Linda の提唱者である David Galernter の論文では C が使われていますが、`? i´のような独自の記法が使われているため、通常の処理系で実装することはできません。

3 http://www2a.biglobe.ne.jp/~seki/ruby/druby.html

図 4　哲学者の食事問題

```
require 'linda.rb'

NUM = 5

def think(i)
  puts "Philosopher #{i} is thinking..."
end

def eat(i)
  puts "Philosopher #{i} starts eating..."
end

def eatend(i)
  puts "Philosopher #{i} ends eating..."
end

def phil(i)
  while true do
    think(i)
    TS.in(["room ticket"])
    TS.in(["chopstick", i])
    TS.in(["chopstick", (i+1) % NUM])
    eat(i)
    eatend(i)
    TS.out(["chopstick", i])
    TS.out(["chopstick", (i+1) % NUM])
    TS.out(["room ticket"])
  end
end

(0...NUM).each { |i|
  TS.out(["chopstick", i])
  Thread.new { #  eval(phil(i))
    phil(i)
  }
  if i < NUM - 1 then
    TS.out(["room ticket"])
  end
}

sleep 100000
```

```
DRb.start_service
TS = TupleSpaceProxy.new⇒
(DRbObject.new(nil, uri))
```

哲学者の食事

有名な〝哲学者の食事〟問題[4]は、図 4 のプログラムで実現できます。room ticket を 4 枚しか発行しないことにより、デッドロックを防いでいます。

実行結果は以下のようになります。

```
Philosopher 0 is thinking...
Philosopher 1 is thinking...
Philosopher 2 is thinking...
Philosopher 0 starts eating...
Philosopher 3 is thinking...
Philosopher 4 is thinking...
Philosopher 3 starts eating...
Philosopher 0 ends eating...
Philosopher 0 is thinking...
Philosopher 3 ends eating...
Philosopher 3 is thinking...
Philosopher 2 starts eating...
Philosopher 4 starts eating...
Philosopher 2 ends eating...
Philosopher 4 ends eating...
  ......
```

Linda の利点

Linda は、以下のような点においてユーザー・インターフェイスの記述に適しています。

- 単純さ

  並列動作の記述プリミティブとして必要なのは out()、in()、rd() の 3 種類だけであり、これらの意味は単純明快です。

- 時間/空間独立性

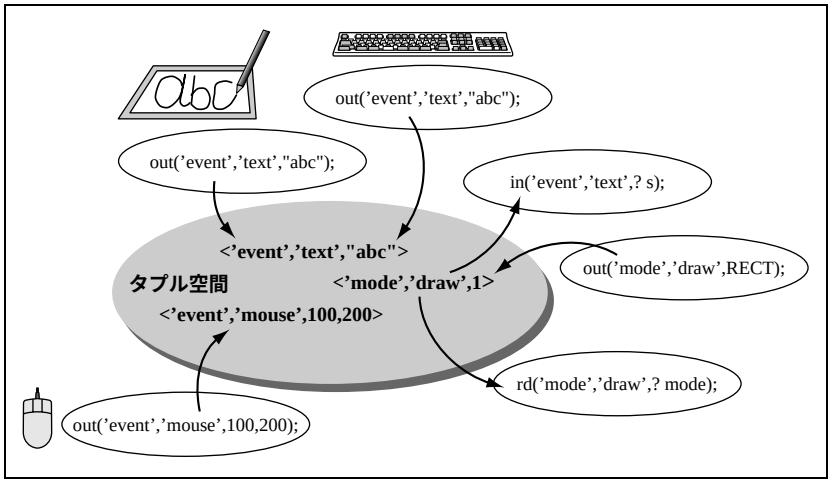
  Linda の通信はタプル空間を介して間接的に実行され、各プロセスが時間的、空間的に独立していてもかまいません。このため、インターフェイス部もアプリケーション部も互いに相手のことを意識せずに、タプルの仕様のみを考慮して動けばよいことになります。

  たとえば、ヘルプ機能を作成するとしましょう。ヘルプ機能は、いろいろなアプリケーションから同じようなインターフェイスで使えると便利ですが、アプリケーションの個々のダイアログ記述にこれを呼び出す機能を実装するのは面倒です。Linda を利用した場合には、ヘルプ機能のためのプロセスを 1 つ用意しておけば、どのようなアプリケーションからでも〝ヘルプが必要〟というタプルを出力するだけで求めるヘルプ機能を呼び出すことができます。

  時間的、空間的にプロセスが独立しているのは、アプリケーションとインターフェイスの分離という点において

4　5 人の哲学者が、中央にスパゲッティが盛られた大皿を囲んで円卓についています。彼らの前には、それぞれ皿が 1 枚とその脇にフォークが 1 本置かれています。スパゲッティをとりわけるには、かならず 2 本のフォークを使わなければなりません。哲学者たちは思索し、空腹になるとフォークに手を伸ばしてスパゲッティをとろうとします。このとき、哲学者たちをうまく協調させ、彼らを餓死させないようにするにはどうすればよいかという問題です。1965 年に Edsger W. Dijkstra が提案したもので、並行プロセス間で同期をとる際の問題としてよく引用されます。

図 5　タプル空間を介したアプリケーションとインターフェイスの通信



とくに効果的です。

- パターンマッチによる選択性

  in のパターンマッチ機能を利用して、タプル空間内のタプルをアプリケーションが選択的に取り込むことができます。たとえば、あるアプリケーションはあらゆるイベントを取得し、別のアプリケーションはキーボード・イベントだけ取り込むといったことも可能です。Rinda の場合は、

```
in(["event", nil, nil])
in(["event","key", nil])
```

  のように指定します。

- 言語独立性

  Linda は並列プログラミング・ライブラリとして実装できるので、各種の言語上での実装も比較的容易です。タプル空間とタプルの形式を同じにしておけば、異なる言語間での通信も簡単です。

- 実現が容易

  Linda は単一プロセッサの計算機はもとより、dRuby などの枠組みを利用すれば、ネットワーク上の複数の計算機でも容易に実装できます。

インターフェイス・モジュールとアプリケーション・モジュールが、タプル空間を使って通信する様子を図 5 に示します。文字列イベントを取得するアプリケーションは、それがどの装置から送られてきたかをまったく意識する必要はありません。選択しなければならない場合も、in を書き換えるだけですみます。

図 6　入出力処理の比較

(a) C による単純な実装

```
  ......
printf("name? ");
scanf("%s", name);
printf("age? ");
scanf("%d", &age);
search(name,age);
  ......
```

(b) Rinda を利用

```
  ......
in(? name, ? age);
search(name,age);
  ......
```

Linda の独立性、選択性が重要であることがよく分かるのではないでしょうか。

ユーザー・インターフェイスの作成

アプリケーションとユーザー・インターフェイスを、それぞれ Linda のプロセスとして分離して実装すると、制御の移動の問題は簡単に解決します。

データベース・アクセス

名前と年齢をキーとし、名簿を検索するデータベースにアクセスするプログラムについて考えてみましょう。

図 6-a は、scanf() などを使って単純に実装したプログラムです。一方、Rinda を利用すれば、アプリケーション部だけを分離して図 6-b のように書くことができます。in に対応する out は入出力プロセスが発行します。端末インターフェイスと GUI のどちらに対応する入出力プロセスであっても、同じタプルを出力するものはアプリケー

図 7　8-Queen プログラム

```
QUEENS = 8

def printqueens(qpos)
  puts ((0...QUEENS).collect { |i|
qpos[i].to_s }.join(" "))
end

def expand(n,col,up,down,qpos)
  (0...QUEENS).each { |c|
    if col[c] == 0 && up[n+c] == 0 &&
        down[n-c+QUEENS] == 0 then
      qpos[n] = c
      if n+1 >= QUEENS then
        printqueens(qpos)
      else
        col[c] = up[n+c] = down[n-c+QUEENS] = 1
        expand(n+1,col,up,down,qpos)
        col[c] = up[n+c] = down[n-c+QUEENS] = 0
      end
    end
  }
end

col, qpos, up, down = [[], [], [], []]
(0...QUEENS).each { |i|
  col[i] = qpos[i] = 0
}
(-QUEENS...QUEENS).each { |i|
  up[i+QUEENS] = 0
  down[i+QUEENS] = 0
}

while true do
  expand(0,col,up,down,qpos)
end
```

ションからは同じにみえるため、アプリケーションとインターフェイスの完全な分離が実現されることになります。また、ここではアプリケーション側もインターフェイス側もコルーチンとして制御の主導権をもっているので、柔軟なアルゴリズムが使えます。

8-Queen のインターフェイス

次に、〝8-Queen パズル〟[5]を解くプログラムについて考えてみます。試行錯誤により解を探索し、解がみつかった時点で結果を印刷するプログラムは、再帰呼出しを用いて図 7 のように比較的簡単に書くことができます。

このプログラムを実行すると、以下のような出力が得られます。

```
0 4 7 5 2 6 1 3
0 5 7 2 6 3 1 4
0 6 3 5 7 1 4 2
0 6 4 7 1 3 5 2
1 3 5 7 2 0 6 4
1 4 6 0 2 7 5 3
1 4 6 3 0 7 5 2
1 5 0 6 3 7 2 4
1 5 7 2 0 3 6 4
  ......
```

ところが、ウィンドウ上にチェス盤を表示し、クリックするたびに次の解を表示するようなプログラムが簡単に作れるインターフェイス・ツールキットはほとんどありません。一般的なツールキットではメインループがツールキットの内部に存在し、インターフェイス部品に対するユーザーの操作にコールバック関数を割り当てる形式になっていますが、図 7 のようなアルゴリズムはコールバック関数として呼ぶことができないからです。このため、そのようなツールキットを使う場合は、解生成のアルゴリズムを全面的に変更しなければなりません。

入出力部と計算部を別プロセスとし、Linda を用いて同期を実行するようにすれば、この問題は簡単に解決します。まず、図 7 のプログラムを図 8 のようにすこしだけ変更し、計算のタイミングと結果報告について別プロセスと通信できるようにします。

8-Queen の入出力だけを扱う図 9 のようなプログラムと通信することにより、さきほどの例とまったく同じ出力結果が得られます。

マウスのクリックによって解の生成と出力表示をおこなうプログラムを作る場合は、マウスクリックに対して、

```
TS.out(['solve'])
```

をコールバックとして定義しておき、別プロセスで、

```
while true do
  ans = TS.in(Array.new(QUEENS,nil))
```

5 8 × 8 マスのチェス盤上に、8 個の Queen をどのように配置すれば、互いにほかのコマが取れないようになるか、という問題です。

図 8　8-Queen のアプリケーション部

```
require 'linda.rb'

QUEENS = 8

def expand(n,col,up,down,qpos)
  (0...QUEENS).each { |c|
    if col[c] == 0 && up[n+c] == 0 &&
        down[n-c+QUEENS] == 0 then
      qpos[n] = c
      if n+1 >= QUEENS then
        TS.in(['solve'])
        TS.out(qpos)
      else
        col[c] = up[n+c] = down[n-c+QUEENS] = 1
        expand(n+1,col,up,down,qpos)
        col[c] = up[n+c] = down[n-c+QUEENS] = 0
      end
    end
  }
end

col, qpos, up, down = [[], [], [], []]
(0...QUEENS).each { |i|
  col[i] = qpos[i] = 0
}
(-QUEENS...QUEENS).each { |i|
  up[i+QUEENS] = 0
  down[i+QUEENS] = 0
}

while true do
  expand(0,col,up,down,qpos)
end
```

図 9　8-Queen のインターフェイス部

```
require 'linda.rb'

QUEENS = 8

while(true) do
  TS.out(['solve'])
  ans = TS.in(Array.new(QUEENS,nil))
  puts ((0...QUEENS).collect { |i| ans[i].to_s }.join(" "))
end
```

```
  （ansの内容をグラフィック表示）
end
```

のようなプログラムを動かしておけばよいことになります。この方式にかぎらず、図 8 のアプリケーション・プログラムは任意のインターフェイスをもつプログラムで利用できます。

Linda を使えば、このようにコルーチンの実現が容易になるだけでなく、異なるタイプの同期手法も簡単に実現できます。タプルのパターン・マッチング機能を利用すると、さらに高度な通信も可能になります。

おわりに

最近は、並列プログラミングをサポートする言語やツールキットも増えてきましたが、細かい粒度でのインターフェイスの並列プログラミングはほとんどおこなわれていないようです。

今回紹介した Linda/Rinda の例からも分かるように、並列プログラミングは単純であるにもかかわらず、きわめて強力な枠組みです。この種のシステムをもっと積極的に活用してもよいのではないでしょうか。

（ますい・としゆき　ソニー CSL）

[参考文献]

[1] David Gelernter, “Generative Communication In Linda”, *ACM Transactions on Programming Languages and Systems*, pp. 80–112, January 1985

[2] Nicholas Carriero and David Gelernter, *How To Write Parallel Programs: A First Course*, The MIT Press, 1990

[3] 関 将俊『dRuby による分散オブジェクトプログラミング』、アスキー、2001 年