並行・並列プログラミングは好きですか？

近山　隆／東京大学

並行処理はいろいろな新しい可能性を開くが，プログラミングは難しい．しかし世の趨勢では大容量処理の性能費用比向上に並列処理が，また並列処理には並行処理が不可欠であり，避けては通れない．言語や言語処理系などのツールを駆使して並行処理の難しさを軽減し，なるべく楽に並列処理のメリットを享受したい．

はじめに

「マルチメディア」という言葉がはやりはじめてからずいぶんになる．マルチメディアというからには数多くのメディアかと思ったら，人間とのインタフェースは目と耳，光と音の2つのメディア以外はほとんど使わない．シングルメディアではないが，マルチメディアというほどのものでもないように思える．色鉛筆と水彩絵具と蛍光マーカを使い分けて描いた絵もマルチメディアだろうか．それはともかく「マルチメディア」の処理にはいろいろなタイミングで動くさまざまな入出力を上手に使うために，並行処理が必要なのだそうである．

あれやこれやといろいろな入出力手段を駆使するというのは，今に始まったことではない．あっちのディスクから1セクタ読み，こっちのラインプリンタはバッファの中身を打ち出し終わったかと思うとオペレータがコンソールからキーインと，うるさく割り込んでくるのをさばきながらバッチジョブを次々とこなすというのは，昔からオペレーティングシステムの得意技だった．それなのに，なぜマルチメディアの処理ではアプリケーションでいまさらこんな面倒な仕事をしなければならないのだ．

昔ながらのOSの並行処理は実はそう複雑なものではなかったのだ．OSの下ではさまざまなプログラムが走っているが，それぞれは個別のタスクであり，他のタスクがどう進もうと自分の仕事さえ進めばよい．だからOSはCPU時間などの資源をどう公平に分け与えるかだけ考えればよかった．各タスクは同時に入力と出力をしたりしないのがふつうなので，入力と出力は独立で無関係と思っていればよかった．「マルチメディア」の並行処理ではそうはいかない．あっちの入力がくればこっちの出力はやめる，こっちの計算が早めに終われば今の画面出力は中断して別の画面を出力する．

それにしてもこんな面倒な処理をアプリケーションでいちいち書かねばならないというのはまちがっている．あまりに面倒なので，入力をポーリングしながらループするなどという，入出力重点の書き方が主流になってしまった．イベントループでは一心不乱に計算することはできない．いつも何かが起きるのではないかとびくびくしながら，少しずつは計算もしているという態である．能率が上がらないことはなはだしく，計算の本体についてはかなり書きにくい．

並行処理は難しい

では本格的に並行処理をするプログラムを自分で書いてみようとするとどうなるだろう．どうしても難しくなる理由は3つある．

(1) 考えることが多い

いろいろな計算や入出力が並行して進んでいくので，計算の進行がどうなるかはプログラムの字面だけからは読み取ることができない．どんな進み方になってもよいように書かなければならない上，共有リソース(大域変数だの，入出力装置だの)アクセスの仕方が不整合にならないように注意し，デッドロックだのスターベーションだの，逐次処理ならまったく必要がなかった面倒なことを考えなければならない．

(2) デバッグが難しい

プログラムは書けたつもりになっても，デバッグするのが難しい．うまく動いたと思っていたプログラムも，ちょっと状況が変わると全然動かなくなってしまう．バグに悩まされ，途中状態をファイルにダンプす

れば，それでタイミングが変わってバグはカクレンボしてしまう．入出力やプロセス切り替えのタイミングがちょっと変われば，全然違う動きをするのは珍しくない．

(3) OSやツールが不親切だ

オペレーティングシステムも言語処理系もデバッグ用のツール類も，ほとんどは逐次処理のために作ったものだ．並行処理のために拡張がついていたりもするが，あまり使い手がいないためか仕様が洗練されておらず，バグだらけで，実際には使い物にならないことが少なくない．OSでも並行処理機能については結構バグが潜んでいて悩まされるし，動いても信じられないような低効率であることもある[1]．

並列処理には並行処理が欠かせない

なるべく並行処理はしないで済ませたいのだが，そうもいかない理由はある．LSIという，設計はたいへんだがコピーはたいへん安く作れる技術が発展してしまった。おかげで，性能費用比を高くするのに並列処理が断然有利になってしまった．物理的な並列処理を実現するには，だれかがどこかで並行処理をしなくてはならない．アプリケーションかもしれないし，コンパイラかもしれないし，ハードウェアの機構かもしれない．とにかく，どこかに並行処理を意識する層が必要である．

もちろん，ハードウェアによる自動並列処理は有力で，パイプラインもスーパスカラも立派に役立っている．しかし，いかんせん到達できる並列度は多寡が知れている．逐次処理用言語のプログラムを自動並列化するコンパイル技術も進んできているが，行列計算や比較的単純な探索など決まりきった処理は別として，ちょっと複雑に絡み合う計算となるとうまくいかないことが多い．問題に依存するさまざまな並列化手法を全部適切に使いこなせるコンパイラなどあるはずもない．

自動的にうまくいかないところは当面人間が介在するしかない．問題の性質をよくわかっている人間が，それに合わせた並列化を行うしかなかろう．私が計算機プログラミングが好きなのは，とりあえず仕様通りに動くように作れるからではない．醍醐味はプログラムの構成をすっきりきれいにして，それが一方では処理速度の向上にもつながったときである．使用メモリ量の削減も面白いのだが，近頃の計算機はメモリはたっぷりついているので，むしろワーキングセットの削減が大事で，結局は処理速度の問題になる．並列プログラミングはこの楽しみに新しい地平を開く．逐次処理で楽しかったアルゴリズムやデータ構造の工夫が，並列処理によって組合せ的自由度を大きく増す．機械語命令数を減らすような低レベルの最適化の楽しみは，最適化コンパイラに奪われてしまった．その一方で開けてきたこの並列処理という次元では，コンパイラが人間を凌駕するのにまだまだ時間がかかりそうで，当分楽しめそうだ．

並列処理ではいかに通信遅延が効率低下の原因にならないようにするかがポイントになるわけだが，通信する代わりに再計算するとかいった定石の他に，通信遅延で空いた時間に何をするかを考えると，アルゴリズムレベルまで戻って大幅な変更をしないとうまく効率化できない場合が少なくない．そこで問題になるのが並行プログラミングの難しさなのだ．ちょっと複雑な並列アルゴリズムを書こうとすると，すぐ上に述べたような問題につきあたる．大幅な変更などしようものなら，並行処理のさまざまな問題にひっかかって，きちんと動くまでにとてつもない労力を強いられる．これでは楽しんでいる余裕がない．

私自身の経験からすると，逐次処理のプログラムはとりあえず書き飛ばしておいても，後からゆっくりデバッグすればなんとか動くようになる．ところが並行プログラムときたら，動かないプログラムをデバッグして動くようにするのは非常に難しい．動くようにしたつもりでも，前述のように何かのきっかけで突然また動かなくなってしまう．なんとかバグがとれて動いても，根本的な方針変更をすると大々的なプログラム構造の変更が必要になることが少なくない．どの部分計算を並行に動かすのか，どこまでをどのプロセスにやらせるのか，というあたりが並行プログラムの重要な設計事項で，しかも選択肢が多い．ここから変えないときちんと動くものにできないことが多い．そうなると，従来は1つのプロセスだけが使っていたリソースを共有することになり，それまで必要がなかったロックが必要になったのに，それを忘れていてまたバグの泥沼に沈む．逆に1つのプロセスに処理が閉じるように変えたためロックが不要になったのを忘れていて，妙に効率が低くなる原因がわからずに悩んだりもする．とても楽しむ領域にまで踏み込めない．

ではどうしよう

並行処理が難しいのは(1)考えることが多く，(2)デバッグが難しく，(3)OSやツールが不親切だ，からである．この状況を変えればよいのだ．

まずは考えることを減らす．どんな順序で計算しても同じ結果になる合流性という性質を持つような記述をするのは，1つの方法である．考えることが減るのはよいのだが，それで楽しみも減ってしまってはしかたがない．計算や通信の順序やプロセッサへの割り付けなど，並列処理のおもしろいところは自分でコントロールしたい．だから合流性を持たせ，なおかつ物理的処理の制御が可能というような記述をしたくなる．

デバッグをやさしくするにも合流性は鍵になる．計算順序やプロセッサへの割り付けを変えただけで大幅に記述を変更しなければならず，その結果としてバグが入るような記述形式は困ったものだ．PVMだのMPIだのがはやっているが，単純な共有メモリよりはずっとましだが，アプ

リケーションユーザが安心して使えるものとは思えない.

この意味で並行並列論理型言語というのは理想的，とこの際我田引水しておく．第五世代コンピュータプロジェクトは人工知能のプロジェクトかのように思われているところがあるが，人工知能の研究もあったが並行並列処理(ただし大規模数値計算のような比較的規則性のある計算ではなく，ずっと不規則性が高い計算についての並行並列処理)に多くの力を注いだ．その中で作ったKL1はPrologのような自動探索機能を持つ言語ではなく，並行処理時の自動同期，並列処理時の自動分散データ管理などが特徴の言語である[2]．知識処理的な応用だけでなく「マルチメディア」の記述にも向いていて，プログラミングコンテストでもWWWブラウザやHTTPDなどの作品が入賞している[3].

おわりに

プログラミングはどんどん複雑になる．いくら書きやすい枠組みを作っても，使いやすいツールを作っても，ソフトウェアが楽に作れるようになればなるだけ，それまではとても作れなかったような複雑なソフトウェアを作ろうということになってしまう．幸か不幸か，いくら巨大で複雑なソフトウェアを作っても地球温暖化には直結しないので，さっぱり歯止めがかからない．我々の飯の種がなくならないという意味では大いに結構なのだが，ときどきどこで楽をして何をがんばるか考えないと，無駄な労力を費やすことになる．並行プログラミングでは楽をしよう．そして，並列プログラミングにもっと力を注げる余裕を作ろう.

参考文献
1)近山　隆: 呼んでも君は振り向いてくれない, 第39回プログラミング・シンポジウム報告集(1998).
2)Ueda, K. and Chikayama, T. : Design of the Kernel Language for Parallel Inference Machine, The Computer Journal 33-6, pp.494-500 (Dec. 1990).
3)第1回KLICプログラミングコンテスト入賞作品, http://www.icot.or.jp/AITEC/FGCS/KLICON/HYOSHO96-J/main-J.html

(1998.3.9)

並列処理に「難しい」並行処理は必要か

佐藤三久／新情報処理開発機構

「マルチメディア」には並行処理が便利そうだ.「ポーリングベースのプログラムは書きにくいし，たとえば，Javaのスレッドで書けばすっきり書けそうだ．並行処理のプログラミングは難しい．そのとおりである．同期のバグで随分悩んだこともあるし，デバックが済んだ(と思った)あとでも，思いもよらない非同期のイベントなどのバグが他にありそうで，すっきりしない.

で，私の論点は，並列処理には並行処理が欠かせないかということである．何のために，並列処理をするか．それは，処理(計算)を早くするためである．現在のところ，並列処理が必要とされる分野は結構時間が必要で，それを短縮したいという場合である．たとえば，物理的なシミュレーションなどの科学技術計算は典型的な例である．「行列計算や比較的単純な探索など決まりきった処理は別として」とあるが，これが並列処理を必要とする分野なのである．複雑な例は探せばあるかもしれないが，並列処理を必要とする時間のかかる処理はは往々にして，内在する並列性が高く，比較的簡単な場合が多いと思う．並列処理で，1時間のかかるプログラムを10分にするのは簡単だが，1分を10秒にするのは難しい．しかし，1分のプログラムは並列処理しなくても，十分早かったりする.

並列処理ではユーザ(プログラマ)は必ずしも並列処理を書く必要はない．むしろ，多くの逐次の世界にいるユーザのために，並列処理を書かなくてもよいことを目指している．並列処理では，並列性を記述することが目的ではない.

多くの場合，並列処理は面倒だが，並行処理ほど難しくないと思う．それは，高速化という観点でみれば，逐次でできていることをベースにできるからである．PMVやMPIでプログラムを並列化するのは面倒であるが，並列性という観点からはある程度の構造がある．完全な自動並列化はまだまだにしても，逐次プログラムにOpenMPのように並列化指示を加えるようなプログラミングという手もある.

PCやワークステションをイーサネットや高速ネットワークで結合して並列システムをつくるクラスタ技術で，高価なMPPがなくても結構な並列処理ができるようになってきた．また，マルチCPUのSMPタイプのPCやワークステーションも普及してきた．これら，身近になってきた並列のプラットフォームを多くのユーザが使いこなすには，やはり，自動並列化とまではいわなくても，これまで書かれてきた，そして，書かれるであろう逐次プログラムから連続性のある並列プログラミング環境が必要であろう．多くの人にとって，楽でも並行処理は必要ないにこしたことはない.

そもそも，並行処理あるいは並行

プログラミングは難しいのだ．逐次のデバッカのようなad-hocな対処ではなくて，論理的な証明のようなものが必要であろう．合流性などの性質を使うような言語からのサポートも有効かとおもう．しかし，並行プログラミングする必要がなければ，もっといい．言語やツールを駆使して，楽に並列処理のメリットを享受したい．まったく，賛成である．

(1998.4.21)

単純な並列処理はマイナーなのだ

近山　隆／東京大学

「行列計算や比較的単純な探索など決まりきった処理が並列処理を必要とする」ことにまったく異論はない．そして，そのような処理が現在の並列処理対象の主流であるのは事実である．だがそういう処理が今後とも主流であり続けるとは思えないのである．現段階の技術では「比較的簡単な場合」にしか並列処理ができない，あるいは，計算時間（あるいは逆に精度）を犠牲にしてでもむりやりそのような形式に押し込めて処理しているので，「並列処理の対象は比較的単純」という印象を与えているのが実情ではないだろうか．

私は「並行処理は面倒だが，並列処理ほどは難しくない」と思う．並行処理を逐次処理に基づいてやろうとするから面倒なので，最初から並行を前提とすればそれほどのこともない．本質的に難しいのは並列処理，つまり効率の追求の方なのだ．速くしようと思うから成熟技術である逐次処理をベースにしようと考え，そのために並行処理を難しくし，簡単に並列化できることしか並列化できないのである．

(1998.5.1)

マルチメディア屋としては，好き嫌いの問題じゃなくて・・・

太田昌孝／東京工業大学

ファラデー以来遠隔作用はさっぱりふるわず，実世界の物理は場の理論に基づいて近接作用で動いている．そして，相対性理論によれば「同時」という概念は観測系を決めないと決まらない相対的なものにしかすぎない．こんな世の中を計算機上でありのままに再現しようとすれば，複数のオブジェクトがメッセージをやりとりしながら非同期的に動作する並行処理になるのは理の当然である．

マルチメディアが単なる文字列による抽象的な入出力を越えて，人間の感覚を活かした実世界の物理をありのままに利用した具象的な入出力を行うものである以上，マルチメディアに並行処理が必要なのはあたりまえである．

ところが，世の中のコンピュータやそのソフトウェアは，世の中をありのままに扱うためではなく，世の中の数学モデルを厳密に扱うために作られている．そこで，並行処理により現実世界を厳密に再現しようとすれば，考えることは多いだろうし，デバッグも難しいだろうし，OSやツールは不親切だろう．まして，本来場としての並列性を持っている現実世界を単一プロセッサで扱おうとすれば，資源の割り当てだとか入力のポーリングだとかいった非本質的な処理に手間をとられるのも，これまたあたりまえである．

しかし，幸いなことに，「LSIという設計はたいへんだがコピーはたいへん安く作れる技術が発展して」くれたおかげで，マルチメディア処理において性能費用比を気にする必要はあまりない．複数オブジェクト間のメッセージのやりとりをありのままに扱いたければ，オブジェクトの

数だけのプロセッサを用意すればいいだけのことである．実際，マルチメディア処理の最先端を担ういまどきの家庭用ゲーム機は，当然のごとく，音声なら音声の処理だけを担当しているプロセッサなどを持つ．このような並列性はもともとマルチメディアに内在しているのであり，並列性の抽出に悩む必要もない．また，別々の場所にあるオブジェクトのあいだのやりとりは通信がすべてであり，もともとリソースなど共有していないので，共有変数の更新だとかロックだとかいう問題も存在しない．

もちろん，複数のプロセッサを厳密に協調させようとするのは，楽なことではない．ところが，マルチメディア処理において要求されるのは，人間の感じる「ありのまま」という臨場感であって，厳密性なんかかけらも必要ない．マルチメディアの並行処理を難しく感じるのは，

(1) 余計な厳密性を考えたり，
(2) 完璧なデバッグを目指したり，
(3) シングルプロセッサを使う

からである．この状況を変えればよいのだ．

まずは考えることを減らす．計算順序の多少の前後は現実世界では不可避であり結果は多少異なることになるが，そんなことはどうせ人間は気にしない．人間は楽しみさええられれば，それでいいのだ．そこで，合流性だのといったありもしない厳密性を言語やプログラマが気にするのはまちがっている．デバッグだって，システム全体がハングするようなバグはこまるが，多少の矛盾などはご愛嬌．ゲームの場合，裏技の種として評価をあげることにもなる．

この意味で，マルチメディア処理に必要なのは，感性的な，つまり非論理的プログラム言語である．全体の整合性など気にせず，部分部分のプログラムをイベント発生とその時に起動するシナリオを組み合わせて，結果的に全体がそれらしくまとまったら，それでいい．オブジェクト指向言語を出発点として，どんどんいいかげんにしてゆくとよいだろう．

なお，マルチメディアを，リアルタイム性の必要ない，単にハイパーリンクをたどるだけのものととらえると，WWWブラウザやHTTPDもマルチメディアということになるかもしれないが，マルチメディアのあまりに狭い捉えかたであり，今のインターネットが品質保証はできないし遅いネットワークなのでなんとか許容されているだけのことである．今後のインターネット上のアプリケーションは，インタラクティブネットワークゲームのほうに発展していくことだろう．

並列プログラムになんか力を注がず，並行プログラムでは楽をしよう．

(1998.4.30)

抽象化は効率の源泉である

近山　隆／東京大学

物理理論が近接作用であるからといって，それをそのまま計算していたのではいくら計算容量があっても足りない．「ありのままに」ではなく，合目的的に抽象化したモデルを扱う以外の方法は考えられない．論理回路を素粒子レベルでしかシミュレーションしないのでは，計算機のような規模のシステムは到底設計できまい．

性能費用比は常に重要である．

「音声処理だけ担当プロセッサ」では平気で無駄をしても，それはこの無駄がハードウェアコストの大勢に影響を与えないからである．ゲーム機コストの半分を占めるようなら別の手法を考える．ゲーム・キャラクタを構成する素粒子ごとに専用プロセッサを，あるいは逆に画面のピクセルごとに専用プロセッサを持たせればソフトウェア記述が楽になるとしても，それは性能費用比上できない．

感性を云々できるためには「臨場感」を持てるだけの現実世界との類似性が必要である．どんな前衛芸術にも古典的イディオムの利用が欠かせない．ゲームでも「臨場感」に直結する制約については決して破らないという，適切な抽象レベルにおいての厳密性が必要である．ゲームに登場する複数のキャラクタが互いの体を平気ですり抜け合ったりしたら，「臨場感」を損なうことこの上ない．「それらしくまとまる」ベースには論理性が不可欠である．

(1998.5.1)

～　議論の続きは，次のURLをご覧ください．http://www.ipsj.or.jp/magazine/interessay.html　～