勉強会資料
(並列言語)

鶴崎 剛大

平成１０年5月8日

1　はじめに

現在、コンピュータは通常の作業ならば問題なくこなせるほど高速になってきている。しかし、一方で流体力学、天気予報、人工知能など非常に大規模な応用も存在し、計算速度は決して十分ではない。このため並列計算機が必要になってくるが、逐次型言語に並列化のためのライブラリ等を追加してアルゴリズムを記述すると、同期の記述が繁雑になったり潜在的なバグ(実行するたびに違う現象が起きる)に悩まされることになる。従っていろいろなアルゴリズムを安全／簡単に記述でき、かつ並列実行による高速化の恩恵を得られるような並列言語が必要である。

また、並列処理が必要な応用のなかでも、流体力学と人工知能とでは主として処理すべき対象が違うため、それぞれに向いた並列言語が必要である。並列処理言語のなかには流体力学のような数値計算が主であるものに向くデータパラレル型並列言語や、人工知能のような記号処理が主であるものに向く並列論理型言語がある。

これら種々の並列処理言語に共通の課題としては、複数のプロセス間の情報交換をいかに扱うか、ということがある。

2　並列処理言語

並列処理言語に必要な機能は非同期に動作する並列(並行)プロセスを記述できることと、並列に動作するプロセス間相互作用を制御できることである。

2.1　並列処理言語の特質

以下に並列処理言語と呼ばれるものが備えるべき特質について述べる。

- 陽に並列／並行処理が書ける

  モジュール分割の単位を1プロセスごとにすることで並列処理を記述できる。あるいはモジュール内の並列実行可能な部分を指定することで、陽に並列処理を記述できる。

- 非定型的／非均質な並列処理が書ける

  高い抽象度でプロセスを記述できる。また、プロセスやプロセス間の関係のダイナミックな変化を記述できる。

- 並列処理の基本操作が安全／簡便に書ける

  排他／同期／通信など、プログラマの負担になりやすく、また、厄介なバグの入りやすい基本操作を安全／簡便に書けるようにする。例えば通信の手段を制限することで、同期を陽に書く必要をなくしたりする。

2.2　プロセス間通信

並列に動作する複数のプロセスが互いに情報交換する必要があるとき、この情報交換をプロセス間通信という。これには受け手が情報を受けとるまで送り手を待たせる同期式と、そうではない非同期式がある。

同期式通信では、送り手は情報が受信されたことを確認できるが、待ち時間の間は送り手はなんら生産的な仕事を行なえない。

非同期式通信では、バッファを介することで送り手が待つ必要がなくなるが、情報が受信されたことを確認できない。

2.3 排他制御と同期

複数のプロセスが並行に動作し、プロセス間通信が必要な場合、正しい実行結果を保証するために実行の順序を管理しなくてはならない。あるプロセスが他のプロセスによって条件を満されるまで待たされるのが同期である。

同期を実現するには、

- 陽に実現

  プロセス間通信に共有資源 (変数) を用いる場合、Condition Variable を用いて実現する。

- 陰に実現

  直接プロセス間でメッセージが転送される場合、自動的に同期されるので、陽に記述しなくてもよい。

といった方法がある。

また、複数のプロセス間通信が不可分に実行されなければならないようなとき、その部分をクリティカル・リージョンといい、あるプロセスがクリティカル・リージョンにあるときは他のプロセスは待合わせる必要がある。この制御を排他制御という。

排他制御は、

- 陽に実現

  プロセス間通信に共有資源 (変数) を用いる場合、共有資源とその操作 (手続) をまとめ (Monitor)、共有資源へのアクセスは Monitor を介してのみ可能とする。また、Monitor に入れるのは 1 プロセスのみとすることで実現できる。

- 陰に実現

  同じ送受信者間でのメッセージ順序が保存される仕組みを用意することで、自動的に実行順序が正しくなり、陽に排他制御を記述しなくてもよい。

といった実現方法がある。

3 並列言語の分類

- 関数型

  関数型言語では、プログラムを関数式のみで記述し、その実行はその関数の値の評価を行なうことでなされる。

  プログラムの各部分 (関数) が単独で意味を持つという参照透明性がある。これによって各関数の独立性が保証されているので、関数型言語で記述されたプログラムの処理には本質的な並列性が内在している。

  しかし、現在の Lisp のように手続き型言語パラダイムを取り入れた言語では、関数の評価を暗黙的に並列化すると処理システムごとに結果が異なるなどの問題が起きやすくなり、実用性が低くなる。そのため、プログラマ自身が明示的に並列処理を指定できるようなシンタクスを追加する方法で並列処理機能を備えたものが多い。

  Concurrent Lisp, Multilisp, CmLisp などの並列 Lisp が主である。

- 論理型

  論理型言語では、プログラムはある論理体系に従った論理式の集合で表される。論理型言語における変数とは、数学の方程式における変数のようなものであり、問題に内在する関係や条件などを論理式として記述すると、未定義の変数の値が決定されていき、問題の解がえられる。つまり、変数の値が一度決まるとそれを書換えることはできない。

  論理型並列言語には、まったく同じ計算を複数回行なっても結果が違うことがあるという非決定性がある。ガードが真である節が複数ある場合、どの節を選ぶかは一意には決まらない。このため、ひとつの述語について排他的になるようにガード条件を書くか、どの節を選んでも正しく動くようにプログラムを書かなければならない。これはデバッグを考えると不利だが、効率良く問題を解くという点では有利である。

  Concurrent Prolog(CP), PARLOG, GHC(Guarded Horn Clauses), KL1 などがある。

- オブジェクト指向

オブジェクトはもともと自己完備で独立性の高い計算実体であり、オブジェクト指向言語で記述されたプログラムは高度な並列性を内在している。

しかし、個々のオブジェクトの機能は様々であり、実使用上は物理的粒度を均一にすることが望まれる並列計算機上で、種々の大きさの論理的粒度のオブジェクトを処理しなくてはならず、性能の点で問題もある。

ABCL, Concurrent SmallTalk などがある。

- データパラレル

  データ並列計算のための言語。主として SIMD 型計算機を対象とし、データを PE(Processor Element) ごとに分散配置して演算を並列化する。手続き型の言語をベースとして拡張を施したものが多い。また、SIMD 型計算機を主な対象とするものの、それはセマンティクス上のことであり、MIMD 型計算機等でも利用可能であることが多い。

  C*, DPCE, WTC(work-time C), NESL, NCX, pC++ などがある。

4 NCX

超並列 C 言語 NCX は、データ並列計算を主とする超並列計算のための拡張 C 言語である。NCX は言語セマンティクス上は SIMD モデルを採用しており、実際の SIMD 型計算機の構造を意識した記述ができる言語である。しかしながら、比較的高水準の言語機能を提供することで、SPMD 型、MIMD 型、ワークステーションのいずれでも利用可能な言語である。

4.1 仮想プロセッサ

NCX のプログラムを実行する主体が仮想プロセッサである。NCX では、原理的に無限個の仮想プロセッサの存在を仮定しているが、プログラム実行中の任意の時点において、実行に参加している仮想プロセッサは有限個である。実行に参加している仮想プロセッサをその時点でアクティブ (active) であるといい、その他の仮想プロセッサをその時点でインアクティブ (inactive) であるという。

個々の仮想プロセッサがアクティブであるかどうか (アクティビティ) を変化させるには、if 文などによって現在アクティブである仮想プロセッサのうち、ある条件を満たすもののみを選択して実行を継続させ、条件を満たさないものをインアクティブにする方法がある。

4.2 フィールド

NCX では仮想プロセッサのある集合をフィールド (field) と呼び、プログラム中で複数のフィールドを宣言することができる。実行中の任意の時点でアクティブな仮想プロセッサ全体の集合は、常にあるフィールドの部分集合である。このようなフィールドをカレントフィールド (current field) と呼ぶ。

フィールドはあらかじめ用意されたいくつかのベースフィールド (base field) と呼ばれるもののうちいずれかのベースフィールドの部分集合である。ベースフィールドはなんらかのネットワークで結合された無限個の仮想プロセッサからなる集合であり、それ自体はフィールドではない。

ベースフィールドは次の 3 つが用意されている。

- mesh

  mesh は無限の次数を持ち、各次元の大きさが無限であるメッシュ(mesh) 状に配置された仮想プロセッサからなるベースフィールドである。

  個々の仮想プロセッサ $P_{mesh}(i_0, i_1, i_2, ...)$ は、

$$P_{mesh}(i_0 \pm 1, i_1, i_2, ...)$$

$$P_{mesh}(i_0, i_1 \pm 1, i_2, ...)$$

$$P_{mesh}(i_0, i_1, i_2 \pm 1, ...)$$

  等と直接結合されている。

- hypercube

  hypercube は無限の次数を持つ超立方体 (hypercube) 状に配置された仮想プロセッサからなるベースフィールドである。

  個々の仮想プロセッサ $P_{cube}(i)$ は、仮想プロセッサ番号を 2 進表記したときに、i とハミング (Hamming) 距離が 1 だけ異なる仮想プロセッサ番号を持つ仮想プロセッサど直接結合されている。すなわち、$P_{cube}(i)$ は、任意の非負整数 kに対して、

$$P_{cube}(i \oplus 2^k)$$

  と直接結合されている。ここで、’$\oplus$’ は排他的論理和を表す。

- binarytree

  binarytree は無限の深さを持つ二進木 (binary tree) 状に配置された仮想プロセッサからなるベースフィールドである。$P_{tree}(0)$ は左の子と右の子と、

$$P_{tree}(1)$$

$$P_{tree}(2)$$

  直接結合されており、その他の仮想プロセッサ $P_{tree}(i)$ は、その左の子、右の子、およびその親と、

$$P_{tree}(2i+1)$$

$$P_{tree}(2(i+1)$$

$$P_{tree}(\lfloor (i-1)/2 \rfloor)$$

  直接結合されている。

仮想プロセッサ間の情報交換は、任意の 2 つの仮想プロセッサ間で可能であるが、直接結合された仮想プロセッサ間の通信は、そうでない仮想プロセッサ間の通信よりも時間がかかったりしない (高速であると仮定してよい)。

4.3 変数

変数が宣言されると、指定されたフィールドを構成するすべての仮想プロセッサにその変数が割り当てられる。そのため、変数を参照するには変数名に加えて、どの仮想プロセッサに割り当てられたものであるかを指定する必要がある。これは、ポインタや配列についても同様である。

4.4 代入

代入式の実行は、まずアクティブなすべての仮想プロセッサが右辺を評価し、全仮想プロセッサがその値を求め終わったときに、代入を行う。左辺の示す場所が仮想プロセッサごとに異なれば、代入は単にＣ言語と同様の代入を、アクティブな全仮想プロセッサが実行するにすぎない。しかし、複数の仮想プロセッサが同じ場所に代入を行う場合は、その場所への代入は逐次的に行なわれる。その順序は規定されない。そのため、ＮＣＸにおける代入式の値は、すべての仮想プロセッサが代入を終えた時点の、代入先の値である。

単純な代入の場合には、意味的には最後に代入された値であるから、１回だけ代入を行なえばよい。もしそれが演算を伴う代入 (+=など) であれば、そこに代入しようとするすべての仮想プロセッサの代入式の値が加算されたものになる。

4.5 文

4.5.1 if文

if (*expr*) *statement1* else *statement2*

は、アクティブなすべての仮想プロセッサが *expr* を評価する。その結果が 0 以外となった仮想プロセッサのみが *statement1* を実行し、0 となった仮想プロセッサのみが *statement2* を実行する。if 文の本体が終了すると、アクティビティは if 文が実行される前の状態に戻る。ただし、

- break;
- continue;
- return [*expr*];

を実行した仮想プロセッサは、その break 文、continue 文に対応する文、あるいは return 文を

含む関数の実行を他のすべての仮想プロセッサが終了するまで、インアクティブになる。return 文に *expr* が与えられているときは、インアクティブになる前にそれを評価し、結果がその仮想プロセッサに対する関数値となる。

4.5.2 all 文

all *statement*

は、カレントフィールドを構成するすべての仮想プロセッサをアクティブにして *statement* を実行させる。すべての仮想プロセッサが *statement* の実行を終了すると、仮想プロセッサのアクティビティは all 文実行前の状態に戻る。

4.5.3 ループ文

- for 文

  for ([*expr1*];[*expr2*];[*expr3*]) *statement*

- while 文

  while (*expr*) *statement*

- do-while 文

  do *statement* while (*expr*);

これらのループ文は、その時点でアクティブな個々の仮想プロセッサが、*expr* を満たす間、*statement* を繰り返し実行する。*expr* を評価して結果が 0 となった仮想プロセッサは、他のすべての仮想プロセッサがループの実行を終了するまでインアクティブとなる。これを *expr* の値が 0 以外となる仮想プロセッサが存在するかぎり繰り返す。

すべての仮想プロセッサがループの実行を終了すると、アクティビティはループに入る前の状態に戻る。ただし、ループ中の return 文を実行した仮想プロセッサはすべてのプロセッサがそのループを含む関数の実行を終了するまでインアクティブとなる。

4.6 関数

関数の呼出しは、アクティブな全仮想プロセッサが同時に行う。同時に呼び出される関数とその実引数の個数はアクティブはすべての仮想プロセッサで同じでなければならないが、その実引数の値は仮想プロセッサごとに異なってよい。関数呼び出しの構文はＣ言語ど同じであり、テキスト上で関数呼び出しが許される場所もＣ言語と同じである。

4.6.1 関数呼び出し時のカレントフィールド

関数が呼び出された時点のカレントフィールドが *F* であるとき、その関数は「*F* から呼び出された」という。ＮＣＸにおける関数は、それを呼び出すことができるフィールドによって、３つに分類される。すなわち、

1. 特定のフィールドのみから呼び出すことのできる関数。
2. 任意のフィールドから呼び出すことのできる関数。
3. あるベースフィールド上のフィールドのみから呼び出すことのできる関数。

ＮＣＸでは、関数の定義時にフィールド限定子 (field specializer) とよばれるもので、定義される関数が上の３つのいずれに属するかを決定する。

4.6.2 関数ポインタ

Ｃ言語と同様に、関数名を式として与えることにより、その関数へのポインタを得ることができる。同一の関数への関数ポインタは、常に同じ値であり、仮想プロセッサごとに異なることはない。

ＮＣＸは SIMD モデルを採用しており、同時に２つ以上の関数を呼び出すことはできない。関数ポインタを使った関数呼び出しの式

$$expr(arg, ..., arg)$$

において、アクティブな仮想プロセッサによって *expr* の値が異なる場合は、そのうちの任意の１つが選択されて、そのポインタによって指される関数が呼び出される。その他の値は無視され、それらによって指される関数は呼び出されない。

各仮想プロセッサが関数ポインタによって指定した関数をすべて呼び出すためには、繰り返し構文を使って、関数を順次呼び出せばよい。

4.7　まとめ

ＮＣＸは SIMD モデルを採用するため、数値計算分野に向いている。また、命令レベルで同期しているため、明示的に同期を記述する必要がない。しかし、データの流れが複雑なため、結合網が複雑になると、効率的なプログラミングが困難になる。

参考文献

[1] 並列記号処理. 柴山潔. コロナ社, 1991.

[2] 計算機システム特論第２ 講義資料.

[3] 超並列Ｃ言語ＮＣＸ言語仕様書 (Version 3). 1993.

[4] KLIC 講習会テキスト -KL1 言語編-. 1993.

[5] 勉強会資料. 津邑公暁. 1995.

[6] 輪講資料. 小川敏一. 1996.

[7] 輪講資料. 重田大助. 1997.