解　説

並列処理とソフトウェア*

村　岡　洋　一**

1. はじめに

並列計算機とひとくちに言うが，Flynnによればこれは次の2つのタイプに分類できる[1]．

（1） SIS-MDS

Single instruction stream-Multiple data stream.

ILLIAC IV に代表されるアレイ計算機．

（2） MIS-MDS

Multiple instruction stream-Multiple data stream.

通常のマルチ・プロセッサ．

この外に，SIS-SDS (Single instruction stream-Single data stream) タイプの CPU でも，CDC 6600 のように内部に複数の演算器があって幾つかの命令がオーバラップして実行されるものがあり，これは広義の並列計算機の範ちゅうに入れて良いと思われる．

このように並列計算機にも各種あるので，ソフトウェアの生産性との関係も一意に議論するのは難しい．本稿ではまず SIS-MDS タイプの並列計算機を対象として，ソフトウェア（特にプログラミング言語）との関係を調べてみる．SIS-MDS タイプの計算機の主用途は科学技術計算であり，処理速度の飛躍的な向上が主眼である．このような計算機は従来は特殊用途機とされ，その使い易さなどは処理速度向上の前に犠牲にされることが多かった．しかし近年これらの高速機を一般利用者の手の届くものとしようという努力が，とみに盛んになって来た．この努力について第2章で説明し，その問題点の指摘を計る．

第3章では MIS-MDS タイプと SIS-SDS タイプの並列計算機をとりあげる．これからの汎用計算機の進む方向は，これらの2つのタイプのうちのどちらかと考えられるが，ここではソフトウェアの生産性との関係から両者の比較を試みよう．

ソフトウェアの生産性自身が well-defined な言葉でもないし，並列処理もあまり良く確立された技術とはいえない．従って本稿の内容も多分に筆者の独善的な意見に基づく「お話し」的なものとなってしまうことをお許し頂きたい．

2. SIS-MDS タイプ計算機のプログラム

SIS-MDS タイプの計算機の命令は，1命令で複数のオペランドの演算を行う．従って当然のことながら，従来の FORTRAN 等の言語で書かれたプログラムを直接マッピングすることは難しい．このためには並列処理を記述できる並列処理言語が各種考えられている．ILLIAC IV 用に開発された TRANQUIL[2] 等がその一例である．構造を持ったデータに対する均一な処理を並列に行う場合の記述には，比較的自然な方法を考えることは容易であり，数学で使われている記法の延長として受け入れられ易い．単純な例として行列の乗算をとり上げよう．2つの $n\times n$ 行列の乗算は数学では

$$c_{ij}=\sum_{k=1}^{n} a_{ik}b_{kj}$$

$$(i=1,2,\cdots,n;\ j=1,2,\cdots,n)$$

と書けるし，TRANQUIL では，

```
for (I, J) sim ((1, 2, ···, n)×(1, 2, ···, n))
for K seq (1, 2, ···, n)
    C[I, J]=A[I, K]*B[K, J];
```

と書ける．

これに対していわゆる数学の公式からはみだした部分の並列処理記述は，自然な方法を考えることは簡単ではない．これも良く出される例として，代入文の tree を使った並列処理をとり上げて見よう．代入文

$$a=b+c+d+e+f+g+h+i$$

は**図-1**（次頁参照）に示すような方法で並列処理できる[2]．この処理を自然な形でどのように記述したら良いであろうか．これまで提案されている並列処理記述

* Parallel Processing and Software by Yoichi MURAOKA (N. T. T., Yokosuka Electrical Communication Laboratory)
** 日本電信電話公社横須賀電気通信研究所データ通信研究部

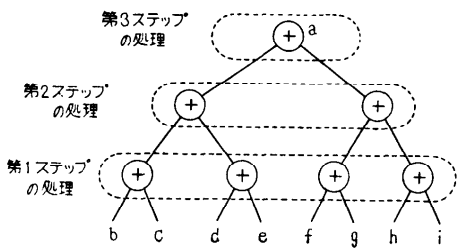


図-1 代入文の並列処理の例

言語で，この問題を解決している例は無い．強いて捜せば，やはり TRANQUIL の **sim** 文が使えないこともない．次に例を示す．

```
sim begin
    t1=b+c;
    t2=d+e;
    t3=f+g;
    t4=h+i;
end;
 sim begin
    t5=t1+t2;
    t6=t3+t4;
end;
a=t5+t6;
```

元の単純な代入文が複雑なブロック構造に変わってしまったことに注意して欲しい．

以上の説明では，均一なデータ構造上の均一な並列処理の記述は比較的単純かつ自然に行えるかのような印象を与えている．次に一見単純であるが，並列処理の記述が難しい例として Fast Fourier Transform (FFT) を**図-2** に示す．同図では初期値 $F[1,i]$ が3回の並列繰返し処理（ステップ）によって，$F[4,i]$ に変換されるさまを示している．各交点では，交点への2つの入力値が適当な重みをつけられて加算される．1つの繰返しごとに並列処理できることは明らかで，全部で3回の並列処理ステップが必要である．この処理を普通の計算機上で行うために，ALGOL 風言語でプログラムした例が**図-3** であるが，特にインデックス処理が複雑になっていることに注目されたい．ここには例を示さないが，このプログラムを例えば TRANQUIL で作成する場合，インデックス処理がより複雑になり手に負えなくなるであろうことは想像に難くない．

並列処理計算機のプログラムを行うもう一つの方法は，従来のプログラム言語（例えば FORTRAN）で書かれたプログラム中から並列処理性を発見するコンパイラ（またはアナライザ）を作成することである．既に実例もありその見通しも悪くはない[2]．しかしその結果得られるオブジェクト・プログラムの性能はいかがであろうか．

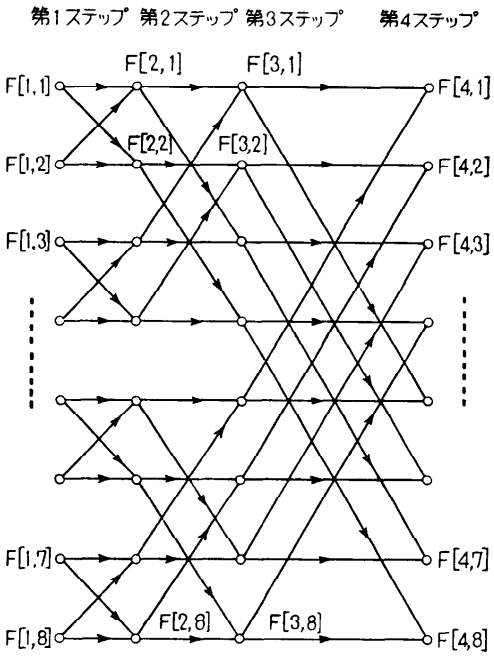


図-2 FFT 処理の例

```
for iter := 1 to 3 do
for i := 1 to 8 do
begin
    it1 :=2**(iter −1);
    it2 := i−1;
    it3 := (it2 mod 2);
    it4 := it2−2*it3;
    if iter=2 then
       it4 := it3−(it3 mod 2)*2;
    if iter=3 then it4 :=(it3 mod 2);
    if it4=1 then it1 := −it1;
    F(iter, i) := function (F(iter, i),
    F(iter, i+it1));
end;
```

図-3 FFT のプログラム例

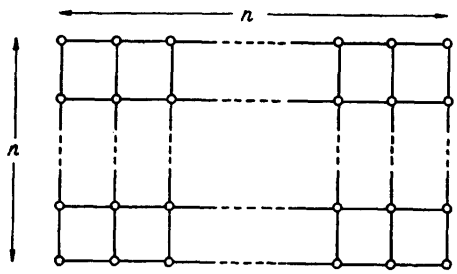
簡単な例として**図-4**（次頁参照）に Laplace の方程式

$$\frac{\partial^2 W}{\partial x^2}+\frac{\partial^2 W}{\partial y^2}=0$$

を，2次元平面上で解く場合の例を示した．平面を $n \times n$ のメッシュに分割して，各メッシュ上で差分近似

$$W_{i,j}=(W_{i+1,j}+W_{i-1,j}+W_{i,j+1}+W_{i,j-1})/4$$

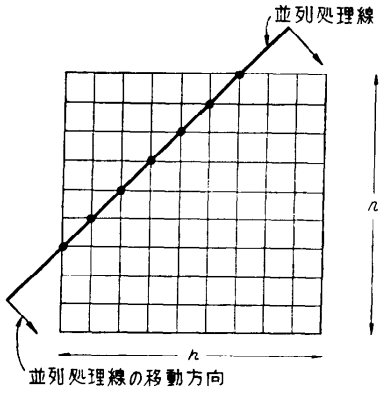
を求めれば良い．これを普通の計算機向きに ALGOL 風言語でプログラムしたのが図-4(b)である．このプ



(a)

repeat: **for** i=2 **to** n−1 **do**
　　for j=2 **to** n−1 **do**
　　$W(i,j)=(W(i-1,j)+W(i+1,j)$
　　$+W(i,j-1)+W(i,j+1))/4.0$;
　　if not converge **then go to** repeat;

(b)　偏微分方程式プログラムの例



(c)

repeat: **for** (i,j) **sim** $((2,3,\cdots,n-1)\times(2,3,\cdots,n-1)$
　　$W(i,j)=((W(i-1,j)+W(i+1,j)$
　　$+W(i,j-1)+W(i,j+1)/4.0$;
　　if not converge **then go to** repeat;

(d)　並列処理アルゴリズム

図-4　偏微分方程式の並列処理

ログラムを解析して並列処理性を見つけた結果が，同図(c)である．1つの並列処理線上の全メッシュ・ポイントについて上記近似処理を並列に行える[2]．並列処理線を矢印の方向に1ステップずつ移動させて，各ステップ毎に並列処理を行えば良い．この方法では高々n個のメッシュ・ポイントについてしか並列処理できず，平面全体について処理するのに約$2n$ステップかかる．しかしもし偏微分方程式の性格を知っていれば，全メッシュ・ポイントについて差分近似を同時に行うことができる．このプログラムが図-4(d)で，全メッシュ・ポイントの値が1ステップで更新されている．図(c)と(d)の違いはこれだけではない．一般的に(d)の方法が(c)の方法よりも，近似の収束速度自身も速い．即ち方式(d)は(c)よりも2つの意味で優れている*．

並列処理記述言語によるプログラムには，その複雑さから限界があることを初めに示唆した．次に従来言語プログラムを解析して，その中から，並列性を発見する方法を提案したが，この方法は性能的に問題があり得ることを指摘した．即ちどちらの方法も一長一短がある．

プログラムを解析して並列性を発見することによって，またその他の新しい問題も生じる．この問題とは，計算誤差の拡大の可能性である．プログラム解析の結果は，プログラム内の演算実行順序の変更を要求する．この例が図-1 の代入文の並列計算で，直列計算機上では，

$$a=((((((b+c)+d)+e)+f)+g)+h)+i$$

と計算されるのが，並列計算機上では

$$a=((b+c)+(d+e))+((f+g)+(h+i))$$

という計算になってしまう．このように計算順序を変えることによって，思いもかけない誤差が生じる可能性があるのは既知の事実である．

以上に述べた事項を総括すると，どうも並列計算機向き言語としては，いわゆる問題向き言語 problem-oriented language が好ましいようである．ソフトウェアの生産性向上等のために，structured programming だとか，go to-less programming だとか騒がれている．しかしこれは計算機を道具として使いたい人，例えば科学技術者には無縁のように思われる．勿論計算のアルゴリズムを論理的かつ簡単に記述できる方法を見つけ出すことは重要である．特にシステム・プログラムの領域等においては，不可欠とさえいえよう．しかしその反面において，計算機の本当の利用者に対してのサービスがなおざりになっているように思うのは，単に筆者一人の偏見であろうか．

実際の演算を並列に実行するか直列に実行するかは，ある意味では計算機の側の勝手な理屈であって，利用者はこれに左右されるべきではなかろう．その意味では Iverson の APL は理想に近いといえるかも知れないが，残念なことにあまりに複雑な記法ゆえにその利点が相殺されている．今後必要なのは，科学技術計算向きの problem oriented language であろう．偏微分方程式の処理等のために作られた例が2,3見られ

* 実際には，偏微分方程式の性格によっては収束性等で問題が生じることもあるが，ここではその可能性は無視した．

るのみで，まだまだこの分野は未開発である[3]．

3. 計算機アーキテクチャとソフトウェア

この章では並列計算機の解釈を広げて，マルチ・プロセッサやパイプライン制御方式についても考慮して見る．汎用計算機は極論すると，比較的単純な論理・演算回路をマイクロ・プログラム化している小型機と，複雑な論理・演算回路に先廻り制御やパイプライン制御（命令解釈から実行部のパイプライン化）を活用している大型機に2分されると思われる．大型機ではさらに複数の演算回路を設けて，CPU 内部で一度に幾つもの命令が多重に実行されるようなものもある．このような複雑なCPUを効率良く使うためには，高度なプログラムの最適化技術が要求される．例えば複数の演算回路を効率良く使うためには，図-1 に例示したように演算レベルでの並列性を見つける必要がある[2]．またパイプライン化された制御回路を効率良く使うためにはハードウェアの性質を良く理解して，例えばジャンプ命令実行時には2つある後続命令のうちのどちらがハードウェアで先き取りされるかを見極めて，ジャンプ頻度の高い方を先き取りされるようにプログラムする等の配慮も必要になってくる．

従来の CPU 設計の思想は，これらの最適化はコンパイラが十分にやってくれるものと「信じて」，ハードウェアの性能を向上させていた．しかしもし今後さらにこの傾向が続いて，CPU 内が高度にパイプライン化され，数十命令にも及ぶような先廻り制御にまで発展すると仮定すると，ソフトウェアによる最適化技術はそこまで追随できるであろうか．特に前章に述べたように，利用者の応用毎に問題向き言語を開発して行かなければならないとすると，この問題は無視できないものとなって行こう．

上記に述べた問題発生の一因は，1台の CPU の性能を極端に向上させることにある．Grosch の法則が成立する世界においては，1台の CPU 当りの性能を高くする程価格性能比が良くなった．しかし，最近のように LSI 製造技術が発達して，1チップ CPU 等が発売される時代には，Grosch の法則も見なおされてしかるべきではなかろうか．もし低性能の CPU でも，高性能の CPU と同等またはそれ以上の価格性能比で製造できるのであれば，低性能 CPU を多数台使うマルチ・プロセッサ・システムの方が，高性能 CPU を極く少数台使う従来構成よりも魅力あるものとなり得る．このシステムでは，CPU の構成は単純なので，最適化にはそれ程高度の技術は必要とされない．CPU 台数増大の欠点の一つはメモリ・アクセス競合であるが，例えば PRIME 等のように CPU 毎に私用メモリを持たせる方式ならばこれも解決できる．

要は，従来のような時分割処理（1台の CPU で多プログラムが多重処理される）の考え方から，空間分割処理（1台の CPU は1プログラムが独占する・．マルチ・プログラムは異なる CPU 上でなされる）の可能性も考えるべきではないかということである．空間分割処理では，時分割処理におけるような割込みやタスク・ディスパッチの処理の複雑さも無い．これらは1台の CPU を多重使用するために必要となって来たものである．

4. おわりに

以上まとまりのない話しで恐縮であるが，言わんとしたところは問題向き言語の必要性と，プログラムしやすいシステム・アーキテクチャの必要性の2点につきる．特に後者については，部品技術の進歩ともあいまって，再考しても良い時代になりつつあるのではないだろうか．

参考文献

1) Enslow, P. H.: Multiprocessors and Parallel Processing, Wiley-Interscience Publication, New York (1974)
2) 村岡：並列処理概論(1)～(3)，情報処理，Vol. 16, No. 1～3 (1975)
3) Gardenas, A. F.: A Problem Oriented Language and a Translator for Partial Differential Equations, Ph. D. Thesis, UCLA Report, No. 68-62 (1968)

（昭和50年6月23日受付）
（昭和50年7月14日再受付）