計算機アーキテクチャ 44－2
（1982. 2. 24）

並列型シミュレーター向アーキテクチャ

越田一郎　中川裕志　（横浜国立大学　工学部）

1.はじめに

従来，待ち行列系解析等の離散型シミュレーションは，大型計算機を用いて行なわれてきた。しかし，大規模システムをシミュレートする際には多大な時間を必要とした。

他方では，マイクロプロセッサの進歩により，マルチプロセッサシステムを構成するのが容易になってきている。そこで，マルチプロセッサによるシミュレーターの研究が行なわれている。[(1)~(3)]

しかし，シミュレーターの場合，シミュレーションモデル上の時間的順序に従ってシミュレーションを実行しなければならない。この制約条件のためプロセッサ間においてデッドロックの可能性を生じる。

これを回避するための一方法として「nullメッセージ」を用いる方法がある。[(3)] この方法に関し，筆者らは，そこで用いられる予測方式，およびシミュレーションモデルによっては，大幅に性能が悪化することを示した。[(4)]

本報告では，シミュレーションモデル上のデータを処理する部分とプロセス間デッドロックを回避するためのコントロールを行なう部分を分離したアーキテクチャを示す。負荷の分散により，特にワーストケースにおける性能が向上すると期待できる。また，これを述べる前に，データフロー計算機との関係について簡単に触れる。

2.データフロー計算機との関係

2.1 離散型シミュレーターの並列化

データフロー計算機との関係を述べる前に，シミュレーターをマルチプロセッサ化する方法を簡単に述べることにする。

シミュレーションモデルは，プロセスを単位にして考える。プロセス間の情報交換はHoareが提案したCSPに類似した形で，メッセージを送ることによって行なわれるとする。シミュレーションモデル上で，プロセスは並列に動作しているのだから，それらを別々のプロセッサで実行することにすれば，負荷が分散するためシミュレーターの性能が向上すると考えられる。

この際，1個のプロセッサあたり，何個のプロセスを割りあてるかということが問題となるが，今のところ，概念的に最も簡単な，1プロセッサあたり1プロセスで考えることにする。実際，プロセスの概念は種々のレベルで考えることができるから，このように見なしても問題ない。

次に，時間的順序を守るための制限条件について述べる。なお，モデル内の仮想的な時間とシミュレーターが動作している現実の時刻を区別するため今後，前者をLT（Logical Time）後者をST（Simulator Time）と呼ぶことにする。

図1(a)のような場合はメッセージを取り出せない。なぜなら，下の入力ラインに到着するメッセージのLTが不明であるから。結局，同図(b)のようにすべての入力ラインにメッセージが揃ってはじめて1つのメッセージを取り

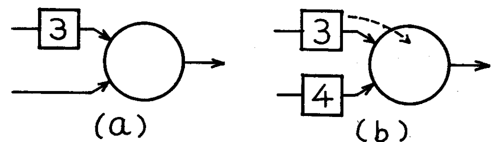


図1 時刻同期の規則

出すことができる。

しかし，この規則に従うと，シミュレーションモデル内に分岐，フィードバックループ等が存在する場合，デッドロックを生じることが知られている[3][4]。このデッドロックをどのように回避するかが，マルチプロセッサシステムでシミュレーションを行なう場合の問題点である。

2.2 データフロー計算機との相違点

前節で述べた，「すべての入力ラインにメッセージが揃うまで処理を開始しない」という規則は，データフロー計算機におけるオペレーターの実行規則と類似しているように見える。しかし，次に示す4点において，シミュレーターはデータフロー計算機と異なっている。

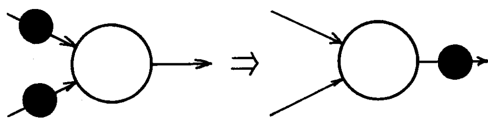
(1) 単一代入規則
(2) オペレーションの副作用
(3) オペレーションが必要とするトークンの数
(4) トークンの順序関係

(1) この規則のため，データフロー計算機では，シミュレーターで最大の問題となるフィードバックループを免れている。

(2) シミュレーターで考えているプロセスは状態変数―たとえばそのプロセスの時刻，待ち行列長など―を持っており，処理の結果はそれらに依存している。逆に，シミュレーションを行なって得たいのは，副作用の結果であると言える。

(3) データフロー計算機では，入力ラインに揃ったすべてのデータを使用して処理を行なう。（図2(a)）シミュレーターの場合は，揃ったメッセージのうち1個だけを使用する。（図2(b)）シミュレーターでは，処理に必要だからメッセージを待っているのではなく，単に時刻同期の必要上待っているにすぎない。

(4) 結局，最大の問題は，シミュレーターのメッセージがLTという順序関係を持っていて，それに従って処理をしていかなければならない，という点にある。


(a) データフロー計算機

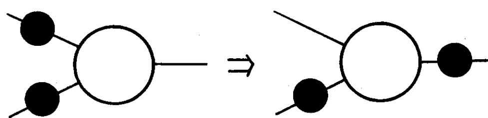

(b) シミュレーター

図2 データが揃った後の処理

3. 並列シミュレーターのアーキテクチャ

3.1 ハードウェア構成

シミュレーターの構成要素は3種類に大別することができる。

(1) メッセージを処理する「プロセッサ」
(2) プロセッサ間通信を行なう「ネットワーク」
(3) プロセッサの初期化などを行なう「コントローラー」

(1)および(2)は，データメッセージ用とデッドロック回避用に各々1組ずつ存在する。この全体を図3に示す。

データプロセッサ（DP）とデータネットワークがシミュレーションモデルを構成する。すなわち，DPがシミュレーションモデルのプロセスで行なわれる処理を実行し，データネットワークを介してプロセス間のメッセージ通信を行なう。DPが行なう他の処理は入力ラインに到着しているメッセージから自分のシミュレートしているプロセスの状態を判断し，それが変化した

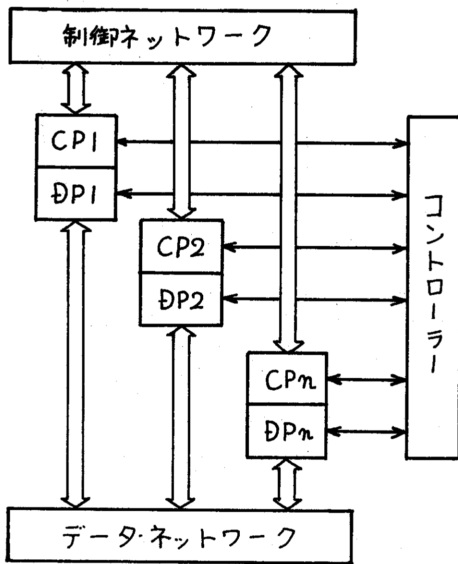


図3　シミュレーターの構成

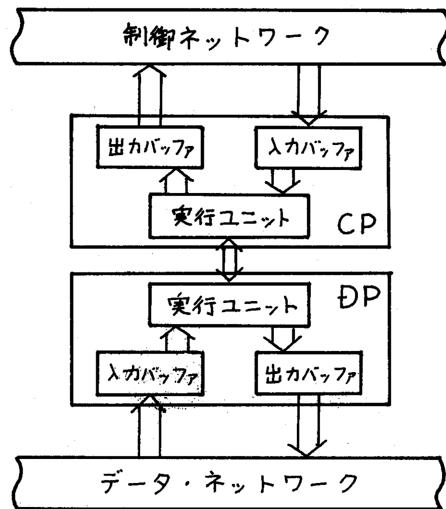


図4　CPとDPの構成

らコントロールプロセッサ（CP）に教えることである。これについては3.2で詳しく述べることにする。

CPとコントロールネットワークはデッドロック回避のために処理を行なう。その主な役割は，DPがシミュレートしているプロセスの状態を他のCPに通知することと，デッドロックを検出したならば，それを解除するように適切なDPを駆動することである。

CP，DPの内部構成はほとんど等しい。それを図4に示す。CPとDPは1対1で極めて密に結合している。そのため，この間の通信オーバーヘッドは，プロセス間のそれに比べて無視できる。入力／出力バッファはネットワークとのインターフェースを行ない実行ユニットが実際の処理を行なう。各バッファは判断機能を備えており，実行ユニットが必要とするデータのみを渡す。

ネットワーク自体の構成は検討中であるが，後で述べるように同時に複数のデータを送ることができる必要がある。また，ハードウェア上も2つのネットワークは分離している必要がある。今のところ，リングネットワークを念頭においている。

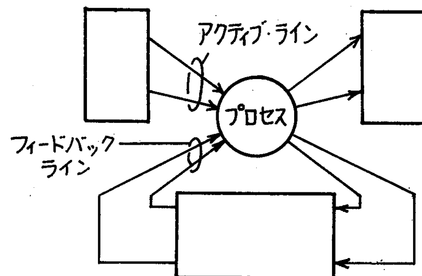


図5　入力ラインの分類

3.2デッドロック回避

ここでは，各DPに割りあてられているプロセスを単位として考え，各プロセスの入力ラインを2種類に分類する（図5）そのプロセスを含むフィードバックループからのラインをフィードバックライン，それ以外の入力ラインをアクティブラインと呼ぶ。この際，フィードバックループ以外の部分では少なくともこのプロセスが原因となるようなデッドロックは生じないと仮定する。したがって，アクティブライン

とは，待っていれば必ずデータメッセージが送られてくる入力ラインであると言える。

また，各プロセスの状態も２種類に大別する。

(1)アクティブ：すべてのアクティブラインにメッセージが到着していないか，データメッセージをDPが処理している状態。

(2)ノン・アクティブ：上記以外，すなわち，すべてのアクティブラインにメッセージが到着しているが，それを取り出して実行することが不可能な状態。

時刻同期とデッドロック回避に関する規則は次の２つである。

(1)すべての入力ラインにメッセージが到着したプロセスは実行してよい。(時刻同期)

(2)フィードバックループ内にあるすべてのプロセスがノン・アクティブになった場合は，最もLTが小さなメッセージから処理を行なう。(デッドロック回避)

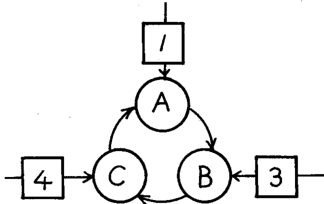


図6　デッドロックの回避

２番目の規則が適用されるのは図6のような場合である。この図ではプロセスAが処理を行なう。この場合，1番目の規則だけでは，どのプロセスもデータを処理できないことに注意する必要がある。

次に，DPとCPについて詳述する

i) DP

DPはシミュレーションモデル中で実際に送られるメッセージを処理する

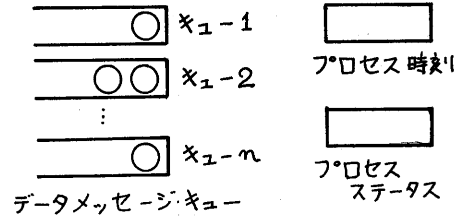


図7　DPのデータ構造

コード	ステータス	時刻(LT)

コード：ステータス変化／処理終了／DP起動

ステータス：アクティブ／ノン・アクティブ／実行可能

図8　データメッセージ

```
cycle
   receive data-message M:
      enqueue(M);
      if process status changes then
         send status-change-message
         to CP
      fi
      ∇
   receive trigger-message:
      dequeue(M);
      execute M;
      send data-message to other DP;
      send end-message to CP
end cycle
```

図9　DPの動作

プロセッサであり，その機能は，

1)DP間でメッセージの通信を行ない，受信したメッセージを入力キューに入れる。

2)キューの状態を監視し，プロセスの状態が変化したらCPに知らせる。

3)メッセージをシミュレーションモデルに従って処理する。

DPが使用するデータ構造を図7にCPとの通信に使うメッセージの形式を図8に示す。

前に，プロセスの状態を２種類に分類すると書いたが，さらに

実行可能，実行中

の2状態を付け加える。実行可能というのは，すべての入力にメッセージが揃った状態，実行中はDPがメッセージを実際に処理している状態である。

DPが行なう処理の概要を図9に示す。ここで *cycle − end cycle* はいわゆる guarded region である。DPがCPに状態変化メッセージを送るのはプロセスの状態がノン・アクティブになった時と，実行可能になった時である。

ii) CP

CPはフィードバックループ内にある各プロセスの状態を維持し，適切な時にDPのメッセージ処理を起動する。CP間の通信に用いられるメッセージをデータメッセージと区別してタプルと呼ぶことにする。CPの機能を整理すると次の3種類になる。

1) タプルを受信し，各プロセスの状態を変更する。

2) DPから状態変化のメッセージを受けると，タプルを発し，他のCPにプロセスの状態を伝える。

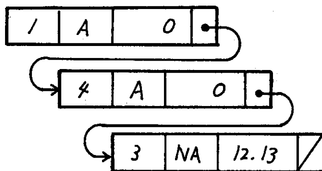
3) DPがメッセージを処理してよい条件が整ったらDPを起動する。

他プロセスの状態は，概念的には図10(a)に示すようなプロセス・ステータス表に記録しておくが，実用上LT順にソートしておくのが望ましいので，同図(b)のようなリストにすることにする。またフィードバック中にアクティブプロセスが何個あるかを知るために変数を1個用意し，APNと名づける。

CP間の通信に使用するタプルを図11に示す。この中の時刻は，ノンアクティブタプルを送る際，そのプロセスの入力ラインに到着しているメッセージの中で最小のLTを他のCPに知らせるために使用する。また，フラグはそのタプルと共にデータメッセージが出力されたことを示すためのものである。たとえば，図12のように DP_1 がメッセージ処理を終え，ノンアクティブになった場合を考える。DPがメッセージを出力すると共にCPはノンアクティブタプルを出力する。DP_2 はメッセージを受信すると実行可能となり，CP_2 はアクティブタプルを発する。このとき，DPよりCPの動作の方が速いとノン・アクティブタプルの方が先行することになり，CPがループの状態を正しく把握できなくなる。したがって，フラグがonの時はCPはそ

プロセスNo.	ステータス	時刻
1	アクティブ	0
3	ノン・アクティブ	12.13
4	アクティブ	0
⋮	⋮	⋮

(a) プロセス・ステータス表



(b) プロセス・ステータス・リスト

図10 プロセス・ステータスの記録

出力プロセス番号	ステータス	フラグ	時刻

ステータス { アクティブ / ノン・アクティブ }

図11 タプル

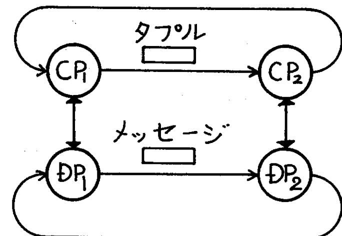


図12 フラグを用いる場合

```
cycle
   receive own tuple T:
      if DP is not executing &
         T is non active
      then
         update process-status-list,APN
      fi
      ▽
   receive other CP's tuple T:
      update process-status-list,APN;
      send T to next CP
      ▽
   receive status-change-message:
      if status is executable then
         DP is executable
      else
         send non-active-tuple to
         next CP
      fi
      ▽
   receive end-message:
      update process-status-list,APN;
      if status is non-active then
         send non-active-tuple to
         next CP
      fi
      ▽
   DP is executable:
      send trigger-message to DP;
      send active-tuple to next CP;
      update process status-list,APN
end cycle
```

図13　CPの動作

のタプルをただちに中継せず，DPにメッセージが到着したことを確認してからタプルを中継する必要がある。

図13に今まで述べてきたCPの動作をまとめて示す。記法はDPの場合と同じである。また，プロセスの状態遷移図を図14に示す。

4. CPのオーバーヘッド

フィードバックループ内の処理を細分化してDPの数を増やせば，DPの負荷は減少する。しかし，CPで行な

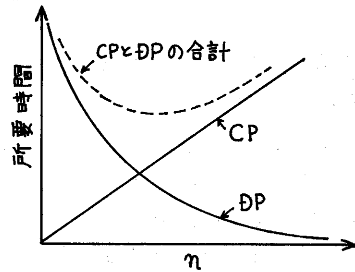


図15　最適点の存在

うべき処理は増加する。このCPのオーバーヘッドとループの分割数nとの関係について考察する。

各CPは，自分も含めてn コのCPの状態を知る必要がある。したがってタプルの数は全体でn^2に，個々のCPが処理すべきタプルはnに比例する。

このため，1度に1タプルの通信しか行なえないようなネットワークでは通信コストがn^2に比例するため，大きなボトルネックとなる。また，1度にnタプルが送れるようなネットワークならば通信コストはnに比例する。

また，CPで1タプルを処理するのに必要な時間は，プロセスステータスリストの維持に要する時間に依存する。この処理にFranta, Maly の提案したTLアルゴリズムを用いれば，nが大きい場合の処理時間はほぼ一定となる。nが小さい時は，このコストが他の処理と比べて問題にならないと考えられる。結局はCPの処理時間は一定と考えてよい。

したがって，CPの行なう，デッド

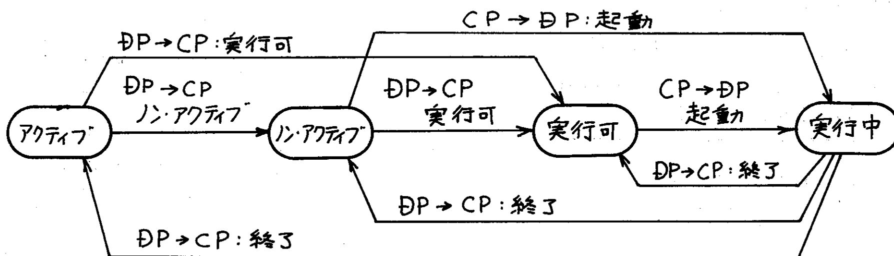


図14　プロセスの状態遷移図

ロック回避によるオーバーヘッドは，nに比例すると言える。

フィードバックプロセスの処理をnコのĐPに分割したのだから，理想的にはĐPの処理時間は$1/n$になる。したがって，あるフィードバックをnコに分割する場合，分割数を多くすればよいというわけではなく，図15のように最適な分割数が存在する。

5. シミュレーションによる評価

これまで述べてきた方法に基づくシミュレーターの性能が，シミュレーションモデル，CPとĐPの処理速度の違いによって，どのように変化するかを調べるために，シミュレーションを行なった。

使用したシミュレーションモデルを図16(a)に示す。サービス率μの指数サービスを行なう2個のサーバーに，到着率λで客がポアソン到着する。サービスが終了すると，確率pで客は外界に出されるが，その他の客は，もう1個のサーバーに渡される。

シミュレーターの構成は同図(b)に示す。各サーバーおよび客の発生源に1個ずつĐPを割り当てる。ĐP$_{2,3}$はフィードバックループを形成しているので，各々にCPを割りあてる。

本来，ĐP$_{1,4}$はĐP$_{2,3}$のフィードバックとは独立であるが，そのようにシミュレーションを行なうとĐP$_{1,4}$の効率が100%となってしまい，シミュレーションの目的にそぐわない。そこでĐP$_{1,4}$はĐP$_{2,3}$からメッセージの要求があった時にメッセージを送ることにした。

各ĐP，CPが1メッセージあるいは1タプルを処理することを1単位と考える。各プロセッサの実行時間（ST）は，この1単位の処理について定めている。

このようなシミュレーターの場合，CPとĐPの実行時間比R

$$R=\frac{\text{ĐPの実行時間}}{\text{CPの実行時間}}$$

と外界に客を出す確率pによってシミュレーターの性能が変動する。そこでこの2者をパラメタとして変化させ，シミュレーションを行なった。その結果を図17に示す。この図で横軸はp，縦軸は，シミュレーションに要した時間を全ĐPの実行時間で割ったものである。

この図を見ると，pが小，すなわちフィードバックループ内にメッセージが増加し，ĐP$_{2,3}$が実行可能の状態に

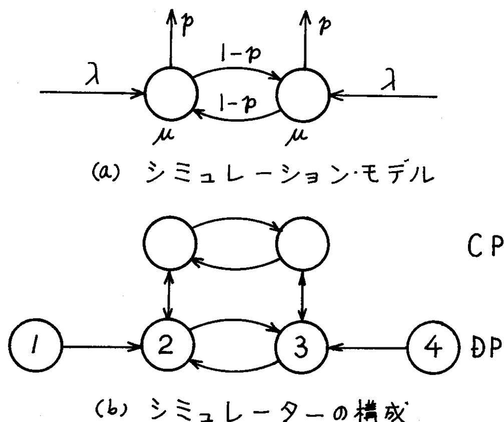


図16　モデルとシミュレーター

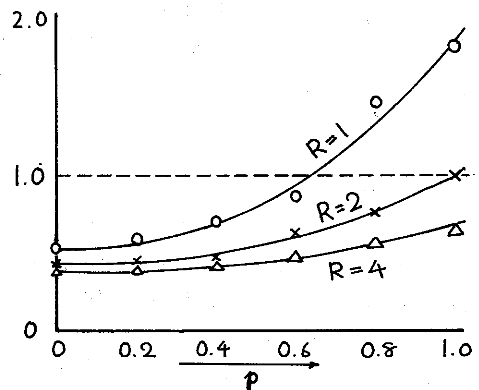


図17 シミュレーション結果

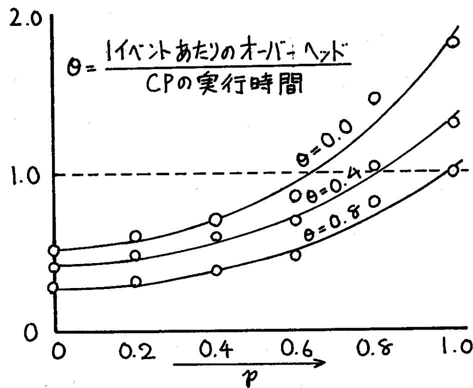


図18 オーバーヘッドを考慮した場合

なる場合が多くなると性能が向上しているのがわかる。ただし，4台のDPを使用したにもかかわらず，0.4程度にしかならないが，これはDP 1,4がDP 2,3に比べて負荷が小さく，待ち時間が多いためである。

またCPの処理時間がDPのそれに比べて長くなると性能はかなり悪化する。ただし，図17の場合，単一プロセッサにおけるオーバーヘッドを考慮していない。これを考慮して，図17でR＝1の場合をプロットしたのが図18である。ここでθは

$$\theta = \frac{1イベントあたりのオーバーヘッド}{CPの実行時間}$$

である。

単一プロセッサのオーバーヘッドがCPの実行時間と同程度ならば，最悪の場合でも本方式の方がよい結果を得ることがわかる。

6. おわりに

本報告では，コントロールネットワークとデータネットワークを分離し，負荷を分散させたマルチプロセッサ・シミュレーターの構成，デッドロック回避法について主に述べた。今後の予定としては，より大規模なモデルに対しフィードバックの分割数nが変動した場合のシミュレーションなどを行ない検討を進めていく予定である。

参考文献

(1) 長谷川他，"Queueing システム・シミュレーションにおけるQSVプロセッサの時刻同期方式"，1979年，信学会大会

(2) 松本他，"待行列網シミュレータHASS-QNのソフトウェア設計"，1981年3月，情報処理学会大会，4E-2

(3) K. Chandy, J. Misra, "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs", IEEE Trans. SE-5, No.5 Sept, 1979

(4) 中川，越田，"並列型シミュレーターの性能評価について"，情報処理学会，分散処理システム研究会，No.10，1981年9月

(5) W. R. Franta, K. Maly, "An Efficient Data Structure for the Simulation Event Set", CACM, vol.20, No.8, Aug., 1977