並列型シミュレーターの性能評価について

越田一郎　中川裕志　（横浜国立大学　工学部）

I. はじめに

従来，離散型シミュレーションは，汎用計算機上でGPSS，SIMSCRIPT等のシミュレーション言語を用いて行なわれてきた。しかし，大規模システムのシミュレーションを実行する際，イベントの管理，統計処理等に多大な時間が必要であった。

そのため，近年価格の低下，性能の向上が著しいマイクロプロセッサを多数用いたシミュレーターの研究が数多く行なわれている。[1][2][3] このような並列処理システムにおいては，シミュレーション実行にあたってイベントをその発生時刻順に取り出して処理しなければならないという制約条件がある。この制約条件のため，イベントの処理を行なう複数のプロセス間でデッドロックを生ずる可能性がある。

デッドロックを回避するには，各プロセスの挙動を予測することが必要となる[3]が，本研究では3種類の予測方式を示し，各方式を用いて同一モデルのシミュレーションを行なうことにより各予測方式の性能評価を試みている。

II. マルチプロセッサ化の方法

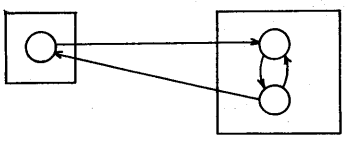
シミュレーションされるシステムを相互に通信するプロセスの集合としてモデル化する。モデル内の各プロセスは，現実のシステムにおいて並列に実行される。したがって，シミュレーター上でも，ハードウェアに制限が存在しなければ，各プロセスを並列に実行することが可能となる。すなわち，各プロセスを別個のプロセッサに割当てマルチプロセッサシステムとして並列に実行する。すると，1つのプロセッサに対する負荷が減少するため，より高速なシミュレーションが期待できる。

従来は，プロセッサが高価であったので，このような方式は不可能だった。しかし，最近，マイクロプロセッサの性能向上，価格低下が著しいため，このような方式も実現可能となった。

この方式では，どのようにプロセスをプロセッサに割り当てるかが重要な問題となる。ここでは，簡単のため，1プロセスに1プロセッサを割当てるすなわち，プロセス数に等しい台数のプロセッサを使用することにする。

また，シミュレーションの実行中に各プロセスはシステム内の他のプロセスと情報の交換を行なう必要がある。これを，「プロセス間通信」と呼ぶことにする。そして，今のところ，プロセス間通信の具体的な実現法は考えずに，任意のプロセス間でメッセージを送ることが可能であるとする。性能評価を行なう際には，プロセス間通信を行なうためのオーバーヘッド，より具体的に言えば所要時間のみが問題であるからこのように考えて差し支えない。

このように，シミュレーションモデルを，プロセスが通信ラインによって結合されたものと考え，図1のように



□：プロセッサ
○：プロセス　→：通信ライン

図1 システムの表記法

示す。通信ラインは矢印で示し，始点から終点へメッセージが送られると考える。

Ⅲ.プロセス間の時刻同期

シミュレーションを実行する際，各プロセスにおける処理の因果性が保たれなければならない。すなわち，プロセス内で実行される処理はシミュレーション・モデル内の時間的順序に従っている必要がある。たとえば，あるプロセスで $t=5$ の時刻に処理を行なった後，$t=3$ の処理を行なったのでは誤ったシミュレーション結果が得られてしまうことになる。

単一プロセッサにおける従来のシミュレーターでは，イベントリストによってこの問題を解決していた。しかしこの方法は，すべてのイベントを集中して管理する必要があり，マルチプロセッサによるシミュレーターでは各プロセッサの独立性を損うため採用できない。そこで，他の時刻同期法を考える必要がある。

今後，シミュレーターの時間とモデル内の時間という2種類の時間を用いるので混乱を避けるため，シミュレーターの時間を"Simulator Time(ST)"モデル内の時間を"Logical Time(LT)"と呼び，記号としてはSTにギリシア文字，LTに英字を用いることにする。

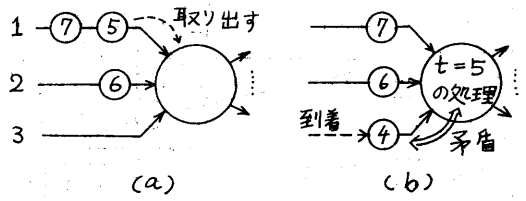


図2 因果性が崩れる場合

さて，シミュレーションが正しく行なわれている限り，プロセスのLTが逆行することはない。すなわち，ある通信ラインに送られる i 番目のメッセージの送信LTを t_i と書くと

$$t_1 \leqq t_2 \leqq \cdots\cdots \leqq t_{i-1} \leqq t_i \leqq t_{i+1} \leqq \cdots$$

となる。結局1本の通信ラインに関しては，送られてきた順に取り出せばよい。問題となるのは，図2のような場合である。同図(a)では $t=5$ のメッセージが最も早く到着したものなのでそれを取り出す。その後，同図(b)のように(a)ではメッセージが未到着であったライン3に $t=4$ のメッセージが到着したとすると，因果性が崩れてしまう。

この例からも明らかなように，メッセージが到着していない入力ラインが存在する限り，メッセージを取り出して処理をすることはできない。この方針に従った，プロセッサの標準的動作を図3に示す。ここで「メッセージの処理」は，そのメッセージに関連した統計量の計算，プロセスの状態変更，LTの更新などを含んでいる。

```
while シミュレーションが終了しない do
    すべての入力ラインにメッセージが
    揃うまで待つ;
    メッセージを1個取り出す;
    取り出したメッセージの処理を行なう;
    他のプロセスにメッセージを送る
od
```

図3 プロセスの標準的動作

この方法によって図4のような直列にプロセスが接続されたシステムのシミュレーションは正しく行なうことができる。ところが図5のように分岐が

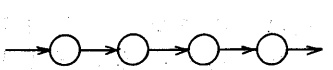


図4 プロセスの直列接続

存在するとデッドロックが生じ，それを防ぐためには時刻情報のみを持つ「null メッセージ」を送る必要があることが知られている。[1][3]

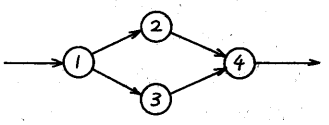


図5 分岐のあるシステム

たとえば，図5においてプロセス1（以下 P_1，P_2,…と言う），から P_2 にのみメッセージが送られるとする。そのとき P_3 にはまったくメッセージが送られないから P_3 はメッセージを出力しない。P_4 は P_3 からメッセージが到着するまで P_2 からのメッセージを取り出すことができないので，全く処理を進めることができない。

これを防ぐためには，メッセージを送らないプロセスにも時刻が変わったことを知らせる，あるいはその時刻までメッセージが送られないということを示す，「null メッセージ」を送る。つまり，P_1 から P_2 にメッセージを送る際，P_3 にも時刻情報として「null メッセージを送ってやる。すると P_3 はその null メッセージを受けて自分の時刻を変え，その情報を P_4 へ送る。P_4 は LT の順に P_2 あるいは P_3 からのメッセージを取り出し，null メッセージは無視して処理を続ければよい。

ところが，このようにしても図6のようにモデル中にループが存在するとデッドロックが生じる。図6のモデルで，P_3 内ではメッセージが生成されないとすると，P_3 は P_2 からメッセージが来ない限りメッセージを P_2 に送ることはできない。また P_2 は P_3 からメッセージが来ない限りメッセージを処理することはできない。従って，シミュレーションは実行されないことになる。

図6 ループのあるシステム

この場合，プロセスの挙動を予測することにより，デッドロックを回避することができる。図7にその例を示す。

(a)：これはシミュレーション開始直後の状況で P_2，P_3 は処理を行なっていないので時刻（LT）は0である。P_1 から P_2 に LT＝3にメッセージが送られて来ているが，P_3 から何も来ていないので，P_2 はそれを取り出せない。そこで P_2 は自分の動作を予測し，将来何らかの処理を行なうと少なくとも LT が2かかる，ということがわかったとする。すなわち，少なくとも LT＝2になるまで P_3 にメッセージを送らないわけで，その旨を P_3 に伝えるため，LT＝2という時刻情報を持った null メッセージを P_3 に送る。

(b)：P_3 はこのメッセージを受け取ると，自分の時刻 t_3 を null メッセージのの時刻に合わせる。次に，処理を行なうのに必要な時間を予測し，それを t_3 に加えて得られた時刻を P_2 に null メッ

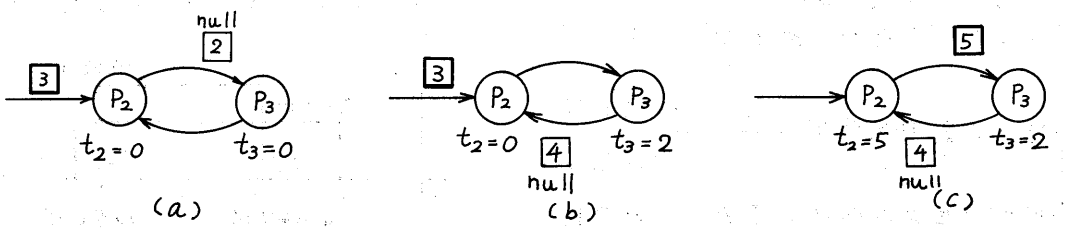


図7 予測によるデッドロックの回避

セージとして送る。

(c)：この場合，P_3からのnullメッセージはLT＝4なのでP_2はP_1からのメッセージを取り出すことができ，その結果をP_3に送ることができる。

もし，予測を行なわずにnullメッセージを送るだけにした場合は，同時刻のnullメッセージがループ内を永久に回り続けることになる。たとえば図7(a)の状況でP_2が予測を行なわず，P_3にLT＝0のnullメッセージを送ったとする。P_3はそれを受けて同じようにP_2に対してLT＝0のnullメッセージを送る。結局，時刻は全く進まないのでP_2はまたLT＝0のnullメッセージをP_3に送らなければならない。このようにして，LT＝0のnullメッセージがループ内を回り続ける。

また，図7の例では，P_2，P_3共に予測可能であるとしたが，最低1個，予測可能なプロセスがループ中に存在すれば，デッドロックなしにシミュレーションを実行することができる。[3]

Ⅳ. 予測法の分類

いままで述べてきたように，ループの存在するモデルではプロセスの挙動を予測することが不可決である。本研究では3種類の予測法を提案するが，これらは図8のように分類される。デッドロックが生じるのは，メッセージが未到着の通信ラインに，将来メッセージが到着する可能性があるか否か，ということを受信側のプロセスでは知ることができないためである。そこで

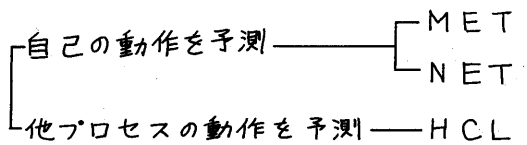


図8　予測法の分類

未到着ラインにnullメッセージを送りその時刻まではメッセージが送られないということを保証してやるのが「自己の動作を予測」する方法であり，逆に，何らかの方法で送信側プロセスの状態を判断し，通信ラインにメッセージが送られるか否かを決定するのが「他のプロセスの動作を予測」する方法である。次に，これらの方法の詳細について述べる。

(a)自己の動作を予測する方法：これは図7で述べた方法である。その動作は，

(i)通常のメッセージを受けた場合：そのメッセージに対する処理を行ない他のプロセスにメッセージを送る。通常のメッセージを送らない通信ラインにはnullメッセージを送る。

(ii)nullメッセージを受けた場合：このとき，プロセスのLTがtであったとする。何らかの方法で，それまではメッセージを出力しないという時刻t'を求め，その時刻を持つnullメッセージをすべての出力ラインに送る。

問題となるのはt'の予測法であるがここでは2種類の方法を提案する。

(i) MET (Minimum Execution Time)
これは，単純に，起こり得るすべての処理の最小実行時間をtに加え，t'とする方法である。たとえばメッセージの実行時間として図9のような確率密度が与えられていれば最小実行時間は1であるからt'=t+1とする。この方法では，0でない最小実行時間が存

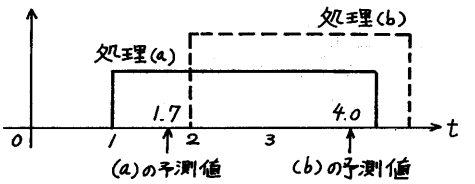


図9　実行時間の確率密度分布

在しなければならないことは明らかである。

(ii) ＮＥＴ（Next Execution Time）
これは，各処理毎にその処理を行なうのに必要な実行時間を計算しておき，それらのうちで最小のものをtに加える方法である。たとえば図9で，(a)の実行時間の予測値が1.7，(b)の予測値が4.0の時 $t'=t+1.7$ とする。この方法では，プロセス内で起るすべての場合の実行時間をあらかじめ計算しておくことができなければならない。受けとったメッセージに含まれるパラメタによって実行時間が決定されるような場合には適用できないのである。ところが，ＭＥＴとは異なり，$t=0$ における密度関数が有限値を持つことを許す。これは，予測される実行時間が0より大であれば問題なくシミュレーションを実行することができるからである。したがって，よく使われる負の指数分布も，この方法によって実行することができる。

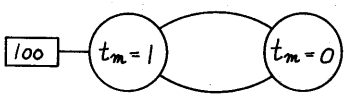
さて，これらの方法では，プロセスの時刻を進めるため，nullメッセージを送ることが必要であり，これに要する時間がオーバーヘッドとなる。そしてこのオーバーヘッドは，予測される実行時間によって大幅に変わる。図10はＭＥＴを想定しており，プロセス内の t_m は最小実行時間を示す。(a)のように予測可能なプロセスが P_1 のみで $t_m=1$ のように小さい時はnullメッセージが200個ループ中に送られないとＬＴ＝100に到着したメッセージを取り出すことはできない。ところが，(b)のように P_1，P_2 とも $t_m=10$ で予測が可能ならば，nullメッセージが送られるのは10回だけですむ。

また，等しい確率密度関数を持つ実行時間分布では，ＮＥＴの方がＭＥＴより大きい予測値を与えるから，ＮＥＴの方がより高い効率を示すと考えられる。

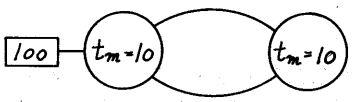
(b)他プロセスの動作を予測する方法：
この方法は(a)と異なり，nullメッセージを必要としない。したがってオーバーヘッドは極めて小さく効率は良いのだが適用条件が厳しい。

(a)の方法では，ある通信ラインにある時刻までメッセージが送られないことを示すnullメッセージを用いてデッドロックを回避していた。しかし，もしある通信ラインに関して，そのラインに将来メッセージが送られるか否かを受信プロセスが知ることが可能ならば，nullメッセージの必要はなくなる。

この方法が適用可能であるのは，図11のモデルを例に述べると次のような場合である。同図のプロセスＰに接続



(a)　$n=200$



(b)　$n=10$

図10 予測時間によるオーバーヘッドの変化

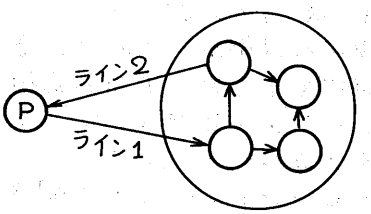


図11　ＨＣＬの例

するライン1，2について考える。もしライン2に送られるメッセージがライン1に送られるメッセージと1対1対応するならば，言い換えればライン1にメッセージを1個送ることによって，その応答としてライン2にメッセージが送られるならば，プロセッサPはライン2に関してメッセージが送られるか否かを知ることができる。すなわち，Pに状態変数Sを設け，最初はS＝0としておく。それで，ライン1にメッセージを送る毎にSに1を加えライン2からメッセージを取り出す毎にSを1減らす。すると

S=0：メッセージは送られない
S=1：メッセージが送られる

となる。したがってS＝0である限りライン2は無視することができる。なぜなら，シミュレーションの因果性が損われるのは，未到着ラインを無視して他のラインからメッセージを取り出した後，未到着であったラインにメッセージが到着したような場合であるから，メッセージが来ないとわかっているラインは無視してもかまわないのである。

このような通信ラインとしてはハンドシェイクによる通信が考えられる。そこで，これをHCL（Handshake Communication Line）と呼ぶ。

V. シミュレーションによる性能評価

時刻同期方式の違いにより，シミュレーターの性能がどのように変化するかを評価するため，汎用計算機上でシミュレーションを行なった。これにはマルチプロセッサ・シミュレーターのシミュレーションのために開発したもSSSと呼ぶシミュレーターを使用した。

使用したモデルは，前に図6で示した簡単なモデルである。P_1がProducer P_2がBuffer，P_3がConsumerであることを想定している。またP_3のキャパシティーは1であるとする。各プロセスにおける処理の概要を図12に示す。ここで"cycle"～"end"はいわゆるguarded regionである。

またモデルにおける実行時間の分布形によっても性能の差が生じることが予想されるので，一様分布と負の指数

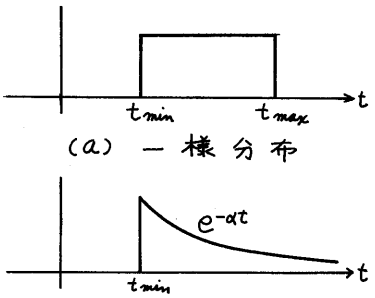
```
プロセス1:
  loop
    メッセージを生成しプロセス2に送る
  end

プロセス2:
  sw := true;
  cycle
   プロセス1からメッセージ到着:
     メッセージをバッファに入れる;
     if sw then
       バッファからメッセージを1個取り
       出しプロセス3へ送る;
       sw := false
     fi |
   プロセス3からメッセージ到着:
     if  バッファが空でない then
       バッファからメッセージを1個取り
       出しプロセス3へ送る
     else
       sw := true
     fi
  end

プロセス3:
  cycle
    プロセス2からメッセージ到着:
    メッセージの処理を行なう;
    プロセス2にメッセージを送る
  end
```

図12　各プロセスの処理

分布の変形についてシミュレーションを行なった。それらを図13に示す。



(a) 一様分布

(b) 負の指数分布

図13 実行時間の分布

また，nullメッセージを送るための処理時間（ＳＴ）は，通常メッセージ処理時間の1/4であるとした。

シミュレーション結果を図14，15に示す。ＭＥＴ，ＮＥＴはt_{min}の変化により性能が大幅に変化することは前に述べたとおりであるが，それを確認するためにt_{min}を変化させた。

さて，両グラフより明らかなことはＨＣＬはt_{min}に依存しないことである。これは，ＨＣＬにはt_{min}に関係する要素がないことから考えて当然であろう。

逆に，ＭＥＴ，ＮＥＴはt_{min}を0に近づけると急激に性能が悪化する。これは，nullメッセージを送るためのオーバーヘッドが増大するためである。特にＭＥＴはt_{min}の値をそのまま用いているため，それが顕著である。そして，単一プロセッサによるシミュレーションより悪化し得る。

ただ，ここで注意しなければいけないのは，単一プロセッサとマルチプロセッサではオーバーヘッドの種類が異なるということである。したがって，両者の性能を比較するのはむつかしい。図14，15で単一プロセッサの場合としているのは，nullメッセージによるオーバーヘッドのないＨＣＬにおける各プロセッサの所要時間を単に加え合わせたものであり，単一プロセッサにおけるイベントリストのようなオーバーヘッドは考慮していない。したがって単一プロセッサの場合の性能は，これよりも悪化するはずである。

また，今回のシミュレーションではプロセッサの台数が3台に過ぎないため，理想的な場合でも，シミュレーションに要する時間は1/3にしかならない。より大規模なモデルに対してプロセッサ台数を増やしてシミュレーションを行なえば，より良い結果が得られるのは当然といえよう。

さらに，このシミュレーションの場

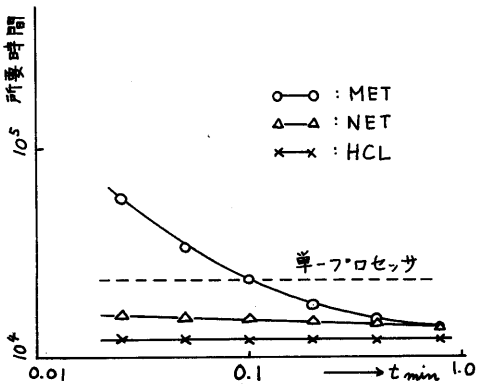


図14 一様分布によるシミュレーション

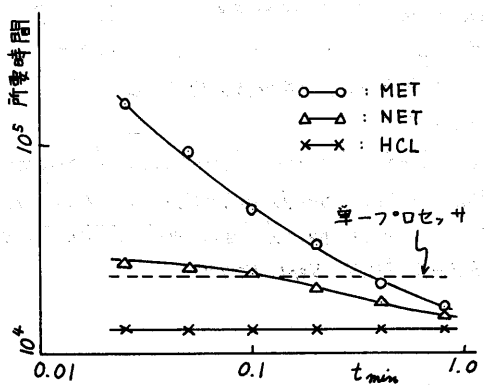


図15 負の指数分布によるシミュレーション

合，図12を見ればわかるように，プロセッサ2に処理が集中している。したがって，プロセッサ2の所要時間が支配的となり，他のプロセッサは待ち時間が多くなっている。それを示したのが表1である。より大規模なシステムのシミュレーションを行なう際には，各プロセッサの負荷が均等になるようプロセスを割り当てることが必要となろう。

Ⅵ.まとめ

以上のシミュレーションにより，マルチプロセッサによるシミュレーターの性能は，予測方法，およびモデルの実行時間分布により大幅に変化することが明らかとなった。

今後の予定としては，1プロセッサに多プロセスを割り当てる方法の検討大規模システムをシミュレートする場合の問題，特に予測法の選択に関する問題を中心として研究を進めたいと考えている。

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