Vol. 23 No. 5 情報処理学会論文誌 Sep. 1982

離散型シミュレーションの並列処理における予測を用いた時刻同期方式†

中川裕志†† 越田一郎†† 小沢年弘††

待ち行列系などの離散型シミュレーションは従来，汎用計算機上で多大な時間をかけて行われていた．近年の半導体技術の進歩で安価になったマイクロプロセッサを多数利用すると，並列処理によって高速かつ経済的なマルチプロセッサ型のシミュレータが可能となる．しかし，離散型シミュレーションにおいては，各プロセッサに割り当てられたプロセスの処理において，シミュレーション・モデルの時間的順序性を正しく保つための時刻同期アルゴリズムが必要である．とくにプロセス間を流れる情報の径路がループを持つ場合は，デッドロックの可能性がある．これを回避するためには，プロセスの動作を予測する方法がある．本論文では，MET, NET, HCL という3種の予測法を提案した．MET, NET は自プロセスの動作を，HCL は他プロセスの動作を予測する方法である．次に，これら3種類の予測法について，計算機シミュレーションにより性能評価を行った．その結果，MET, NET は，トランザクションの発生間隔やサービス時間の最小値に依存し，これが小さくなると性能の劣化が著しいが，HCL は，最小値の如何にかかわらず，高い性能を示すことが明らかとなった．

1. まえがき

従来，待ち行列系の解析などを行う離散型シミュレーションは汎用計算機上で GPSS などのシミュレーション向き言語を用いて行われてきた．しかし，大規模なシステムのシミュレーションを実行する場合には，客の到着，サービスの終了などのイベントの管理，統計処理等に多大の計算時間を要していた[1)~3)]．そのため，近年価格低下，性能向上の著しいマイクロプロセッサを多数用いた高能率なシミュレーション専用マルチプロセッサシステムの研究が数多く行われている[4),5)]．このような並列処理システムにおいては，各イベントに対応する処理（以下，これをプロセスと呼ぶ）を別々のプロセッサで実行する方法が検討されている．しかし，シミュレーションの実行にあたっては，イベント処理順序の因果性を守るために，各プロセスでは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．このため各プロセスは互いに全く独立に処理を進めることができず，この制約条件を満たすための時刻同期を行う必要がある．各プロセス間でイベントを送る径路がループを持つ場合は，時刻同期方式に起因するデッドロックが起こる可能性があるため，何らかの対策が必要である．本論文では，プロセスの動作を予測することによってデッドロックを回避する3種類の方法を提案し，計算機シミュレーションによりこれらの方法の性能の比較評価を行っている．

2. マルチプロセッサ化の方法

ここでは，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，シミュレーションによって表現している世界，すなわち待ち行列系などでは並列に動作している．たとえば，複数窓口でのサービスの同時実行があげられる．したがって，シミュレータにおいてもハードウェアが許せば，各プロセスを並列に実行することが可能である．すなわち，各プロセスを別個のプロセッサに割り当て，マルチプロセッサシステムとして並列に実行できる．この場合は，1台のプロセッサにおけ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．

この方式では，プロセスのプロセッサへの割り当て方法が重要な問題になる．ここでは簡単に，1プロセスに1プロセッサを割り当てる方法について検討する．したがって，プロセス数とプロセッサ数は等しい．さて，シミュレーションを実行するためには，各プロセスはシステム内の他のプロセスとイベントの発生などに関する情報の交換を行う必要がある．ここでは，プロセス間に張られた単方向性の通信ラインによって，イベント情報などが書かれたメッセージがプロ

† Synchronization Methods with Prediction of Parallel Discrete Simulation System by HIROSHI NAKAGAWA, ICHIRO KOSHIDA and TOSHIHIRO OZAWA (Faculty of Information Engineering, Yokohama National University).

†† 横浜国立大学工学部情報工学科

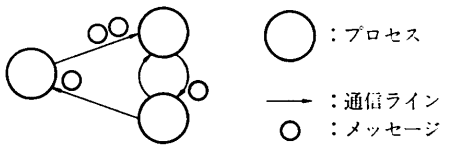


図 1 シミュレータのモデル
Fig. 1 Simulator's Model.

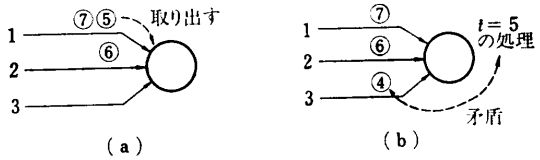


図 2 メッセージ処理順序の乱れる場合
Fig. 2 The case of losing message order.

セス間で送受信されるモデルを考える．モデルの一例を**図1**に示す．通信ラインの方向は矢印で示す．また，1プロセッサ1プロセスなので，図1のプロセスは物理的に1台のプロセッサに対応している．

3. ループのない場合のプロセス間の時刻同期方式およびその問題点

図1のようなモデルによってシミュレーションを実行する際，正しい結果を得るためには，プロセスで実行される処理がシミュレーションしている対象の系の時間的順序に従っている必要がある．このためには，プロセスにおいて各入力通信ラインに到着したメッセージの処理順序が重要である．たとえば，あるプロセスでシミュレーション対象の系の時刻 $t=5$ で発生したメッセージを処理した後，$t=3$ で発生したメッセージを処理したのでは，正しい結果が得られない．単一プロセッサによる従来のシミュレーションでは，メッセージ(すなわちイベント)を発生時刻順につないだ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，すべてのメッセージを集中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には不適当である．したがって，他の時刻同期方式を考える必要がある．

今後，混乱を避けるために，シミュレーションを実行しているプロセッサの物理的時刻を ST (Simulator Time)，シミュレーションの対象となっているモデルの時刻を LT (Logical Time) と呼ぶことにする．

さて，シミュレーションが正しく行われている限り，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，

$$t_1 \leq t_2 \leq \cdots \leq t_{i-1} \leq t_i \leq t_{i+1} \leq \cdots \qquad (1)$$

となる．結局1本の通信ラインに関しては，メッセージ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**のような複数の通信ラインがある場合である．図2(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，図2(b)のように(a)ではメッセージが未到着であった通信ライン3に $t=4$ のメッセージが到着するとメッセージ処理の時間的順序が乱れてしまう．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通信ラインが存在する限り，メッセージを取り出して処理することはできない．したがって，プロセスの動作の概略は次のように表せる．

```
while シミュレーションが終了しない
  do すべての入力通信ラインにメッセージが到着
     するまで待つ;
     最小の送信 LT を持つメッセージを取り出
     す;
     取り出したメッセージの処理を行う;
     他のプロセスにメッセージを送る
  od                                    (2)
```

ここで，“メッセージの処理”とは，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 の更新などである．

この方法で，ループのないトリー状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実行できる．ところが，**図3**のように分岐と結合が存在する場合はデッドロックを生じ，それを防ぐためには時刻情報のみを持つメッセージ (以下これを，null メッセージ，あるいは null と呼ぶ) を送る必要があることが知られている[4),5)]．たとえば，図3においてプロセス1 (以下 $P_1, P_2, \cdots$ と呼ぶ)から P_2 にのみメッセージが送られるとする．その時 P_3 には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せないので全く処理が進まずデッドロックになる．これを防ぐためには，メッセージを送ら

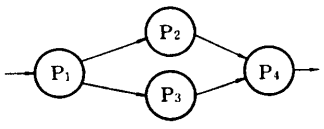


図 3 分岐結合のあるモデル
Fig. 3 Fork and join type model.

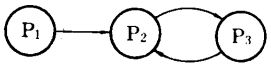


図 4 ループのあるモデル
Fig. 4 Model with loop.

ないプロセスにも時刻が変ったこと，つまりその時刻まではメッセージが送られないことを示す null を送る．つまり，P_1 から P_2 へメッセージを送る際，P_3 にも時刻情報の null を送ってやる．すると P_3 はその null を受けて自分の時刻を更新し，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 は読み捨てて処理を続ければよい．

しかし，このようにしても**図 4** 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 4 のモデルで，P_3 では自立的にメッセージを生成しないとすると，P_3 は P_2 からメッセージが到着しない限りメッセージを P_2 に送らない．また P_2 は(2)の時刻同期方式の制約のため P_3 からメッセージが到着しない限りメッセージを処理できない．したがってデッドロックになる．このようにループのある場合は，(2)に示したアルゴリズムだけでは，時刻同期方式は不完全である．

4. 予測を用いたプロセス間の時刻同期方式

4.1 予測によるデッドロックの回避

3 章の(2)で示した時刻同期方式を完全なものとするためには，ループのある場合のデッドロック回避方法を導入しなければならない．プロセスの挙動を予測することによりデッドロック回避が可能になる．**図 5** にその例を示す．

(a) これはデッドロック状態で P_2, P_3 は処理を行っていない．LT=0 と仮定する．P_1 から P_2 に LT=3 のメッセージが送られているが，P_3 から何も到着していないので，P_2 はこのままで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て，将来何らかの処理を行うと少なくとも LT が 2 かかる，ということがわかったとする．つまり，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ので，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 を P_3 に送る．

(b) P_3 はこの null を受けとると，自分の時刻 t_3 を，到着した null の時刻に合わせる．次に，P_3 で処理を実行する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書いた null を P_2 に送る．

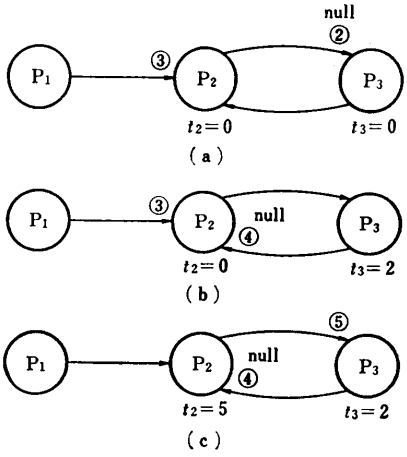


図 5 予測によるデッドロックの回避
Fig. 5 Avoiding deadlock by prediction.

(c) この場合，P_3 からの null は LT=4 なので，P_2 は P_1 からの LT=3 のメッセージを取り出すことができ，その処理を行い，結果を P_3 に送ることができる．

もし，予測を行わずに null を送ると，同時刻の null がループを永久に回り続ける．たとえば図(5)(a)の状態で P_2 が予測を行わず，LT=0 の null を送ったとする．P_3 はそれを受けて同様に P_2 に対して LT=0 の null を送る．結局，LT は全く進まないため，LT=0 の null がループを回り続ける．

図 5 の例では，P_2, P_3 とも予測可能であるとしたが，ループ上に最低 1 個予測可能なプロセスが存在すれば，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．

このように，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欠である．本論文では，3 種類の予測法を提案するが，これらは，自プロセスの動作を予測する方法と他プロセスの動作を予測する方法とに分類される．デッドロックを生じるのは，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，未到着通信ラインに送信側プロセスである時刻まではメッセージを送らないことを保証する null を送る方法が“自プロセスの動作を予測する方法”である．逆に，何らかの方法で送信側プロセスの状態を判断し，入力通信ラインに将来メッセージが送られてくるか否かを決定する方法が“他プロセスの動作を予測する方法”であ

る．以下に，これらの方法の詳細について述べる．

4.2 自プロセスの動作を予測する方法

これは，図5で説明した方法である．その動作は，

（1） 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 を送る．

（2） null を受けた場合：

このとき，プロセスの LT=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送信しないという時刻 $\hat{t}(>t)$ を求め，$\hat{t}$ の時刻情報を持つ null をすべての出力通信ラインに送出する．

ここで問題となるのは，$\hat{t}$ の予測法であるが，ここでは2種類の方法を提案する．

（1） MET (Minimum Execution Time)

これは，LT においてメッセージを処理した場合に起こりうる最小の間隔 t_{min} を，t に加えて $\hat{t}$ とする方法である．たとえば，LT のメッセージ発生間隔の確率分布が**図6**の形で与えられたとすると，$t_{min}=1$ であるから，$\hat{t}=t+1$ とする．この方法は，$t_{min}>0$ でなければならない．

（2） NET (Next Execution Time)

これは，次にメッセージを処理した時のメッセージ発生間隔 t_p をあらかじめ計算し，t に加えて $\hat{t}$ とする方法である．たとえば図6でメッセージ発生間隔の予測値 $t_p=4$ であったとすると，$\hat{t}=t+4$ とする．この方法は，プロセスにおいて次のメッセージ発生間隔を，メッセージを処理する前に計算できなければならない．t_p が，到着メッセージに含まれるパラメータによって連続的に変化するような場合は適用できない．

さて，これらの方法では，プロセスの時刻を進めるために null を送ることが必要であり，これに要する ST の時間がオーバヘッドである．オーバヘッドはプロセスにおいて予測された LT の値に依存する．**図7**は，MET を想定しており，各プロセスの t_{min} が図に示すように与えられていたとする．(a)のように予測可能なプロセスが P_1 のみで $t_{min}=1$ のように小さい場合は，null が 200 個ループ中に送出されないと LT=100 に到着したメッセージを取り出せない．(b)のように P_1, P_2 とも $t_{min}=10$ で予測可能なら，null が送られるのは 10 回だけである．このことからもわかるように，MET では t_{min} が小さくなるとオーバヘッドが急激に増加する．また，MET と NET を比べると，NET のほうがより大きな予測値を与えるから，オーバヘッドが少なく，効率は良いと考えられる．

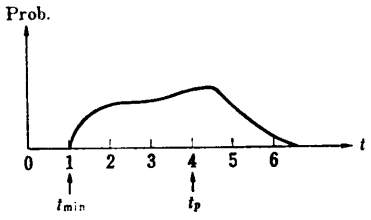


図6 メッセージ発生間隔の確率密度分布の例
Fig. 6 An example of distribution of message interval.

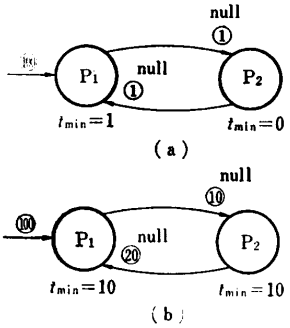


図7 予測値 t_{min} によるオーバヘッドの変化
Fig. 7 Overhead when $t_{min}=0, 1$ and $t_{min}=10$.

4.3 他プロセスの動作を予測する方法

(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 を用いてデッドロックを回避している．しかし，もしある通信ラインに関して，その通信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 を送る必要がなくなり，オーバヘッドはきわめて小さくなる．この方法が適用可能であるのは，**図8**のモデルを例に述べると次のような場合である．図8のプロセス P_1 に接続する通信ライン 1，2 について考える．もし，入力通信ライン2に送られるメッセージが出力通信ライン1から送出するメッセージと1対1に対応するならば，つまり通信ライン1にメッセー

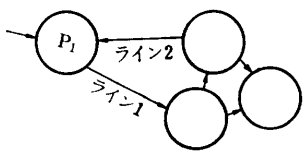


図8 HCL の例
Fig. 8 Example model of HCL.

ジを1個送ることによって，その応答として通信ライン2にメッセージが送られてくるならば，プロセス P_1 は通信ライン2にメッセージが将来到着するかどうかを知ることができる．すなわち，初期値=0 の状態変数 S を設け，通信ライン1にメッセージを送るごとに S に1を加え，通信ライン2に到着したメッセージを取り出すごとに S を1減らす．すると，P_1 には

$S=0$：将来メッセージが通信ライン2に到着する可能性がない

$S\geqq 1$：将来メッセージが通信ライン2に到着する可能性がある

となる．したがって，$S=0$ である限り，P_1 は通信ライン2を無視して，他の通信ラインに到着したメッセージを取り出して処理できる．なぜなら，シミュレーションにおけるLTの時間的順序性が損われるのは，未到着の通信ラインを無視して他の通信ラインからメッセージを取り出して処理した後，未到着の通信ラインに，より小さいLTを持つメッセージが到着した場合であるから，メッセージが来ないとわかっている通信ラインは無視してかまわないのである．このような通信ラインはハンドシェイク方式と考えられるので，この予測方式を HCL (Handshake Communication Line) と呼ぶことにする．

5. 予測を用いた時刻同期方式の性能評価

4章で述べた3種類の予測方法による時刻同期方式を用いた場合のマルチプロセッサによるシミュレータの性能を評価するために，汎用計算機上でシミュレーションを行った．ここでは，3種のモデルについて検討した．

モデルⅠ：図4で示したループを持つ最も簡単なモデルである．P_1 が客の発生源，P_2, P_3 は直列につながった窓口に相当する系のモデルと考えられる．また P_3 では，メッセージは1個しか受けつけない．これは，2段目の窓口に待ち行列を許さない場合と考えられる．各プロセスにおける処理の概要を次に示す．ただし，cycle～end は，いわゆる guarded region であり，loop～end は無限回繰り返しを示す．

```
P1:
  loop
    メッセージを生成しプロセス2に送る
  end
```

```
P2:
  SW:=true;
  cycle
    P1 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=true then
        バッファからメッセージを1個取り出し,
        P3 へ送る
        SW:=false
      fi |
    P3 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,
        P3 へ送る
      else
        SW:=true
      fi
  end
```

```
P3:
  cycle
    P2 からメッセージ到着:
    メッセージの処理を行う;
    P2 にメッセージを送る
  end
```
(3)

ここで，P2 におけるバッファとは，第1段窓口のサービスおよび待ち行列，P3 からのメッセージは，P2 からのメッセージ送信許可信号と考えられる．

モデルⅡ：**図9**に示す多段直列型モデルである．これは，多段直列窓口の待ち行列系のモデルと考えられる．各プロセスの処理はモデルⅠと同様なので省略する．

モデルⅢ：**図10**に示す分岐型モデルである．P_2 に

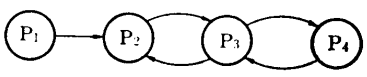


図9 モデルⅡ
Fig. 9 Model Ⅱ.

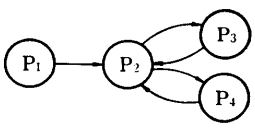


図10 モデルⅢ
Fig. 10 Model Ⅲ.

表 1 各モデルの各プロセスにおける α の値
Table 1 Values of α of each process in each model.

	P_1	P_2	P_3	P_4
モデル I	0.2	1.0	3.0	—
モデル II	0.2	1.0	1.0	3.0
モデル III	0.2	1.0	3.0	3.0

おいては，P_3, P_4 のうち空いているほうにメッセージを送る．これはたとえば，計算機システムで CPU (P_2 に対応) が複数の入出力装置 (P_3, P_4 に対応) を用いる系のモデルと考えられる．

これらのモデルにおいて，各プロセスの LT におけるメッセージ送信間隔の確率分布 $P(t)$ は，負の指数分布を t_{min} シフトした次式の分布とした．

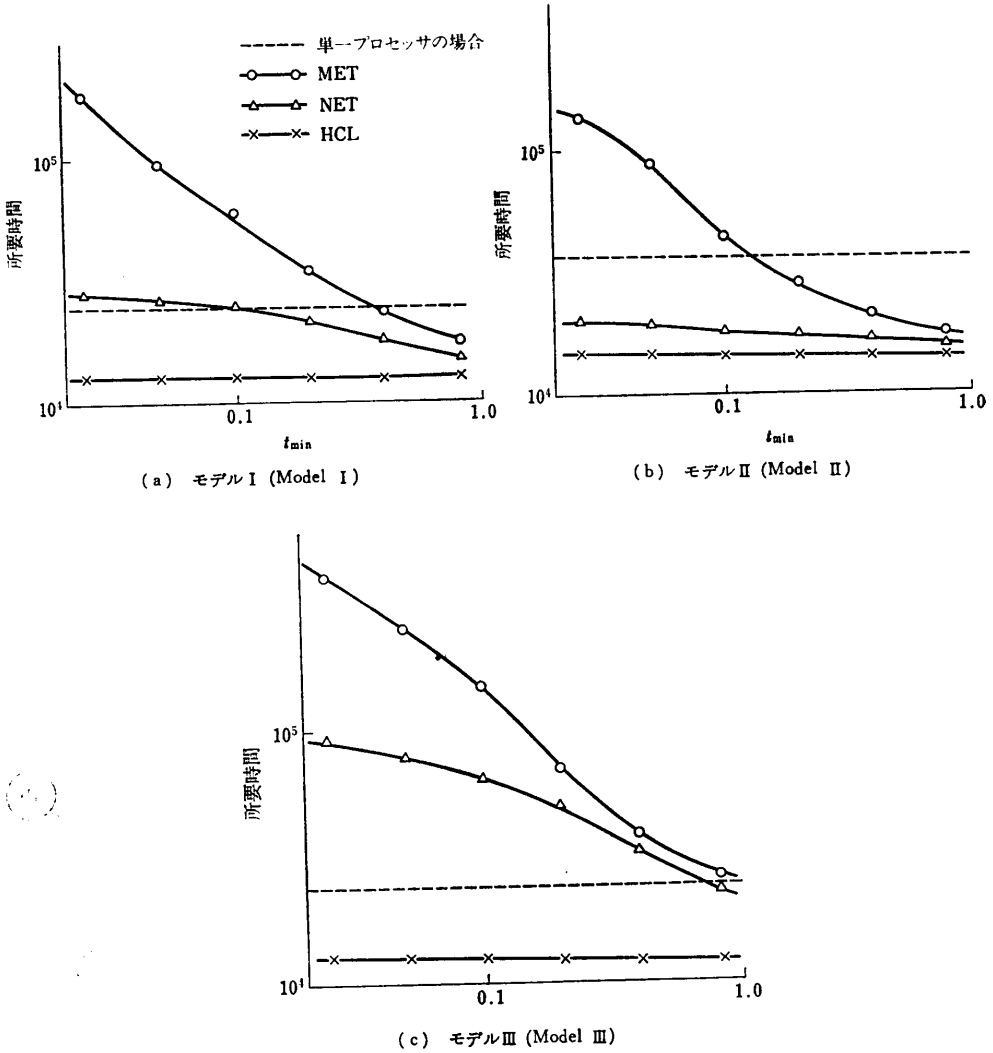
$$t<t_{min} : P(t)=0$$
$$t\geqq t_{min} : P(t)=\alpha e^{-\alpha(t-t_{min})} \quad (4)$$

各プロセスの α は**表 1** の値を用いた．

また，各プロセスを実行しているプロセッサのメッセージ当たりの ST での処理時間は，2〜4 の一様分布，null の場合は 0.5〜1 の一様分布とした．以上の条件において，MET, NET, HCL の各方式について，2000 トランザクションを処理するシミュレーション実行に要する ST での所要時間を求めた結果を**図 11** (a)〜(c)に示した．

各方式のシミュレーション結果の比較

(1) MET, NET は t_{min} が小さくなると，null



(a) モデル I (Model I)　(b) モデル II (Model II)

(c) モデル III (Model III)

図 11 各方式のシミュレーション所要時間
Fig. 11 Time for simulation by MET, NET and HCL of model I, II, III.

送信のオーバヘッドが大きくなることがすべてのモデルにおいて確認された.

(2) NET のほうが MET より常によい効率を示すことが,すべてのモデルにおいて確認された.

(3) MET, NET は, t_{min} が小さくなると,単一プロセッサでの処理時間より長くなる場合がある.ここで単一プロセッサでの処理時間とは, null によるオーバヘッドのない 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る.実際は,この上さらに,イベントリスト処理のようなオーバヘッドが加わるから,単一プロセッサの処理時間はこれより長くなる.

(4) HCL の所要時間は,すべてのモデルについて, t_{min} にほとんど依存しないことが確認される.これは HCL の場合, t_{min} によって変化する null 送信のオーバヘッドがないことから予測されたことである.

(5) モデルⅢにおいては, NET の性能が MET 近くまで劣化している.この理由は, P_3, P_4 のいずれか一方が空きであっても, P_2 を実行するためには,時刻同期方式の制約から空きプロセスと null を交信して時刻更新を行うオーバヘッドが大きいためである.この例のように, NET はモデルによって性能が変わる.

プロセッサ効率

HCL 方式において, P_1〜P_4 の各プロセスを実行しているプロセッサの ST における実行効率は**表 2** のようになる.ただし,実行効率は次式で定義される.

$$\text{実行効率} = \frac{(\text{所要時間} - \text{プロセッサの待ち時間})}{\text{所要時間}} \quad (5)$$

時刻同期の制約のない P_1 の効率が悪いのは,トランザクションの発生が他のプロセッサの処理に比べて早早に終了し,残りの時間がアイドリングの待ち時間になってしまうからである.この結果から,モデルによって各プロセッサの効率が異なることがわかる.モデルⅡでは,各プロセスを1台のプロセッサに割り当てても高い効率が得られるが,分岐型のモデルⅢでは, P_3 と P_4 を1個のプロセッサに割り当てることによって,効率の改善が期待される.

ここで説明したシミュレーション結果は,プロセッサ台数が, 3, 4 台であるため,理想的な場合でも,シミュレーションに要する時間は単一プロセッサの場合に比べて 1/3〜1/4 にしかならないが,より大規模なモデルに対しプロセッサ数を増せば,並列度が増してより良い結果が得られることが予想される.

表 2 各モデルの各プロセッサにおける実行効率
Table 2 Efficiency of each processor in each model.

	P_1	P_2	P_3	P_4	平均
モデル I	0.47	0.95	0.47	—	0.63
モデル II	0.42	0.88	0.88	0.44	0.65
モデル III	0.47	0.95	0.33	0.14	0.47

6. むすび

マルチプロセッサによる離散型シミュレーションの並列処理においては,その性能が時刻同期方式で用いるプロセス動作の予測方法によって,大きく変わることがわかった.とくに, null を送る必要のない HCL 方式は,適用条件は厳しいが,モデルによらず高い性能を示すことが確認された.

今後の課題は次のようなものがある.

(1) 本論文では,各プロセッサのメモリ容量はとくに制限しなかったが,これに制限を加えると効率はどうなるか.

(2) 1プロセッサに複数プロセスを割り当てる方式の検討.

(3) 提案した時刻同期方式向きの計算機アーキテクチャの検討.

などが挙げられる.

参考文献

1) Vaucher, J. G. and Duval, P.: A Comparison of Simulation Event List Algorithms, *Comm. ACM*, Vol. 18, No. 4, pp. 223-230 (1975).
2) Wyman, F. P.: Improved Event-Scanning Mechanims for Discrete Event Simulation, *Comm. ACM*, Vol. 18, No. 6, pp. 350-353 (1975).
3) Franta, W. R. and Maly, K.: An Efficient Data Structure for the Simulation Event Set, *Comm. ACM*, Vol. 20, No. 8, pp. 596-602 (1977).
4) 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, 昭54信学会全国大会 1413, pp. 6-33 (1979).
5) Candy, K. and Misra, J.: Distributed Simulation, A Case Study in Design and Verification of Distributed Programs, *IEEE Trans. on* SE-5 No. 5, pp. 440-452 (1979).

(昭和56年12月9日受付)
(昭和57年2月16日採録)