計算機アーキテクチャ 136－7
（2000．1．7）

# 共有メモリ型並列計算機の分散シミュレータ

今福 茂† 大野和彦† 中島 浩†

本報告は，共有メモリ型並列計算機のアーキテクチャ・レベルのシミュレーションに内在する豊富な並列性を利用して，効率のよい分散シミュレータを構築する手法について述べたものである。一般に分散シミュレータでは，シミュレータ内の計算ノード間での時刻管理の効率が問題となる。我々はこの問題の解決策として，遠隔アクセスを含めた命令レベルの論理的な結果を再現する論理レベルのシミュレーションと，遠隔アクセスに伴うコヒーレンス制御メッセージのタイミングを高精度で再現する物理レベルのシミュレーションとを分割する方法を提案する。前者は精密な時刻管理が不要であることから，Virtual Shared Memory の手法を用いて効率よく分散実行される。一方後者は，前者によって得られたアクセスの履歴に基づく逐次シミュレーションにより，対象システムのネットワークの挙動を含めて動作タイミングを高精度に再現する。ここで問題となるアクセス履歴の圧縮については，前者においてコヒーレント・キャッシュを部分的にシミュレーションし，遠隔アクセスの可能性があるものだけを抽出することにより解決している。また論理／物理レベルのシミュレーションを並行実行することにより，アクセス履歴保存のオーバヘッドを削減しつつ，全体的なシミュレーション時間の短縮も図っている。

# A Distributed Simulator for Shared Memory Multiprocessors

SHIGERU IMAFUKU,† KAZUHIKO OHNO† and HIROSHI NAKASHIMA†

This paper proposes an efficient techinique for distributed simulation targeting shared memory parallel machines. Our approach to solve the problem of distributed clock management, which is the well-known bottleneck of distributed simulators, is as follows. We decompose our simulator into *logical* and *phisical* ones. The former simulates the instraction level logical behaviour of the target machine in parallel, without precise clock management, using virtual shared memory technique. The latter is responsible to determine precise timing of each interprocessor event in the target machine taking account of its physical network behaviour, based on the access trace generated by the logical simulator. In order to make the trace as small as possible, the logical simulator partially simulates the coherent cache and extracts only the accesses potentially remote. The logical and phisical simulators are concurrently executed in order to reduce the overhead for the access trace preservation, and also to shorten the total simulation time.

## 1. はじめに

新たなアイディアや技術を採り入れた並列計算機を考案・設計する際には、その動作の正当性や性能面での有効性を事前検証することが必要であり、そのための重要な道具としてアーキテクチャレベルのシミュレータが用いられることが多い。このようなシミュレータに関する重要な技術的課題は，多数のプロセッサの局所的および大域的な振る舞いを模擬するために必要な膨大な計算を，いかに短時間で実行するかにある。

この課題に対する一つの自然な解決策は，対象システムである並列計算機に内在する並列性を利用して、シミュレーションも並列・分散化することであり，実際にいくつかの試みがなされている。しかし性能検証のためには，対象システムのプロセッサ間通信タイミングを高精度に再現する必要性など，並列・分散シミュレーションの一般的な課題である時刻管理に関わる困難な問題があり，一般的かつ決定的な効率のよい手法は見い出されていない。

そこで本研究では，共有メモリ型並列計算機を対象とする高速な分散シミュレータを構築するために，その論理的な動作と物理的な挙動を分離してシミュレーションする手法を考案した。この手法は我々がすでに提案した論理／物理事象の分離手法[1)]に基づくものであり，Virtual Shared Memory (VSM) を応用した論理的シミュレーションの分散化と，物理的挙動の再現に必要なメモリ・アクセス履歴の圧縮処理が大きな特徴である。また論理的／物理的なシミュレーションの

† 豊橋技術科学大学情報工学系
Dept. of Computer and Information Sciences, Toyohashi Univ. of Tech.

並行実行により，アクセス履歴保存のオーバヘッドを削減しつつ，全体的なシミュレーション時間の短縮も図っている。

以下，本報告では、2章で本研究の課題を関連研究と対比する形で述べた後、3章でシミュレータの構成を示し，続いて4章でアクセス履歴の圧縮手法を示す。最後に5章で本研究の成果についてまとめ、今後の課題を示す。

## 2. 分散シミュレーションの課題

我々のシミュレータが対象とする共有メモリ型並列計算機はもちろん，離散シミュレーションの対象システムの多くは豊富な並列性を内在している。したがって対象システムをサブシステムに分割し，サブシステムごとのシミュレーションを並列に実行することは，極めて自然な発想である。たとえば我々のシミュレータにおいて，対象システムの1個あるいは複数個のプロセッサとキャッシュを，分散シミュレータの一つのノードにマップすれば☆，キャッシュミスやコヒーレンス維持操作が生じない限り，ノード内で局所的にシミュレーションを行なうことができる。

しかし一般にはサブシステム間に何らかの相互作用があるため，それらに対応してノード間で通信される遠隔イベントと，ノード内で生じる局所イベントとの前後関係を整合させるための分散時刻管理が必要となる。これを単純な方法，たとえば対象システムのクロックごとに全ノードが同期するといった方法で実現すると，ノード間の同期・通信によって禁止的なオーバヘッドが生じるため，並列処理の効果が得られないことは明らかである。

**一般的な分散時刻管理**

この分散時刻管理の問題を一般的に解決する手法は既にいくつか提案されており，中でも Jefferson らによる Time Warp 法[2]は，論理回路シミュレーションをはじめ広く応用されている代表的手法である。しかしこの手法は，ノード間でのイベントの前後関係逆転を，保存しておいた状態への巻き戻しにより補償するものであるため，プロセッサのように複雑な論理構造を持つサブシステムでは非現実的である。また巻き戻しの確率を低くするためには遠隔イベントを迅速にノード間で伝える必要があるため，共有メモリに関する細粒度通信を分散環境で直接的にシミュレートしなければならず，分散シミュレーションには適合しにくい。

**論理的／物理的シミュレーションの分離**

一方我々は，プロセッサが実行する命令の論理的な動作と，キャッシュミスとコヒーレンス維持操作（以下,「広義のミス」という）に伴うプロセッサ間の物理的な通信を分離することにより，シミュレーション効率を改善する方法を提案した[1]。前者における時刻管理はプログラムの意味を保つに十分な精度で行なえばよく，後述するように Virtual Shared Memory の技法を用いれば分散環境でも効率よく実行できる。一方後者は，プロセッサ間ネットワークでの競合などをシミュレーションするために精密な時刻管理が必要であるが，前者に比べて対象の論理構造が単純であるため，逐次シミュレーションでも十分な性能が期待できる。

朴らによる分散メモリ型並列計算機のシミュレータ VIPPES[3]も，ある意味で同じ考え方に基づいている。VIPPES は，プログラムを実行しながらプロセッサ間通信の履歴を保存するフェーズと，通信履歴に基づいてネットワークの挙動をシミュレーションするフェーズからなり，前者が論理的な，後者が物理的なシミュレーションに対応する。

**履歴の削減**

この分離シミュレーションで問題となるのは，物理的シミュレーションのソースとなる履歴の大きさである。VIPPES は分散メモリシステムが対象であるため，通信処理を特定することが用意であり，かつ履歴の大きさが実行命令数に対して十分小さいことが期待できる。しかし我々が対象とする共有メモリシステムでは，命令フェッチも含めた全てのメモリアクセスは，物理的事象である広義のミスの潜在的な要因である。したがってごく単純に履歴を保存すると，実行命令数と同量以上の履歴量となってしまい，たとえばファイルへの保存には非現実的な大きさとなる。

そこで我々は，以下の二つの方法により，アクセス履歴に関する問題を解決することとした。

- 論理的シミュレータにおいてキャッシュを部分的にシミュレーションし，実行タイミングによらず確実にヒットするようなアクセスの履歴は残さないようにする。この結果，命令フェッチについては全ての，またデータ・アクセスについても大部分のヒットが除去される。
- 論理的／物理的シミュレーションをパイプライン的に並行実行することにより，履歴のファイル保存を不要とするとともに，全体的なシミュレーション時間を短縮する。

## 3. 分散シミュレータの構成

### 3.1 対象システム

我々のシミュレータが対象とするシステムは，図1に示すようなコヒーレント・キャッシュを持つ共有メモリ型並列計算機である。共有メモリの構成方式は集中型／分散型のいずれでもよく，物理的シミュレータを改変することによって任意の構造のネットワークにも対応できる。またメモリ・モデルは Release Consistency (RC) あるいはより強いモデルとする。

☆ 以下，混乱が生じない限り，対象システムの計算ノードを単に「プロセッサ」といい，分散シミュレータの計算ノードを単に「ノード」という。

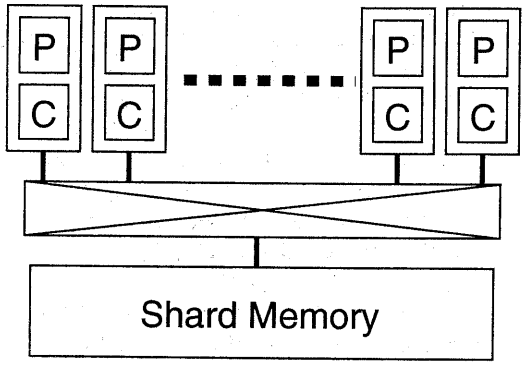


図1　対象システム

プロセッサの命令セット・アーキテクチャは原理的には任意であるが，現在開発中のシステムではSPARC Version 8に限定している。またシミュレーションの効率を重視して，キャッシュミスを起こさない限りCPIが1であるようなハードウェアを想定しているが，これも原理的には任意である。ただし後述する方法によってアクセス履歴を削減しつつ，対象システムの動作タイミング・シミュレーションを完全に精密なものとするためには，以下の条件が必要となる。

**条件1**　広義のミスを生じるメモリ・アクセスと同期操作の集合を$\mathcal{M}_{miss}$としたとき，$m \in \mathcal{M}_{miss}$の対象システムにおける実行タイミングは$\mathcal{M}_{miss}$のみから求めることができる。ただし全ての$m \in \mathcal{M}_{miss}$について，操作の種類やアドレスの他，論理的シミュレーションによる仮想実行タイミング☆は既知であるとする。

この条件は，たとえばin-orderで実行／完了する簡単なプロセッサでは成り立つが，out-of-order実行／完了するものについては成り立たない。しかし多くのプロセッサについては，アクセス遅延の影響をアクセス間隔やデータフローにより近似的に求めることにより，実用上は十分な精度が得られるものと見込まれる。

### 3.2　対象プログラム

シミュレータの入力は，POSIXあるいはSolarisスレッドライブラリを用いて記述されたマルチスレッド・プログラムの実行形式である。同期操作や入出力等のライブラリ関数は，シンボルデバッグ情報に基づき抽出して処理するため，ソースレベルでの特別な指示などは不要である[1)]。

また論理的／物理的シミュレーションを分離して行なうため，以下の定義に基づく決定的であるという条件が，対象プログラムに課せられる。

**定義1**　二つの共有メモリシステム$S$と$S'$においてプログラム$\pi$を実行した際に行なわれるメモリアクセスと同期操作の集合を$\mathcal{M}(\pi, S)$と$\mathcal{M}(\pi, S')$とする。アクセスを行なったプロセッサ，アクセス操作の種類，アドレス，およびアクセスのプログラム順に基づき，$\mathcal{M}(\pi, S)$が$\mathcal{M}(\pi, S')$に一対一写像できるとき，またその場合に限り$S$と$S'$は$\pi$について**相似**であるという。プロセッサ数が同じであり，かつRelease Consistencyを満たすような任意の$S$が$\pi$について相互に相似であるとき，またその場合に限り$\pi$は**決定的**であるという。

☆ たとえばヒットと同じ遅延を仮定したもの。

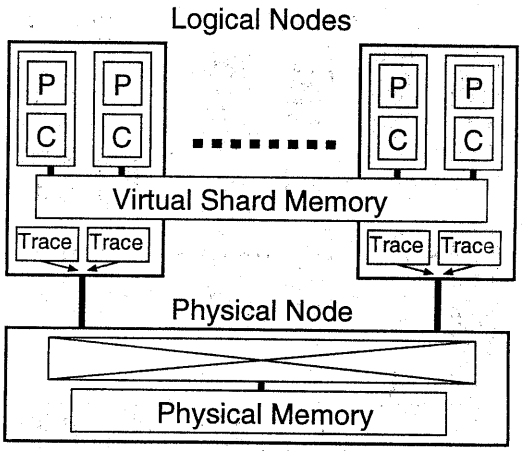


図2　シミュレータの構造

この条件は3)における制約と同様であり，spin lockやバリア同期のように実装上は複数のメモリアクセスからなる同期操作を仮想的に単一操作とみなすことにより，多くのデータ並列プログラムについて成立すると見込まれる。また決定的なプログラムは，以下の意味で同期的でもある。

**定義2**　メモリアクセス$m$が，同一プロセッサにおけるプログラム順あるいはプロセッサ間の同期操作に基づきアクセス$m'$に対して先行することを，$m \prec m'$と表記する。少なくとも一方が書込であるような同一アドレスに対するアクセス$m$と$m'$は競合するといい，アドレス$a$に関する任意の競合アクセス$m$と$m'$について$m \prec m'$または$m' \prec m$であるとき，またその場合に限り，$a$は同期的であるという。また任意のアドレスについて同期的であるようなプログラムも，やはり同期的であるという。

### 3.3　シミュレータの構造

シミュレータは図2に示すように，論理的シミュレーションを担当する複数の論理ノードと，物理的シミュレーションを行なう単一の物理ノードである。現在開発中のシステムでは各ノードはPCであり，100Base-TXのスイッチングハブを介して相互に結合されている。

**論理ノード**

各論理ノードは1個あるいは複数個のプロセッサ

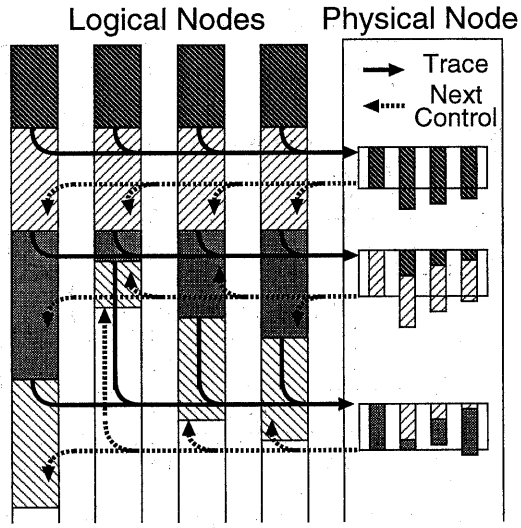


図3　論理／物理ノードのパイプライン動作

を担当し，Lazy Release Consistency に基づく Virtual Shared Memory (LRC-VSM)[4]の技法を用いて，各プロセッサの動作をシミュレートする。なお一つのノードでシミュレートする複数のプロセッサの間でも，LRC-VSM におけるページ単位のコヒーレンス制御を仮想的に行なう。また後述するようにキャッシュの部分的なシミュレーションによって，広義のミスを生じうるメモリアクセスのみを抽出し，その履歴を物理ノードに送出する。

物理ノード

物理ノードは論理ノードから受けとったアクセス履歴に基づき，以下の処理を行なう。

- ネットワークや物理的なメモリの動作タイミングの決定
- 個々のアクセスの実行タイミングの決定
- 個々のアクセスの際のキャッシュ状態の決定

また図3に示すように，論理／物理ノードはパイプライン的に動作する。このパイプライン動作の円滑な進行のために，個々の論理ノードにおけるシミュレーション進行度を制御し，論理ノード相互および論理／物理ノードの間で極端な時刻乖離が生じないようにする。

## 4. アクセス履歴の削減

論理ノードから物理ノードへ送るアクセス履歴をできるだけ小さくするために，論理ノードでは前述のように広義のミスの可能性があるアクセスだけを抽出する。以下，その具体的な方式と正当性について議論するが，より厳密な議論については5)を参照されたい。

### 4.1 論理ノードでのキャッシュ・シミュレーション

論理ノードでは以下のようにコヒーレント・キャッシュを部分的にシミュレートする。なお説明を簡単にするために，write-invalidate プロトコルと $\{\mathbf{M},\mathbf{S},\mathbf{I}\}$ の3状態キャッシュを仮定するが，原理的には他のプロトコルや状態追加にも対応可能である。

(1) 対象システムのキャッシュ $C_t$ が容量 $\gamma_t = \gamma \cdot w$ の $w$-way set associative であるとき，論理ノードでシミュレートするキャッシュ $C_s$ をラインサイズが同一で容量 $\gamma$ の direct map とする。この結果 $C_t$ で容量性あるいは競合性のミスが生じるならば，必ず $C_s$ で容量性ミスが生じる。なおラインサイズは LRC-VSM のページサイズよりも小さく，両者の境界は整合しているものとする。

(2) プロセッサのアクセスによるラインの状態遷移は，$C_t$ と $C_s$ で同一とする。すなわち読出では $\mathbf{I} \Rightarrow \mathbf{S}$，書込では $\{\mathbf{S},\mathbf{I}\} \Rightarrow \mathbf{M}$ と遷移する。

(3) 他のプロセッサのアクセスによるコヒーレンス維持操作では，$C_t$ では通常の状態遷移（読出要求の際に $\mathbf{M} \Rightarrow \mathbf{S}$，書込要求の際に $\{\mathbf{M},\mathbf{S}\} \Rightarrow \mathbf{I}$）を行なう。一方 $C_s$ では，ページの更新情報である diff がプロセッサ $p$ から $q$ へ渡される際に，以下の状態遷移を行なう。

- $p$ では，送信する diff の生成者が $p$ 自身であって，かつその diff について初めての送信である場合，diff に含まれる全てのラインが $\mathbf{M} \Rightarrow \mathbf{S}$ と遷移する。
- $q$ では，diff に含まれる全てのラインが $\{\mathbf{M},\mathbf{S}\} \Rightarrow \mathbf{I}$ と遷移する。

### 4.2 同期的ラインのアクセス履歴

次にラインが同期的であれば，$C_s$ における広義のミスだけをアクセス履歴として残せばよいことを示す。すなわち以下の定義による同期的ラインについては，$C_t$ でコヒーレンス・ミス（$\mathbf{S}$ のラインへの書込を含む）が生じるならば，必ず $C_s$ でもミスが生じることを示す。

**定義3**　少なくとも一方が書込であるような同一ラインに対するアクセス $m$ と $m'$ はライン競合するといい，ライン $L$ に関する任意の競合アクセス $m$ と $m'$ について $m \prec m'$ または $m' \prec m$ であるとき，またその場合に限り，$L$ は同期的であるという。

**定理1**　同期的なライン $L$ に対するアクセス $m$ が $C_t$ でコヒーレンス・ミスを生じるならば，$m$ は $C_s$ でもミスを生じる。

[証明（概略）]　$m'$ を $m$ と同じプロセッサ $p$ による $L$ に対するアクセスであって，プログラム順で $m$ の直前に位置するものとし，$m'$ と $m$ の間には $L$ をリプレースするアクセスは存在しないものとする（存在すれば $C_t$，$C_s$ の双方でミス）。$m$ の実行時におけるキャッシュ $C_x$ のライン $L$ の状態を $state_x(m)$，また $m$ がミスすることを意味する述語を $miss_x(m)$ とすると，$state_s(m) = \mathbf{I}$ であれ

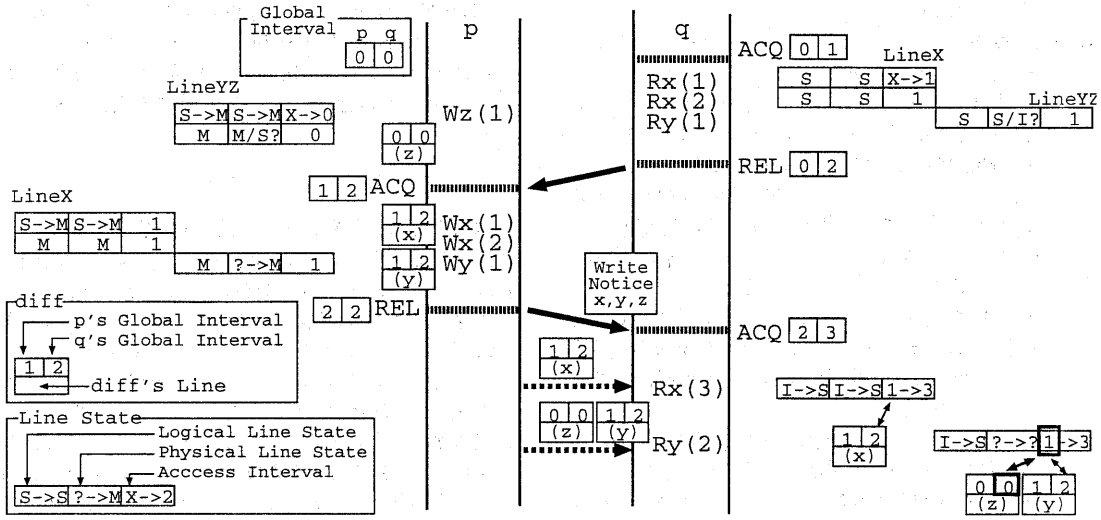


図 4　同期的／非同期的ラインとキャッシュミス

ば明らかに $miss_t(m) \rightarrow miss_s(m)$ である。

$state_s(m) = \mathbf{S}$ である場合，$\neg(miss_t(m) \rightarrow miss_s(m))$ を仮定すると $state_t(m) = \mathbf{I}$ となり，$m'$ と $m$ の間に他プロセッサ $q$ による書込 $m_w$ が生じている。一方 $L$ は同期的であり，かつ対象システムのメモリ・モデルは最弱でも RC であるので，$m' \prec m_w \prec m$ である。したがって $p \neq q$ であることから，LRC-VSM では $m'$ と $m$ の間に $p$ が $m_w$ に関する diff を受理することが保証され，$state_s(m) \Rightarrow \mathbf{S}$ と矛盾する。

$state_s(m) = \mathbf{M}$ である場合についても，同様の議論が成り立つ。

たとえば図 4 の $x$ を含むラインは同期的であるので，$C_t$ でミスが生じる $W_x(1)$ と $R_x(3)$ は，$C_s$ でもミスが生じている。なお図には示していないが，diff には含まれるが結果的に参照されないラインの状態が，$C_s$ では $\mathbf{S}$，$C_t$ では $\mathbf{M}$ となるため，そのラインに再び書込む際に冗長なアクセス履歴が生成される。しかし物理ノードでは，ラインの遠隔参照の有無とそれに伴う状態遷移を正確にシミュレートできるので，このような書込がヒットであることを正しく認識できる。

### 4.3　非同期的ラインの判別

一方，図の $y$ と $z$ を含むラインは非同期的であるので，対象システムでは $W_z(1)$ と $R_y(1)$ の前後関係によって，$R_y(1)$ のヒット／ミスが定まる。しかし論理ノードでは，$W_z$ に関する diff の送受信が行なわれていないため，$R_y(1)$ はヒットと判断される。したがって，$R_y(1)$ のアクセス履歴を残さないとすると，$W_z(1)$ が対象システムにおいて時間的に先行した場合に，$R_y(1)$ によって生じる読出要求が欠落してしまう。また同様の問題が $W_y(1)$ についても生じる。

この問題を解決するために，論理ノードは以下の方法によってラインが同期的であるか否かを判別する。

(1) 4) と同様に，同期操作のたびに増加するインターバルと，その大域的な推定値である**大域インターバル**☆を各プロセッサごとに用意する。プロセッサ $p$ が持つ大域インターバル $T_p$ のプロセッサ $q$ に関する値，すなわち $p$ が推定する $q$ のインターバルを $\tau_p^q$ とする。一般に $q$ による release に直接または間接的に対応する acquire を $p$ が行なうと，release 時点での $\tau_q^q$ が $\tau_p^q$ に反映される。

(2) プロセッサ $p$ が diff $\Delta$ を生成する際に，その時点での $T_p$ を $\Delta$ に付加する。$\Delta$ に付加された $\tau_p^q$ の値を $\tau_q(\Delta)$ とする。

(3) プロセッサ $q$ が保有するページの全てのラインについて，アクセスごとに $q$ 自身のインターバル $\tau_q^q$ を記録するフィールドを設ける。$q$ が保有するライン $L$ のアクセス・インターバルを $\tau_q(L)$ とする。

(4) プロセッサ $p$ が生成した diff $\Delta$ をプロセッサ $q$ が受理した際に，$\Delta$ に含まれる全てのライン $L$ について，$\tau_q(L)$ と $\tau_q(\Delta)$ を比較する。$\tau_q(L) \geq \tau_q(\Delta)$ であれば，$L$ を非同期的であるとマークする。

たとえば図 4 では，$z$ に関する diff $\Delta_z$ に付加された大域インターバル中の $\tau_q(\Delta_z)$ は 0 であるのに対し，$y$ と $z$ を含むライン $L_{yz}$ の $q$ でのアクセス・インターバル $\tau_q(L_{yz})$ は 1 である。この結果 $L_{yz}$ が非同期的であることが判明する。

この方法により，全ての非同期的ラインが発見できることは，以下のように証明できる。

**定理 2**　プログラムの実行完了時に，どのプロセッサにおいても非同期マークが付いていないラインは

☆ 4) では vector timestamp と呼ばれている。

同期的である。

[証明（概略）] ライン $L$ に対するプロセッサ $p$ による書込アクセスを $m$ とし，$q$ によるアクセスを $m'$ とする。また $m$ に関して $p$ が生成する diff を $\Delta(m)$，$\Delta(m)$ を $q$ が受理した時点での $\tau_q^q$ を $\tau_q^m$，その時点での $\tau_q(L)$ を $\tau_q^m(L)$ とする。

$L$ にはプログラムの実行完了時に非同期マークが付いていないので，任意の $m$ について $\tau_q^m(L) < \tau_q(\Delta(m))$ である。したがって $m'$ を $q$ が実行した時点でのインターバル $\tau_q^q$ を $\tau(m')$ とすると，$\tau(m') \leq \tau_q^m(L) < \tau_q(\Delta(m))$ であるか（$\Delta(m)$ 受理時には $m'$ を実行済），あるいは $\tau(m') \geq \tau_q^m$ であるか（$\Delta(m)$ 受理時には $m'$ を未実行）のいずれかである。

$\tau(m') \leq \tau_q^m(L) < \tau_q(\Delta(m))$ であれば，$m'$ の実行後に行なわれた同期操作によって $p$ に $q$ のインターバルが伝わったのであるから，$m' \prec m$ である。一方 $\tau(m') \geq \tau_q^m$ であれば，インターバル $\tau_q^m$ の開始あるいはそれ以前の同期操作によって $q$ は $p$ による書込 $m$ を伝達されたのであるから，$m \prec m'$ である。したがって定義 3 により $L$ は同期的である。

以上のようにプログラムを一度実行すれば非同期的ラインを全て特定できるので，シミュレーションを以下の 2 フェーズに分けて，必要なアクセス履歴を生成する。

**フェーズ 1** 論理ノードは，全てのラインは同期的ラインであるとの仮定でアクセス履歴を生成しながら，非同期的ラインのチェックとマーキングを行なう。このフェーズで非同期的ラインが発見されなければ必要なアクセス履歴が得られているので，シミュレーションは完了する。一方，非同期的ラインを発見した論理ノードは，その旨を他の全ての論理ノードと物理ノードに伝えるとともに，アクセス履歴の生成を中止して非同期的ラインのマーキングだけを行なう。非同期的ラインの発見を通知された論理ノードも同様に非同期的ラインのマーキングに専念し，物理ノードはシミュレーションを中止する。

**フェーズ 2** フェーズ 1 が完了して全ての非同期的ラインが発見されると，プログラムを再実行する。このフェーズでは非同期的ラインのチェックやマーキングは不要であり，必要なアクセス履歴の生成が行なわれる。すなわち同期的ラインについては広義のミスのみ，また非同期的ラインについては全てのアクセスについて履歴を生成する。物理ノードはこの履歴に基づいて物理的シミュレーションを行なう。

なお論理ノードのメモリに余裕がある場合，フェーズ 1 で同期操作の際に送受されるページの無効化情報を全て保存しておくことができる。この結果，フェーズ 2 では同期操作をノード間通信をせずに行えるため，2 フェーズ実行の損失を緩和することができる。また，さらに余裕があって diff も全て保存できる場合には，フェーズ 2 での論理ノード間の通信を完全に排除できる。

## 5. おわりに

本報告では，共有メモリ型並列計算機を対象とする高速な分散シミュレータの構築するために，論理的／物理的シミュレーションを分離する手法を提案した。この手法の最大の特徴は，複数の論理ノードによる分散シミュレーションにおいてコヒーレント・キャッシュを部分的にシミュレートし，広義のキャッシュミスを生じる可能性のあるアクセスだけを物理ノードに伝達することにある。このアクセス履歴の削減方式は LRC-VSM に軽微な拡張を施すことによって実装でき，その正当性も証明されている。また論理／物理ノードのパイプライン的並行実行により，アクセス履歴保存の手間を除去しつつ，シミュレーション時間を短縮できることも，我々の手法の特徴である。

本手法に基づくシミュレータは現在開発中であり，その完了と性能評価，特にアクセス履歴の削減効果を評価することが最大の課題である。また out-of-order 実行のプロセッサに対応したアクセス遅延の近似方法や，物理的シミュレーションの分散化も，今後検討していく予定である。

**謝辞** 本研究の一部は並列・分散処理研究機構の研究課題「並列分散処理アーキテクチャ技術—超並列共有メモリ型マルチプロセッサの研究」による。

## 参考文献

1) 今福茂, 大野和彦, 中島浩: 共有メモリ型並列計算機シミュレータの実現, 情報処理学会研究報告, 99-ARC-134, pp. 37–42 (1999).

2) Jefferson, D.R.: Virtual Time, *ACM Trans. on Programming Languages and Systems*, Vol. 7, No. 3, pp. 404–425 (1985).

3) Boku, T., Mishima, M. and Itakura, K.: VIPPES : A Virtual Parallel Processing System Simulation Environment, *High Performance Computing Asia*, pp. 843–853 (1998).

4) Keleher, P.: *Lazy Release Consistency for Distributed Shared Memory*, PhD Thesis, Department of Computer Science, Rice University (1994).

5) 中島浩, 今福茂: 共有メモリの分散シミュレーションにおけるアクセス履歴の削減方式, http://www.para.tutics.tut.ac.jp/~nakasima/papers/accred.ps.gz (1999).