計算機アーキテクチャ 134－7
（1999．8．2）

共有メモリ型並列計算機シミュレータの実現

今福 茂　大野 和彦　中島 浩

豊橋技術科学大学 工学部 並列処理研究室
E-mail: {shige, ohno, nakasima}@para.tutics.tut.ac.jp

並列計算機の高速化アイディアを特定の実装に因われずに検証する手段としてシミュレーションは有効な手法である。しかし、逐次環境下ではシミュレーション自体に長い時間を要する。その高速化要求を満たす手法として並列シミュレーションがあるが、同期イベントやメッセージ通信を忠実に再現するとシミュレータ上でのメッセージが多発しボトルネックとなる。我々はバリア同期や排他制御などのソフトウェア的な同期イベントのみをシミュレータの同期イベントとして扱う同期イベント削減手法を提案し、並列シミュレータ開発の第一ステップとして、同期イベント削減手法を用いた逐次シミュレータを試作し、最大24%の速度向上を確認した。

A parallel machine simulator with reduction of synchronization event.

Shigeru IMAFUKU　Kazuhiko OHNO　Hiroshi NAKASHIMA

Department of Computer Science, Toyohashi University of Technology

Simulation of a large computer system such as a parallel machine takes much computation time. Parallelization will be a good solution, but it cause a serious performance bottleneck if we directly map inter-processor hardware events for synchronizations onto those of parallel simulation. Our idea for efficient parallelization is to map software level synchronization events, such as barriers and mutexes, to simulator's events to avoid the bottleneck. This idea is also applicable to a sequential simulator because it reduce the frequency of the context switch. In this research, we implemented a sequential simulator with the event reduction method. We evaluated its performance and compare it with a direct event mapping. As the result, the event reduction achieves about 24% speed up proving its efficiency.

1　はじめに

並列計算機の高速化アイディアを汎用シミュレータシステムを用いて検証することは多々行なわれてきており、高機能な汎用シミュレータについても既に様々な研究がなされている。だが、プロセッサ自体にそのアイディアの中核となる機能を付加したい場合などは、新たなシミュレータシステムが必要となる。

また、並列計算機のような大規模システムにおいては、シミュレーション時間短縮のために並列シミュレーションが用いられ、研究も盛んに行なわれている。しかし、アーキテクチャレベルのシミュレーションに対応した並列シミュレータの研究はこれからである。

並列シミュレーションにおいて高速性を得るためには、シミュレータ内の処理をなるべく独立して進行させなければならない。これはシミュレータ内の同期メッセージを削減していくことに他ならない。

そこで、アーキテクチャレベルの特性を生かした新たな並列シミュレーション手法として、複数のプロセッサが生成する同期メッセージをできるだけ削減する「同期イベント削減手法」を提案し、これを実装した逐次シミュレータの試作及びその評価を行なった。

本論文では、3章でシミュレータのモデルを示した後、同期イベント削減手法について4章で提案を行ない、実装方法を交えて内容及び物理事象の再現方法について解説し、5章では同期イベント削減手法及び高速化の効果を示すために実行時間比較及び考察を行ない、6章で本研究の成果についてまとめる。

2　背景と目的

並列計算機における高速化アイディア検証の手段としてシミュレーションは有効である。しかし、並列計算機のような大規模システムを扱う場合その実行時間は大きな問題となる。

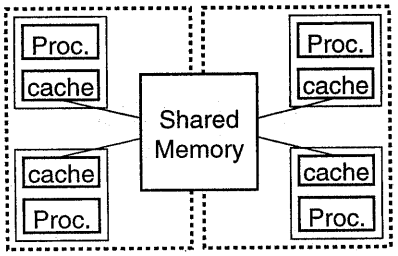

図 1: 対象共有メモリ並列計算機のモデル

こういった高速化要求に対しては並列シミュレーション化が有効であるが、単純に並列化しただけではメッセージ通信のオーバーヘッド等により高速化は望めない。このため、オーバーヘッドとなるメッセージ通信を極力排除した新たなアーキテクチャの並列シミュレータが求められている。

また、ハードウェア研究者の間では、汎用クロックトレース型シミュレータを利用したり、既存のプロセッサシミュレータやネットワークシミュレータ等を組み合わせることで並列シミュレーションを実現している場合がある。これらの方法は比較的容易に高レベルなシミュレータを構築できる反面、高機能ゆえに処理自体が重い場合がある。また組み合わせるシミュレータの仕様によって制限が生じる場合がある。例えば対象プログラムとして、専用ライブラリを用いての再コンパイルや独自形式を要求するシミュレータもあり、前処理が繁雑となる。

そこで本研究では スレッド関数によって対象プログラムの同期点を特定し、同期点までのシミュレータ上のメッセージ通信を削減する手法を考案し、高速性を引き出している。また、対象プログラムとして通常の実行形式をそのまま用いることを可能とし、特殊な前処理を必要としない並列シミュレータを実現している。

現在並列シミュレータは実装中であるため、本稿ではその中核となる同期イベント削減手法の実現及び逐次シミュレータでの評価について述べる。

3 対象システム及び並列シミュレータのモデル

シミュレータが対象とするシステムは 図1 のような共有メモリ型並列計算機であり、1つの共有メモリを持ち、キャッシュ容量やその方式、プロセッサ数や接続方式などは任意である。

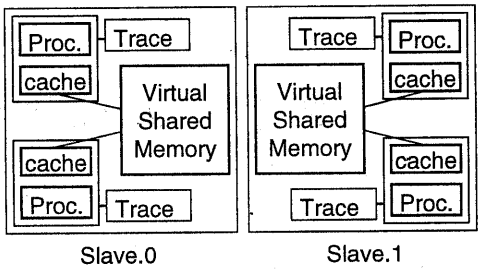

図 2: スレーブプロセスのモデル

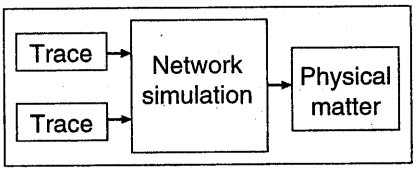

図 3: マスタプロセスのモデル

この対象システムを並列シミュレーションする場合、図1の点線のように分割し、点線部分を 図2のようなスレーブプロセス (以下スレーブ) として割り付け、それらスレーブを起動し管理する図3のようなマスタプロセス (以下マスタ) を用いて構成する。

スレーブ内のプロセッサごとに4.4章で述べる物理事象再現に用いるトレース情報を生成し、同期点ごとにマスタ及び他のスレーブに送信する。マスタはスレーブからのトレース情報を受け取りネットワークを考慮した物理事象の再現を行なう。各スレーブでは受信したトレース情報をもとに仮想共有メモリの補完をすることで同期点ごとに一貫性を保つ処理を行なっている。

マスタでは物理事象の整合を取り、スレーブでは論理事象の整合を取る処理を行なうが、これらは独立しており、スレーブはマスタの物理事象再現処理の完了を待たずにシミュレーションを再開することができる。このように物理と論理の整合処理をオーバーラップすることで並列シミュレーションを高速化している。

実行環境として PC クラスタを用いた分散メモリ環境を想定しており、マスタ、スレーブを1計算機にマッピングして並列シミュレーションを行なう。トレース情報送受信などのマスタ・スレーブ間の通信には PVM ライブラリを用いている。

現在、並列シミュレータは実装中であるため、本稿では試作した逐次シミュレータについて評価を

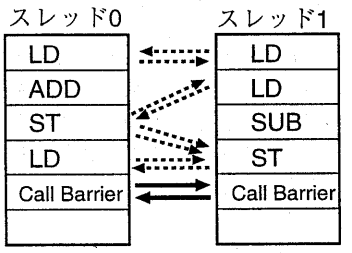

図 4: 同期イベント削減

行なった。逐次シミュレータはマスタとスレーブの機能をそれぞれを 1 つずつ持つ単一プロセスとして作成した。

4 同期イベント削減手法

4.1 提案

並列計算機の並列シミュレーション高速化にはシミュレータ上での通信頻度を低く抑えることが重要である。通信を発生させる同期イベントは以下のように分類できる。

- ソフトウェア的同期イベント
  ⇒ バリア同期や排他制御など
- ハードウェア的同期イベント
  ⇒ 対象システムでの共有メモリアクセスなど

これらの同期イベント全てを忠実にシミュレーションせずとも、トレース駆動シミュレータのように、発生したであろう事象について後から辻褄合わせを行うことは可能である。よってハードウェア的同期イベントはシミュレータの同期メッセージの対象から外すことができる。

本手法により、図 4のようにハードウェア同期メッセージ (点線矢印) を削除することで同期イベントが大幅に削減され、ソフトウェア同期点までは各スレッドの独立性が増すため並列シミュレータへの実装に適している。

また、多数のプロセッサを少数のプロセッサでシミュレートする場合、シミュレートしているプロセッサの「切替え」が発生するが、同期イベント削減手法によりその頻度を少なくすることができる。その結果、シミュレータ上の対象プロセッサを直線的に動作でき、処理の局所性用いた以下のような逐次部分の高速化が望める。

- シミュレータのコード自体の命令キャッシュヒット率が向上する。
- シミュレータ上のプロセッサで使用頻度の高い変数をローカル変数にコピーすることでアクセスに要するオーバーヘッドを削減できる。

4.2 実装方法

本章では同期イベント削減手法について具体的な内容について述べる。本手法はスレッドを用いた並列ソフトウェアを対象にしており、アーキテクチャや OS などを特に限定していない。しかし、実装環境のスレッド関数名を利用する等、実装に深く関わっており、今回実装を行なった SPARC アーキテクチャVersion8 [1] 上の POSIX スレッド [2] [3]、特に Solaris スレッド [4] 関数を例にとって説明する。

4.2.1 ソフトウェア同期点の取得

ソフトウェア同期はプログラム中に**スレッド関連のシステム関数**として現れる。

例えば バリア同期は 図 4の `CALL barrier` のようなユーザー定義の関数呼び出しであり、内部では `mutex_lock()` 及び `mutex_unlock()` といった排他制御を行なうシステム関数で構成されている。つまり、ソフトウェア同期点を特定することは、スレッド関連のシステム関数を特定することである。

4.2.2 システム関数の特定

しかし、これらシステム関数とユーザーが作成した関数も実行ファイル中では、どちらも「`CALL xxx` 番地 」となっておりバイナリコードのみからは両者を判別することはできない。

そこで、解決アプローチとして対象プログラムに手を加える手法と手を加えない手法が考えられる。

対象プログラムを読み込むタイプのシミュレータの場合、専用ライブラリを用いてコンパイルしたオブジェクトコードを用いる *SimpleScalar* [5] や、シミュレータ上で並列化させる部分に特殊コードを埋め込むタイプなどが考えられるが、このような対象プログラムに手を加えるタイプではシミュレーションの前処理が煩雑となる。

本シミュレータでは実行ファイル中に含まれる関数の飛び先番地と関数名が対になったシンボルデバッグ情報を利用して関数名を特定し、対象プログラムに手を加える事なくそのまま読込むことを可能とした。

4.2.3 シンボルデバッグ情報による関数の特定

実行形式に含まれているシンボルデバッグ情報とは、CALL 命令の飛び先番地 と 飛び先関数名の文字列が対となったテーブル形式の情報である。このテーブルから飛び先番地をキーとして関数名を特定しシステム関数名とマッチしたならば、シミュレータが区別できるように疑似的な命令 PSEUDO に置換しておく。

SPARC アーキテクチャの特性として遅延分岐が存在し、関数呼び出しの前に実行されるべき命令が挿入されているので疑似命令と遅延スロットの命令を入れ換える操作が必要になる。

4.2.4 プリデコード

シミュレータでは、フェッチ ⇒ デコード ⇒ 実行 といったフェーズを経るわけであるが、フェッチ、デコード時に動的に文字列テーブルサーチや比較を行なうと CALL 命令の度に高コストの操作が挿入され効率が悪い。そこで、シミュレータが実行ファイルを読み込み解析する時点で あらかじめ全バイナリコード部分に対し命令中に含まれる、ソースレジスタ、ターゲットレジスタ、即値などを分離し、シミュレータの扱いやすい構造体配列に格納するプリデコードをかけ、同時に CALL 命令の疑似命令への置換操作を行うことで、効率低下を防いでいる。

また、プリデコードを行うには、自己書き換えする対象プログラムに対して特別な配慮を要するが、自己書き換えプログラム自体が希なこともあり、対策は今後の課題とした。

4.3 システム関数のエミュレート

これまではシステム関数とはスレッド関連のものだけを指していたが、ユーザー定義の関数以外の全ての関数についてもシミュレータ側がフックし、その処理を行わなければならない。

つまり、C 言語の提供する標準関数である、printf や atoi 等についても同様のテーブルサーチ、文字列比較を行なって疑似命令に置換し、エミュレーションを行なう必要がある。

現在の実装ではスレッド関連の関数と、標準関数の一部をエミュレートしている。

4.3.1 エミュレーション手法

基本的に関数呼び出しは必要な引数をレジスタを介して受け取り、また関数ごとに必要とするレジスタ数は取得可能であるので疑似命令が現れた時点でプロセッサのレジスタ変数から引数の値を読み取ることは容易である。

引数が得られれば、その関数がすべき内容をシミュレータが行なえば良い。例えば、スレッドを生成する関数 thr_create() ならば、シミュレータプロセッサの PC を引数である開始関数のアドレスに設定したり、レジスタを収めたプロセッサ構造体を 適切な値に設定してやり、スケジューリング対象に入れてやれば良い。

このようにして、シミュレータは関数をフックし、そのエミュレーションを可能としているので、新規ハードウェアをシミュレーションする場合でも詳細なドライバを構築する必要がない。

また本手法の問題点として、実際の関数内では飛び先のライブラリ内でさらに数百の命令を実行した後ユーザープログラムに復帰するが、シミュレータでは関数をフックした時点でエミュレーションしてしまい、エミュレーション分のクロック経過の再現が行われていないことが挙げられる。現在の実装では、エミュレートした関数ごとの大まかな固定数を経過クロックに加えているだけだが、動的な経過クロック数を考慮することは今後の研究課題としたい。

4.4 物理事象の再現

同期イベント削減手法は、ハードウェア的同期イベントを一旦無視し、同期点までシミュレートすることで論理的結果には影響を与えることなく高速性を引き出す手法である。したがって、無視したハードウェア的同期イベントがもたらす物理的な事象を再現する必要がある。

物理事象の再現性が高いシミュレータとしてクロックトレース型シミュレータがある。これは全てのプロセッサ上での全事象のログをとった後、それらログをつき合わせて物理事象の再現を行っている。この手法の欠点として、ログの保存に要する資源量とその資源のアクセス速度が大きく関係するためあまり高速化は望めないことが考えられる。

そこで本研究では、シミュレータが同期点を特定できることを利用して、プロセッサ上の同期点までを１ブロックとし、ブロック内で発生したメモリアクセスの履歴をトレース情報として取得し、

マスタ上でブロック毎の物理事象を再現する手法を考案した。この手法により、プログラム終了まで全トレース情報を保持しておく必要がない。

4.4.1 トレース情報

スレーブによって生成されるトレース情報は以下の内容を持つ固定エントリ数の構造体配列であり、同期点に到達した場合、あるいはエントリを全て使い切った場合にマスタに送信している。

発行クロック	対象アドレス	R/W BIT

同期点のみを送信のトリガとすると、トレース情報の必要エントリ数は実行時のみ確定し実行前は不確定である。不確定数の情報量に対応する為、構造体リストとして実装しなければならず、エントリを増すごとにメモリ確保作業や使用後の解放作業が必要となり効率が悪い。そこでトレース情報を一定数固定のエントリ数とし、送信トリガにエントリが尽きた場合も加えることで、動的メモリ確保及び解放作業を解消した。現在の実装では1つのプロセッサ当たり20000エントリ分確保しているが、最適なエントリ数を求めることについては対象プログラムの同期間隔を考慮する必要があり、今後の研究課題としたい。

4.4.2 スレーブ：トレース情報生成

スレーブはプロセッサで発行された全てのロード／ストア系の命令に対し、その発行クロックと対象アドレス及びReadまたはWriteを示すBITについてのトレース情報の蓄積を行なう。同期点にたどりついたプロセッサは他スレーブ及びマスタへ取得したトレース情報を送信する。

またスレーブではキャッシュのシミュレーションを行なう必要がなく、発行された全ての命令は1クロックで完了したものとして経過クロックを加算してゆく。これは、マスタがトレース情報からネットワーク及びキャッシュを考慮して物理事象を再現するに当たって、スレーブが渡すべき情報はどの順番でどの番地へメモリアクセスを行なったか示すもののみでよいからである。

4.4.3 マスタ：物理事象再現

マスタで行なわれる物理事象再現の方法について現在実装しているバス結合ネットワークを例にとって説明する。

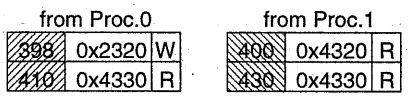

図5: proc.0,proc.1からのトレース情報の例

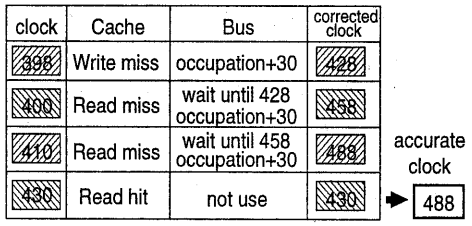

図6: 物理事象再現

図5のように、各スレーブのプロセッサ0と1から送られてきたトレース情報をクロック数の小さいものから取り出し、キャッシュにヒットしたかミスしたかを調べる。

キャッシュミスした場合はバスを介して共有メモリアクセスが発生するのでバスが空くまでのwait値とバスアクセスのレイテンシ(ここでは30clock)を加算して補正したクロック値を求めることができる。キャッシュヒットした場合はバスを利用しないので補正はかからない。このようにして物理事象を再現した最終的な終了クロック値を得ることができる。

5 予備性能評価

5.1 テストプログラム

long変数 64 × 64 の正方行列について20乗($R = S^{20}$)を求める。これをR及びSを4つのスレッドで横方向に均等分割し並列演算を行なう。なお、RにSを掛け合わせる毎にSolarisスレッド関数群を利用したバリア同期関数によって書き込み保護を目的として2回の同期とっている。親スレッドと演算担当の子スレッドで計5プロセッサをシミュレートし、それぞれキャッシュ容量は64KB、ダイレクトマップ方式、トレース情報の格納数は20000エントリとして、実行時間を計測し本手法の効果を検証する。

- SPARCStation5/110 上で実行
- SPARCStation5/110 上で実行した「1命令毎に同期」版シミュレータ

	実行時間	速度向上
SPARCStation5/110	3.30(秒)	——
1命令毎に同期	293.8	——
同期イベント削減版	236.7	24(%)

表 1: 実行時間及び速度向上

- SPARCStation5/110 上で実行した「同期イベント削減」版シミュレータ

1命令毎に同期するシミュレータはスレーブ上のプロセッサを1命令シミュレートするごとに切替えるものである。

5.2 比較結果

同期イベント削減手法を用い逐次部分に対して局所性を利用した高速化を施すことで、各プロセッサを同期点まで直線的に動作することができ、1命令ごとに同期を行なうシミュレータに対し約24%の高速化を達成した。

またこのシミュレータはSPARCStation5/110上で実行した場合の約72倍の時間でシミュレート可能であることを確認した。

6 おわりに

6.1 研究の成果

本稿では同期イベントをできるだけ削減できるような逐次シミュレータの試作について、スレッド関数を用いた並列プログラムに対しソフトウェア同期イベントのみをシミュレータの同期イベントとする同期イベント削減手法の提案を行なった。

SPARCアーキテクチャVersion8及びSolaris／POSIXスレッドに対して実装を行ない、提案した同期イベント削減手法が逐次シミュレータにおいても有効であること、また局所性を用いた逐次シミュレータの高速化についても有効であることを確認した。

現在、これら成果をもって並列シミュレータを実装中である。

6.2 今後の課題

今回の評価で用いた評価プログラムは小規模なものであったので、NAS Parallel Benchmarks 等を用いて高負荷状況下での性能評価を行ない、より信頼性のある評価を行いたい。

物理事象再現レベルの向上として、現在の実装がバス結合のネットワークのみシミュレート可能であるのを、標準的な並列計算機の結合方式であるメッシュネットワークについても行なえるように拡張したい。あるいは、ネットワーク構成入力型 [6] にすることで、より汎用性を増すことも検討したい。

Out of order 実行機構や階層的キャッシュ機構などは未実装であるが、Out of order 実行については同期点ごとに実際の命令発行の順番を再現するにはメモリアクセスだけを取得する現在のトレース情報だけでは不足しており、命令発行情報もトレースする必要があり情報量が増してしまうため、情報量を抑えつつどのように実装していくか検討することなどが考えられる。

参考文献

[1] 相越克久, 田中長光＝訳, 多田好克＝監訳: SPARC アーキテクチャ・マニュアルバージョン8, トッパン (1992).

[2] Northrup, C. J.: *Programming with UNIX Threads*, Johon Wiley & Sons (1996).

[3] Nichols, B., Burrlar, D., Farrel, J. P., 榊 正憲訳: Pthread プログラミング, オーム社 (1998).

[4] Cantanzaro and B.J: *The SPARC technical papers*, Springer (1991).

[5] T.Austin, D. and S.Bennett: Evaluating future microprocessors: The simplescalar toolset. Techinical Report CS-TR-96-1308 (July 1996).

[6] Boku, T., Harada, T., Sone, T., Nakamura, H. and Nakazawa, K.: INSPIRE : A general purpose network simulator generating system for massively parallel processors, *Proceedings of PERMEAN'95, pp.24-33* (1995).

[7] Pai, V. S., Ranganathan, P. and Adve, S. V.: RSIM: An Execution-Driven Simulator for ILP-Based Shared-Memory Multiprocessors and Uniprocessors, *Proceedings of the Third Workshop on Computer Architecture Education* (February 1997).

[8] Murthy Durbhakula, V. S. P. and Adve, S. V.: Improving the Accuracy vs. Speed Tradeoff for Simulating Shared-Memory Multiprocessors with ILP Processors, *HPCA 1999* (1999).