同期操作に対するメモリアクセスの投機的実行の提案

佐藤 貴之　　中島 浩
豊橋技術科学大学 情報工学系

マルチプロセッサ環境下において並列処理を試みる場合、並列処理に関する依存関係の充足を保証するための同期操作は重要である。また共有メモリシステムにおいては、同期操作によるメモリアクセス順序の制約が、メモリのコンシステンシィと密接に関わってくる。

ところが、同期操作と順序制約の維持に関するコストは高く、これらの操作が実際に不要である場合でも隠蔽はできず、オーバーヘッドの大きな要因となることは確かである。

そこで、同期操作の完了後に行われるべきメモリアクセスを、同期の成立を仮定して投機的に実行することにより、同期操作とメモリアクセス順序制約に要するコストを隠蔽するような共有メモリ機構を提案する。

Proposal of speculative execution of memory accesses following synchronization

Takayuki Satou　　Hiroshi Nakasima
Department of Information and Computer Sciences,
Toyohashi University of Technology

Synchronization to satisfy the dependence between parallel processes is one of the most important and frequently used operations for parallel processing. In shared memory systems, the order of memory accesses is tightly related to synchronization to preserve the memory consistency.

The cost of synchronization, however, is not only significantly high but also hardly hidden in conventional schemes even if its completion is not necessary to be confirmed.

In order to hide the cost, this paper proposes a shared memory mechanism in which memory accesses are speculatively executed supposing that the synchronization preceding them is completed.

1. はじめに

ある条件に依存して処理の流れが定まるようなプログラムにおいて、条件の成立（あるいは不成立）を予測した上で、条件が確定する以前に条件に依存する処理を行なう投機的実行は、プログラムの高速処理に重要な役割を果たす。たとえばスーパスカラーなどにおける分岐予測に基づく投機的実行はよく知られた例であり、制御依存による先行制約を緩和して命令レベルの並列度を高くすることができる。

一般にマルチプロセッサ環境下において並列処理を試みる場合、あるプロセッサが生成する条件に他のプロセッサの処理が依存する場合に，その依存関係を充足する操作、すなわち同期操作のコストが高いため、投機的実行が果たす役割は極めて大きい。

一方、共有メモリシステムにおいては、メモリのコンシステンシィ・モデルとそれを充足するための操作の順序制約が密接に関わっており、メモリ・アクセスの効率化のためには、この順序制約を緩和することが必要である。というのも、同期操作や順序制約維持が実質的に不要である場合に、これらのコストの隠蔽はできず、並列処理におけるオーバヘッドの大きな要因となるからである。

そこで、同期操作の完了後に行なわれるべきメモリ・アクセスを、同期の成立を仮定して投機的に実行することにより、同期操作／順序制約維持に要するコストを隠蔽する方式を提案し、その実装方法を検討する。

2. 設計方針

2.1 投機的実行

投機的実行によりコストを隠蔽することができる同期操作の例として、まず図 1における共有変数に関する依存制約を考える。

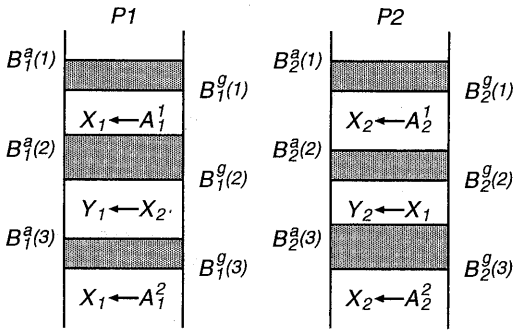

図1 バリア同期による共有変数の依存制約充足

この例では、二つのプロセッサ P_1と P_2が共有変数 X_1と X_2を介した通信を伴う処理を行なっており、その更新／参照順序をバリア同期によって規定することにより、依存関係を正しく保っている。すなわち、P_1は二つのバリア同期操作 $B_1^a(1)$と $B_1^a(2)$にはさまれた区間（以下バリア区間 $B(1)$とする）で X_1を更新し、それを P_2がバリア区間 $B(2)$で参照することにより、X_1に関するデータ依存制約が満たされ、Y_2の値が A_1^1となることが保証される。また P_1は X_1を再度更新して A_1^2とするが、その更新は $B(3)$で行なわれるため、$B(2)$での P_2の参照結果が A_1^1であること、すなわち X_1に関する逆依存制約も満たされる。同様に P_2による X_2の更新と、P_1による参照についても、データ依存／逆依存制約が満たされる。

このように同期による依存関係の充足を説明してきたが、同期が成立したことを知るまでには一定の時間を要し、図に示される $B_i^a(k)$から同期成立確認 $(B_i^g(k))$までの期間（図で濃い影を施した区間）は、アイドル状態となる。さらに、バリア区間に差が生じるとプロセッサのアイドル時間に影響してくる。一般に、プロセッサ数の増加に伴ってアイドル時間も増加してしまう。なお、これらの例では線状バリア[1]を用いているが、面状バリアであってもバリアの「入口」から「出口」の間に同期コストを隠蔽するだけの「局所的」処理がなければ、同様のアイドル期間が生じる。

ところが実際には、共有変数 X_iの更新／参照の時間間隔が十分大きく、かつ P_1と P_2の「歩調」がある程度揃っているならば、同期成立確認以前に X_iをアクセスしても正しい結果が得られる。歩調が完全に揃っておらず、図2のようにバリア区間の長さや X_iのアクセス・タイミングにズレがあっても、更新／参照の正しい順序関係が守られる範囲であれば、同期操作 $B_i^a(k)$の時点で実質的に同期が成立しており、同期成立確認 $B_i^g(k)$以前の区間（図の薄い影を施した区間）に X_iをアクセスしても構わない。

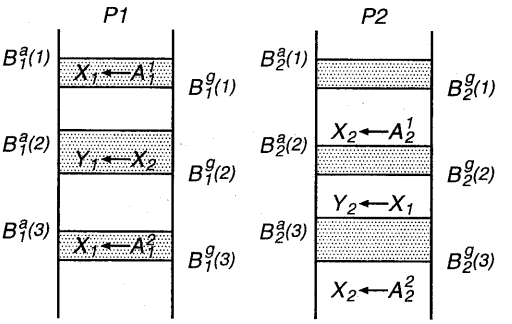

図2 メモリアクセスの投機的実行

そこで、未確認ではあるが同期が成立していること、あるいは非成立ではあるが正しいアクセス順序が保たれることを期待して、同期成立確認以前にメモリ・アクセスを含む処理を投機的に実行することが基本的な提案である。$B_i^a(k)$から $B_i^g(k)$までの区間（バリア投機区間）で共有変数のアクセス等の「大域的」処理を含む任意の操作をを行なえるようすれば、バリア区間が十分に長くかつ投機が成功する限り同期操作のコストを完全に隠蔽することができる。

2.2 投機失敗への対処

2.2.1 不正参照の検知

投機区間での共有変数アクセスはあくまで投機的であり、共有変数の依存制約を満たさないような不正なアクセスが生じた場合には、それを検知して投機的に実行してしまったアクセスに起因するあらゆる計算状態の変化を無効化しなければならない。

例えば図3では、P_1がバリア区間 $B(2)$で X_2の参照を投機的に行ない、その後に本来先行すべき P_2による X_2の更新が行われている。この結果、Y_1には A_2^1ではない不正な値が代入され、さらにその値が W_1に伝搬している。したがって、操作 $Y_1 \leftarrow X_2$に時間的に後続してしまった操作 $X_2 \leftarrow A_2^1$により、$Y_1 \leftarrow X_2$が不正な参照であったことを検知しなければならない。

またこの不正参照の検知は、検知の契機となる更新 $X_2 \leftarrow A_2^1$を行った P_2ではなく、不正な計算を行ってしまった P_1で成されなければならない。そのため P_1では、$Y_1 \leftarrow X_2$が $B_1^g(2)$に先行して

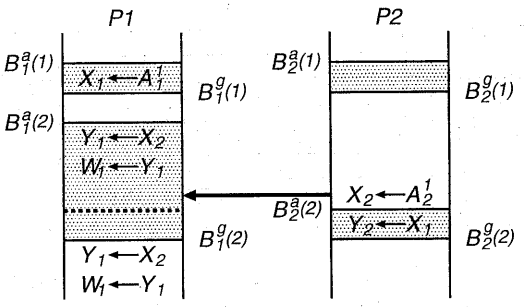

図3　投機的参照の失敗

いるため、潜在的に危険な参照であることを記憶し、$B_1^g(2)$ 以前に X_2が更新されたならば、不正参照を行っていたと認識できるようにしなければならない。

次に不正参照を検知したならば、その参照に起因する全ての計算を無効化しなければならない。この無効化を比較的単純に行う方法としては、計算状態をバリア投機区間の開始時 ($B_1^a(2)$) へ戻すロールバックが考えられる。ロールバックを行なうためには、$B_1^a(2)$ での計算状態がメモリを含めて全て保存されていなければならない。

2.2.2 不正更新の検知

次に、逆依存制約充足の問題について考える。例えば図4では、P_1がバリア区間 $B(3)$ で X_1の更新を投機的に行ない、その後に本来先行すべき P_2による X_1の参照が行なわれている。

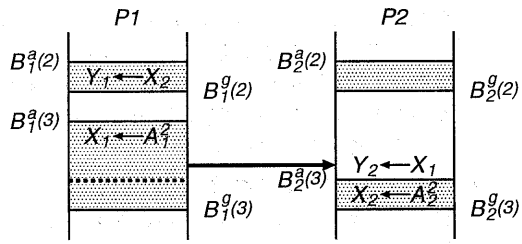

図4　投機的更新の失敗

この不正更新の検知も不正参照と同様に、更新を行った P_1で成されなければならない。すなわち、P_1は $X_1 \leftarrow A_1^2$が $B_1^g(3)$ に先行していることから、潜在的に危険な更新であることを記憶し、$B_1^g(3)$ 以前に X_1が P_2から参照されると、不正更新を行っていたことを認識できなければならない。

一方、不正更新を検知した際の対応は不正参照の検知とは異なり、必ずしも $B_1^a(3)$ へのロールバックを行う必要はない。すなわち、$B_1^a(3)$ での X_1の値は、不正参照によるロールバックに備えて何らかの形で保存されており、この値を P_2による参照値とすることができればよい。

3. 実装モデル

前節で示した投機的メモリアクセスを実現する方法として、本節ではキャッシュを用いた実装モデルを示す。

このモデルでは、潜在的に危険な参照／更新を行ったことをキャッシュの状態の一つとして記憶し、不正アクセスを検知できるようにする。また、適切な状態遷移を定義して、投機の開始、成功、失敗に対処する。

なお、ロールバックを実現するための計算状態の保存は、レジスタについてはシャドウ・レジスタ[2]、あるいはそれに類似した機構を用いるものとする。一方メモリに関する状態保存は、キャッシュとそのバックアップ・メモリを用いて行う。すなわち、投機区間でのメモリ更新の際に保存すべき旧値をバックアップ・メモリに書き戻し、状態の復元や逆依存制約を充足するための旧値の参照ができるようにする。

このメモリ状態の保存については、リオーダ・バッファ[3]のような機構を用いて、投機的な更新前または更新後の値を連想リストに保持する方法も考えられる[4]。しかし、ここで対象とする投機区間は分岐予測[5]などに比べて長く、多量の投機的更新を許容しなければならない。したがって、ハードウェア・コストの面から容量が制約され、かつ投機の成功／失敗の際に投機的更新の数に比例した操作を必要とすることから、連想リストによる実現は不適切である。

一方、我々が提案するキャッシュを用いた状態保存法では、更新回数を制約するのはキャッシュの容量のみであり、また後述するように、投機の成功／失敗時の操作は更新回数に依存せず定数時間で行うことができる。

3.1 システムモデル

システムは共有分散メモリ構成であり、物理的に近接したメモリとプロセッサ（群）からなるノードが、ネットワークによって多数結合されたものとする。

また、各プロセッサはキャッシュを持ち、自ノードのメモリをキャッシュのバックアップとして使用できるものとする。

バリア同期は任意の機構により実現され、同期

表1　キャッシュの状態遷移

from	to					
	I	S	M	US	UM	X
I	$r(s,s)+RB, B^a, B^g, RB$	$r(n,n)$	$w(n)+W$	$r(s,n)$	$w(s)+WB$	$r(n,s)$
S	W, v	$r(n), B^a, B^g, RB$	$w(n)+W$	$r(s)$	$w(s)+W+WB$	—
M	$W(c), v+WB$	$R(c)$	$r(n), w(n), B^a, B^g, RB$	$r(s)+WB$	$w(s)+WB$	—
US	$W+RB, v+RB, RB$	B^g	—	$r(s)$	$w(s)+W+WB$	—
UM	$W(m)+RB, v+RB, RB$	—	B^g	—	$r(s), w(s), R(m)$	—
X	W, B^a, v	—	$w(n)+W$	—	—	$r(n)$

操作のためのメモリ使用は任意である。ただし、同期成立事象はキャッシュに可視であるものとする。

3.2 キャッシュの動作

まず説明を簡単にするために、キャッシュの一貫性保持を write-invalidate で行ない、かつ基本的な状態を MSI (Modified/Shared/Invalid) とする。このときキャッシュの状態は表1に示すように，基本的な状態に US (Unsafe Shared)、UM (Unsafe Modified)、X (eXpiring) の 3 状態を付加したものとなる。

(1) US (Unsafe Shared)
投機的参照が行われ、かつ投機区間中であることを示す。他のプロセッサからの書込が行われると、不正参照を行っていたことが検知され、ロールバックを引き起こす。またリプレースされると他プロセッサの書込が検知できなくなるため、やはりロールバックを引き起こす。投機区間が終了すると、S へ遷移する。

(2) UM (Unsafe Modified)
投機的更新が行われ、かつ投機区間中であることを示す。UM に遷移する際にはノード・メモリへのライトバックが行われ、更新前の値が保存される。状態が UM である時には他プロセッサの参照に対して（キャッシュではなく）メモリの値を返し、危険な更新を行う前の値が参照されるようにする。さらに他プロセッサからの書込とリプレースの際には、更新前後の値を保持することができなくなるためロールバックを引き起こす。投機区間が終了すると、M へ遷移する。

(3) X (eXpiring)
状態が UM であるような他プロセッサのキャッシュを参照し、その旧値をノードメモリから得られたことを示す。次のバリア区間では値が更新されることが確実であるが、その更新要求は UM 状態のキャッシュからは得られない。そこでバリア区間が終了すると、I に遷移して不正に旧値を参照することを回避する。

状態遷移に関する事象としては、以下のようなものがある。

- 自プロセッサによる非投機的参照 $r(n)$
キャッシュ・ミスにより、状態が UM であるような他プロセッサのキャッシュから値を得た場合には ($r(n,s)$)、X へ状態遷移する。また状態が Xであれば，Xに留まる。それ以外の場合には ($r(n)$, $r(n,n)$)、通常の MSI プロトコルと同じ遷移を行う。なお状態が US、UM であることはない。
- 自プロセッサによる非投機的更新 $w(n)$
どのような状態であっても M に遷移し、旧状態が M でなければ他のキャッシュ・コピーの無効化を行う ($+W$)。なお状態が US、UM であることはない。
- 自プロセッサによる投機的参照 $r(s)$
状態が UM であれば UM に留まり、それ以外の状態にあれば US へ遷移する。ただしキャッシュ・ミスにより状態が UM である他プロセッサのキャッシュから旧値を得た場合には ($r(s,s)$)、そのキャッシュからの無効化要求が得られないため I に遷移してロールバックする ($+RB$)。
- 自プロセッサによる投機的更新 $w(s)$
状態が UM でなければに UM 遷移するとともに、更新前の値をノード・メモリにライトバックして保存する ($+WB$)。
- 他のプロセッサによる参照 R
状態が M または UM の場合にのみ生じ、M の場合には S に遷移するが、UM の場合には状態遷移は行われない。状態が M であればキャッシュの値を ($R(c)$)、UM であればノード・メモリに保存した値を ($R(m)$)、それぞれ参照元に返す。
- 他のプロセッサによる更新 W

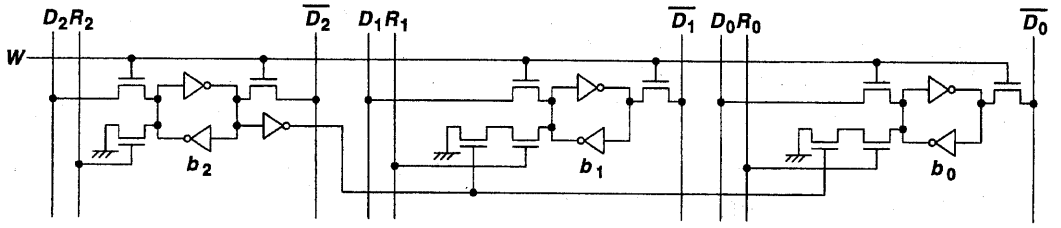

図5　状態情報を保持する機能メモリの構成

どのような状態であってもIに遷移し、UMかUSであればロールバックする(+RB)。旧状態がMであればキャッシュの値を($W(c)$)、UMであればノード・メモリに保存した値を($W(m)$)、それぞれ更新元に返す。

- リプレース v
  どのような状態であってもIに遷移し、UMかUSであれば投機的アクセスに関する情報が失われるためロールバックする(+RB)。旧状態がMであればノード・メモリへのライトバックを行う。
- バリア同期 B^a
  状態がXであれば、Iへ遷移する。状態がUSかUMであることはなく、また他の状態では状態遷移を行わない。
- 同期成立確認（投機区間終了）B^g
  状態がUSであればSへ、UMであればMへ遷移する。状態がXであることはなく、また他の状態では状態遷移を行わない。
- ロールバック RB
  状態がUSかUMであればIへ遷移する。状態がXであることはなく、また他の状態では状態遷移を行わない。

3.3 キャッシュ・タグの構成

前節で述べた状態遷移のうち、バリア同期(B^a)、同期成立確認(B^g)、およびロールバック(RB)による状態遷移は、複数のキャッシュラインに対して行われる。これらの状態遷移を、対象となるライン数に無関係に定数時間で行うことができれば、投機の開始、成功、失敗の時間コストを定数オーダにすることができる。

この定数時間での一括状態遷移は、マスク付きの一括リセットが可能な比較的簡単な機能メモリを用いて、低いハードウェア・コストで実現することができる。すなわち、

- 全てのワードについて、あるビット b_r を0にする機能：$reset(b_r)$
- 全てのワードについて、あるビット b_m が1であれば別のビット b_r を0にする機能：$masked\_reset(b_m, b_r)$

を備えたメモリを用いる。

この機能メモリを用いて、表2に示すように状態情報をエンコードすると、複数ラインの状態遷移を以下のように実現できる。

表2　状態情報のエンコード

state	$b_2b_1b_0$	B^a	B^g	RB
I	000	I (000)	I (000)	I (000)
S	001	S (001)	S (001)	S (001)
M	010	M (010)	M (010)	M (010)
X	100	I (000)	—	—
US	101	—	S (001)	I (000)
UM	110	—	M (010)	I (000)

(1)　同期操作 B^a
状態がXであるような全てのラインについてX → Iを行なえばよく、かつ状態がUS、UMのラインは存在しない。したがって $reset(b_2)$ により実現できる。

(2)　同期成立確認 B^g
状態がUS、UMであるような全てのラインについてUS/UM → S/Mを行なえばよく、かつ状態がXのラインは存在しない。したがって $reset(b_2)$ により実現できる。

(3)　ロールバック RB
状態がUS、UMであるような全てのラインについてUS/UM → Iを行なえばよく，かつ状態がXのラインは存在しない。したがって $masked\_reset(b_2, b_1)$、$masked\_reset(b_2, b_0)$、$reset(b_0)$ の順で行なえば実現できる。

なお上記の機能メモリは、図5のように実現することができる。すなわち R_2 を1にすれば $reset(b_2)$

が、またR_1、R_0を1にすれば$masked_reset(b_2, b_1)$と$masked_reset(b_2, b_0)$がそれぞれ行なわれる。また通常のCMOS-SRAMに付加されるトランジスタは7個であり、ほぼ1ビットの増加分に過ぎない。

4. おわりに

同期操作の完了後に行われるべきメモリ・アクセスを、同期の成立を仮定して投機的に実行することにより、同期操作／順序制約維持に要するコストを隠蔽する方式を提案し、その実装方法を検討した。今回提案した機構は以下の特徴を持つ。

(1) メモリに関する投機開始時の状態保存と投機的状態更新を、キャッシュとノード・メモリを用いて行う。すなわち、投機開始時の状態は必要に応じてノード・メモリに保存され、投機的な状態更新はキャッシュに対してのみ行われる。

(2) 投機の失敗は、投機的に参照／更新したメモリの値が同期成立確認以前に他のプロセッサによって更新されることにより検知する。このため、投機的に参照／更新したことをキャッシュに記憶し、他プロセッサからの更新要求によってロールバックを行なう。

(3) 投機的に更新したメモリの値を他のプロセッサが参照することは許容され、主記憶に保存された更新以前の値が得られる。この値をキャッシュしたプロセッサは、自身の同期操作によってキャッシュを無効化し、その時点では非投機的に更新されている（可能性がある）値を参照できるようにする。

(4) キャッシュ・ディレクトリのハードウェア構成を工夫することにより、投機的／非投機的なメモリ・アクセスはもちろん、投機開始や成功／失敗による投機終了に関する操作も全て定数オーダで実現する。

以上が今回の提案で実現できるシステムのまとめであるが、あくまで投機的な実行が理論上は有効であることを説明してきたのであり、今後の課題として以下のようなことがある。

- シミュレーションによる評価
- バリア同期だけでなく、spinロックなどに対する投機的実行の検証

参考文献

1) 山家陽, 村上和彰. バリア同期モデル—Taxnomyと新モデルの提案およびモデル間性能比較. 並列処理シンポジウムJSPP'93, pp. 119–126, May 1993.

2) M.D.Smith, M.S.Lam, and M.A.Horowitz. "Boosting Beyond Static Scheduling in a Superscalar Processor." In *Proc. 17th Int. Symp. on Computer Architecture*, pp. 344-355, May 1990.

3) J.E.Smith, and A.R.Pleszkun. "Implementation of Precise Interrupts in Pipelined Processors." In *Proc. 12th Int. Symp. on Computer Architecture*, pp. 36-44, June 1985.

4) 王造潤史, 松本尚, 平木敬. Loopを並列実行するアーキテクチャ. 情報処理学会研究会報告, 96-ARC-119-1, pp. 61-66, August 1996.

5) M.D.Smith, M.Johnson, and M.A.Horowitz. "Limits on Multiple Instruction Issue." In *Proc. Third Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 290-302, April 1989.