計算機アーキテクチャ 95－2
(1992. 8. 19)

イベント対応型キャッシュ・コヒーレンス制御方式とそのバリア同期への応用

齋藤 秀樹†, 森 眞一郎†, 富田眞治†
田中高士‡, David FRASER‡, 城 和貴‡
†：京都大学 工学部, ‡：(株) クボタ

大規模な密結合型マルチプロセッサ・システムにおいて、キャッシュ・コヒーレンス制御のオーバヘッドを軽減することは必要不可欠である。筆者らはそれを実現するためにメモリ・アクセス・パターンの性質を利用することを考えている。本稿では、メモリ・アクセスに対するイベントという概念の導入およびイベントの種類に対応したコヒーレンス動作を行うキャッシュについての提案をした後、その応用例のバリア同期について原理と動作、さらに、開発中の実験システムにおける実装方法および予測される性能についても論じる。

The Event Correspondent Cache Coherency Scheme and Its Application to Barrier Synchronization

Hideki SAITO†, Shin-ichiro MORI†, Shinji TOMITA†
Takashi TANAKA‡, David FRASER‡, and Kazuki JOE‡

†: Department of Information Science
Faculty of Engineering, Kyoto University
Yoshida-hon-machi, Sakyo-ku, Kyoto 606-01 Japan

‡: Office of Computer Business, KUBOTA Corporation
ASTEM RI
17 Chudoji, Minami-machi, Shimogyo-ku, Kyoto 600 Japan

E-mail: {saito, moris, tomita}@kuis.kyoto-u.ac.jp
E-mail: {takashi, david, joe}@kocb.astem.or.jp

It is essential to reduce the overhead of cache coherence especially in large scale shared-memory multiprocessors. The authors have been conducting research on efficient cache coherency mechanisms which reflect the characteristics of memory access patterns. In this paper, we present the concept of "events" concerning memory accesses and a new cache coherency scheme which changes its coherence protocols correspponding to each event. We also present the structure and behavior of our new barrier synchronization method, its implementation on our prototype machine: ASURA, and discuss about its estimated performance.

1 はじめに

キャッシュはプロセッサとメモリとの間の速度差を埋める役割を果たしており、プロセッサの性能を十分に引き出すために必要な要素の1つとなっている。大規模な密結合型マルチプロセッサ・システムにおいてもキャッシュは一般的になりつつある [4][6][11]。大規模なシステムにおけるプロセッサ-メモリ間ネットワークの構成は、一般にブロードキャスト機能を前提としない[1]という点で小規模なシステムのものと異なっており、そのことに起因するキャッシュ・コヒーレンス制御のオーバヘッドは大きな問題となっている。筆者らは不要なコヒーレンス動作をなくすことによってこの問題に対処することを考えている。

本稿では、メモリ・アクセスに対するイベントという概念の導入および大規模な密結合型マルチプロセッサ・システムを対象とした、イベントの種類に対応したコヒーレンス制御を行うキャッシュについての提案を行い、その応用例についての考察を行う。本方式はディレクトリ方式のキャッシュ・コヒーレンス制御 [3] を行うシステムを前提として考案されたものであるが、他の方式に応用することも可能であると思われる。

まず、第2章でイベント対応型キャッシュ・コヒーレンス制御方式についての提案を行う。第3章ではその応用例として同方式を用いたバリア同期操作の原理と動作について述べた後、開発中の実験システムにおける実装について述べる。第4章では第3章で述べたバリア同期操作について定性的な性能評価を行い、第5章で簡単なまとめを述べる。

2 イベント対応型キャッシュ・コヒーレンス制御方式 (ECCC)

従来のキャッシュは、ソフトウェアに対し透過性を保証するという方針で設計が行われてきた。そのため、キャッシュのコヒーレンス制御は非常に厳密かつ、柔軟性に欠けるものであった。具体的には、すべてのキャッシングされたデータに対して、ただ一つのプロトコルによって、そのデータに対するメモリ・アクセスが起こるたびにコヒーレンス動作を行っていた（図 1(a) 参照）。そのために不必要／不適切なコヒーレンス動作が発生し、無駄なトラフィックの増加を招いていた。このような不必要なコヒーレンス動作によるオーバヘッドは、大規模なシステムにおいては致命的な問題となる。

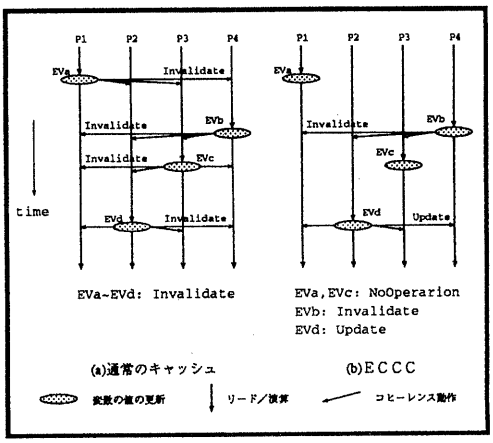

図 1: イベント対応型キャッシュ・コヒーレンス制御

ところが実際には、特有のアクセス・パターンを持つデータや特殊な意味をもつデータ等が多数存在し、それぞれのデータには最適なコヒーレンス制御プロトコルというものが存在する。そこで、アクセス・パターンや意味といったデータの属性をキャッシュのコヒーレンス制御に反映させ、不必要なコヒーレンス動作をなくすとともに、必要なものに対しては最適なプロトコルでそれを実現するイベント対応型キャッシュ・コヒーレンス制御方式 (Event Correspondent Cache Coherency Scheme: ECCC) を提案する。

ECCC におけるイベントとは、キャッシュあるいはその制御機構（ディレクトリ等）が、コヒーレンス制御を開始する誘因となるものであり、

- メモリ・アクセスそのものによって発生するイベント
- メモリ・アクセス結果の関数として発生するイベント

の2つに分類することができる。

前者のイベントとしては、通常のメモリ・アクセス自体をイベントと考えることもできるが、

[1] ブロードキャストが O(1) のコストで実行できないことを言う。

- アドレス対応になんらかの属性を定義し、アドレスによってイベントを区別する
- 複数のアクセス・タイプを定義できる場合には、アクセス・タイプを属性として、イベントを区別する

といったことも可能である。

一方、後者のイベントは主にデータ自体の意義（使われ方）を反映した属性であり、

- アクセス・データそのもの
- アクセスの履歴

あるいは、

- それらを用いて簡単な演算を行った結果

などがある条件を満たした等の事象に対応する。例えば、メモリ・アクセスの結果、値が0になったという事象もイベントの1つである。

次に、このようなイベントに対して、それぞれ行うべきコヒーレンス制御、具体的には、

1. コヒーレンス動作を行うか否か
2. コヒーレンス動作を行う場合には、そのプロトコル

を定義する。

このように、イベントの種類に応じた最適なプロトコルを事前に定義しておき、イベント発生時にその定義にしたがった制御を行うことで、無駄のないキャッシュ・コヒーレンス制御が可能となる（図 1(b) 参照）。

従来のキャッシュ・コヒーレンス制御は、ECCC において、メモリ・アクセスそのものをイベントとして定義し、イベント発生時には必ず何らかのコヒーレンス動作を行った場合にほかならない。

3 ECCC のバリア同期への応用

密結合型マルチプロセッサ・システムではプロセッサ間の同期が重要な課題である。特に大規模なシステムにおいては、同期のオーバヘッドを小さくすることが必要不可欠である。マルチユーザ・マルチタスク型の大規模な密結合型 MIMD 計算機における同期を考える場合には、以下のことを考慮しなければならない。

- ハードウェアによる同期機構に求められる条件
  - 設置コストがそれほど大きくないこと
- ソフトウェアによる同期操作に求められる条件
  - ネットワークに対する負荷が小さいこと
  - 必要とするメモリ量が少ないこと
- ハード／ソフト両方に求められる条件
  - 高速であること
  - 柔軟であること
  - 文脈切替時のオーバヘッドが小さいこと

従って、ハードウェアのみによる同期機構あるいはソフトウェアのみによる同期操作だけでは不十分であると予測される。筆者らは、ソフトウェアによる同期操作をハードウェアでサポートすることによって、オーバヘッドの少ない柔軟な同期操作を実現することを考えている。

ソフトウェアによる同期操作としては、排他制御のために用いられるロック操作と複数のスレッドが同時に待ち合わせるために用いられるバリア同期操作が一般的である。ECCC はどちらにも応用可能であるが、今回はバリア同期操作を取り上げる。ロック操作については別論文で発表予定である。

3.1 バリア同期に関連する研究

バリア同期に関連する研究について、簡単にまとめておく。（表 1参照）

ハードウェアによるバリア同期機構の最も単純なものとしては1本の信号線を用いて wired-AND をとる方式が知られている。R.Gupta による Fuzzy Barrier[5] の概念を用いてこれを拡張したものが松本の Elastic Barrier[9] と高木らの Ultimate Barrier[8] である。Fuzzy Barrier とは、プログラムの命令列をバリア領域と非バリア領域の 2 つに分類し、同期点ではなく同期区間で待つことによりプロセッサの待ちを減らすというものである。

ソフトウェアによるバリア同期アルゴリズムは、集中型とネットワーク型に分類できる。後者はさらにツリー型と非ツリー型に分類できる。

集中型バリア同期の原理は、バリアに参加するスレッドの数をカウンタの初期値として設定した後、スレッドが同期点に達したときカウンタをデクリメントし、カウンタの値を監視して値が 0 となったことを確認して後続の命令を実行するというものである。集中型のバリアとしては、Hensgen らが 1 つのカウンタと 2 状態フラグを用いたもの、Tang らと Lubachevsky は 2 つの共有カウンタを用いた

表 1: バリア同期操作のための機構／アルゴリズムの分類

ソフト＋ハード	集中型		ECCC に基づいたキャッシュを用いたバリア同期
			Broadcast Networks を用いたバリア同期
ソフトウェア	集中型		従来型のキャッシュを用いる／用いない
	ネットワーク型	ツリー型	Software Combining Tree Barrier
			Tournament Barrier
			New-Tree Barrier
		非ツリー型	Dissemination Barrier （ハイパーキューブ構造）
			Butterfly Barrier （バタフライ構造）
ハードウェア	1本の信号線		最も単純なバリア同期ハードウェア
	1本の信号線＋カウンタ		Elastic Barrier / Ultimate Barrier

ものを提案している。また、Agarwal らは集中型バリアにおけるネットワーク上のトラフィックと競合を減少させるための adaptive backoff scheme を提案している。[7]

ツリー型のバリア同期とは、バリアに参加するスレッド全体を n 進のツリー構造に論理分割し、「ツリーの各階層を構成する各々のクラスタ内で集中型のバリア同期を行い、その中の代表が次のレベルのバリア同期に参加する。」という操作を階層的に繰り返すことで全体のバリア同期を達成するものである。ツリー型バリアとしては、Software Combining Tree を用いたバリアが Tang らによって、Tournament 方式が Hensgen らと Lubachevsky によって、Fan-in が 4 で Fan-out が 2 の木構造を持った New-Tree Barrier が Mellor-Crummey らによってそれぞれ提案されている。[7]

非ツリー型のバリア同期アルゴリズムとしては、Hensgen らが Dissemination Barrier を、Brooks が Butterfly Barrier をそれぞれ提案している。[7] 前者はハイパーキューブ型の構造を持ち、後者は FFT のバタフライ演算と同じ構造を持っている。

また、Beckmann らは多段結合網をもとにした Broadcast Network を提案し、バリア同期操作に応用可能であることを示している [2]。

3.2 ECCC に基づいたキャッシュを利用したバリア同期操作

本節では、ECCC のバリア同期への応用例を示す。バリア同期のアルゴリズムとしては集中型を考えることとする。それは、集中型のアルゴリズムがバリア同期の基本的な概念に最も忠実であると思われるからである。ネットワーク型のアルゴリズムにも集中型バリアの概念が内包されている。集中型のバリアを高速化することができれば、ネットワーク型

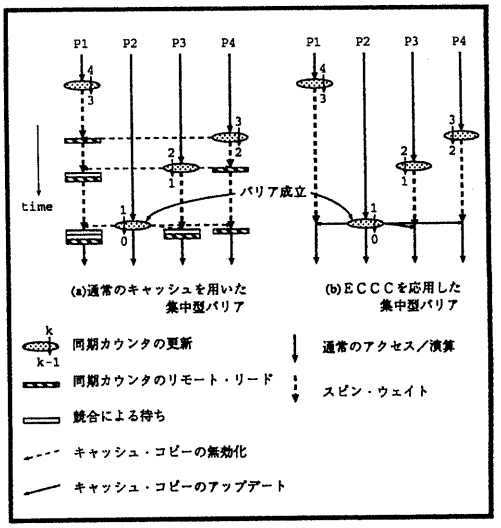

図 2: ECCC のバリア同期への応用

のアルゴリズムにもそれを応用可能であると思われる。

従来の高速化技法として、同期変数をキャッシングし同期待ちのスピンをキャッシュ上で行う方法が考えられている。しかしながら、ブロードキャスト機能を持たない大規模システムでは、この方法は同期変数の更新に伴うコヒーレンス動作のオーバヘッドのため十分な性能を得ることができない（図 2(a) 参照)。

ところが、バリア同期における同期変数の意義を考えると、各スレッドにとって同期変数の値自身が何であるかは重要ではなく、その値が定められた値になった、即ち、「バリアが成立した」という事象を知ることが重要なのである。このような同期変数の参照の性質を、当該同期変数に対するキャッシュ・コヒーレンス制御に反映することで、バリア同期の高速化を図ることにする。

```
shared counts : array [0..1] of integer
   //各要素は別々のメモリ・ブロックに配置
processor private turn : integer
   //turn は 1 と 0 の 2 値変数
processor private vpid : integer
   //各プロセッサに固有なプロセッサ番号
count[0] := T
   //T はバリアに参加するスレッドの数
turn := 1

procedure counter_barrier
  if vpid = 0
    count[turn] := T
  turn := 1 - turn
  fetch_and_decrement(&count[turn])
  repeat until count[turn] = 0
  return
```

図 3: バリア同期アルゴリズム

具体的には、各スレッドによる同期変数の更新時ごとのコヒーレンス動作を止め（一時的な inconsistency を許す）、当該同期変数に対する最後の更新時、即ち、「バリア同期成立時」にのみ write-update 型のコヒーレンス動作を行う。これにより、コヒーレンス動作のオーバヘッドが削減でき、バリア同期の高速化が実現できる。ECCC において「バリア同期の成立」というイベントを定義することによりこの制御が実現可能である。

図 3に ECCC に基づいたキャッシュを利用したバリア同期のためのアルゴリズムを示す。1回のバリア同期操作では同期変数配列 counts の要素のうち一方だけが使用される。（以下、これを count と呼ぶ。）

ここでは、fetch_and_decrement の実行結果として同期変数 count の値が 1 から 0 に変わることを 1 つの独立したイベント（イベント A）として定義し、count の値が 2 以上であるときの fetch_and_decrement の実行というイベント（イベント B）とは区別することとする。同期変数 count のあるブロックに対する fetch_and_decrement の結果、イベント A が発生した場合にのみコヒーレンス動作を行い、イベント B が発生した場合にはコヒーレンス動作を行わないこととすると、バリア同期操作によって引き起こされるネットワーク上のトランザクション回数は各スレッドについて O(1) である。また、イベント A の発生時には

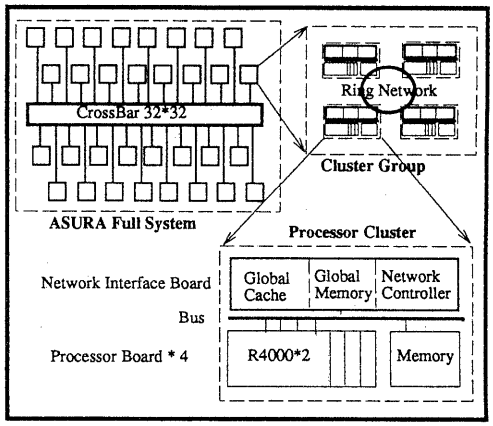

図 4: ASURA のアーキテクチャ

write-update 型のコヒーレンス動作を選択することにより、invalidation によるキャッシュ・ミスは発生しない。よって、同期変数 count についてのコヒーレンス動作による通信競合の発生を防止することが可能となる（図 2 (b) 参照）。また、キャッシュがバリア同期のために使用されていることはわかっているので、キャッシュ・コピーを update する時に転送するデータは 1 ワードでよい[2]。

3.3 ASURA への実装

3.3.1 ASURA のアーキテクチャ

ASURA は京都大学と（株）クボタが共同で開発している実験システムである [10][11]。図 4にそのアーキテクチャを示す。ASURA は 3 階層のキャッシュを持つプロセッサ・クラスタ方式の密結合型マルチプロセッサ・システムである。

プロセッシング・エレメント (PE) はオンチップの 1 次キャッシュと 4MB の 2 次キャッシュを持つ R4000MC である。これらのキャッシュはともに write-back 方式を採り、シナプス・プロトコル [1] に従ったスヌープ方式のキャッシュ・コヒーレンス制御を行う。

8 個の PE と 256MB のローカル・メモリ、ネットワーク・インタフェース (NIF) をバスによって結合し、1 つのプロセッサ・クラスタ (PC) を構成する。このローカル・メモリはク

[2] 1 ワードが 32bit であると仮定すると、スレッドの数が 4G 個を越えない限りはこれで十分である。本当に必要であるなら、2 ワードとしてもよいと思われる。

ラスタ内の PE によって共有される。NIF は 4GB のグローバル・メモリ[3]と 32MB のグローバル・キャッシュから構成される。グローバル・メモリは ASURA システムの分散共有メモリとして全ての PE によって共有される。グローバル・キャッシュはクラスタ内の PE で共有される。クラスタ間のコヒーレンス制御はシナプス・プロトコルに従ったクラスタ単位のフルマップ・ディレクトリ方式により実現する。

複数の PC を階層型のネットワークによって結合することにより ASURA システムを形成する。現段階では、4 つの PC をリング結合してクラスタ・グループを形成し、32 クラスタ・グループをクロスバ結合することにより、最大 1024CPU を持つことが可能である。

3.3.2 ASURA への実装

ECCC に基づいたキャッシュを利用したバリア同期機構は、以下のように ASURA システムに実装される。

イベント A およびイベント B を 3.2節と同様に定義する。それらは共有メモリの各コヒーレンス・ブロックの先頭 1 ワードについてのみ有効であるものとする。イベント A およびイベント B に対応する属性を持った共有メモリ・アクセスとして fetch_and_decrement_Ev (以下 F&D_Ev) と load_Ev の 2 種類を用意する。プロセッサは F&D_Ev を load 型の命令として発行する。

プロセッサが共有メモリ上のある変数（コヒーレンス・ブロックの先頭にあるものとする。）に対して F&D_Ev を発行した場合、以下のように動作するものとする。

1. F&D_Ev は 1 次キャッシュおよび 2 次キャッシュにおいて必ずミスヒットさせる。
2. F&D_Ev はグローバル・キャッシュでのヒット/ミスヒットにかかわらず、共有メモリにフォワードされる。ただし、ミスヒットの場合には領域を確保した後にフォワードされる。
3. F&D_Ev が共有メモリに到達すると、不可分な fetch_and_decrement が行われる。すなわち、x を実行の戻り値とした後に $x \leftarrow x-1$ が実行される。このとき、ディレクトリ・エントリ中の該当する存在ビットをセットする。
4. F&D_Ev の戻り値はグローバル・キャッシュにキャッシングされ、プロセッサに渡される。この時にネットワーク上を流れるデータは 1 ワード (32bit) である。
5. $x \leftarrow x-1$ の結果として x が 0 となった場合、イベント A が発生し、当該キャッシュ・コピーは 0 にアップデートされる。この時もネットワーク上を流れるデータは 1 ワード (32bit) である。

プロセッサが共有メモリ上のある変数（コヒーレンス・ブロックの先頭にあるものとする。）に対して load_Ev を発行した場合の動作は通常の load 命令によるものと同様である。

各スレッドは図 3に示された手続きを実行することにより、全てのスレッドが同期点に達したことを知ることができる。

スレッドに対応したバリア同期の情報は PC に対応した directory entry によって管理される。これはグローバル・キャッシュの管理が PC 対応になされるからである。グローバル・キャッシュは、F&D_Ev をヒット/ミスヒットにかかわりなく共有メモリにフォワードすることによってスレッドと PC との間のギャップを埋める働きをする。

4 バリア同期操作の性能評価

バリア同期操作の性能を評価する対象としては、ブロードキャスト機能を持たない大規模密結合型 MIMD 計算機を考えることとする。

比較対象と評価基準

性能の比較対象として以下の 5 アルゴリズム（8 方式）を考える。

Centralized1 集中型バリア
（従来型キャッシュ使用）
Centralized2 集中型バリア
（キャッシュ不使用）
Combining1 software combining tree barrier
（n 進木：従来型キャッシュ使用）
Combining2 software combining tree barrier
（n 進木：キャッシュ不使用）
Tournament1 Tournament Barrier
（Hensgen らによるもの）
Tournament2 Tournament Barrier
（Lubachevsky によるもの）
NewTree New-Tree barrier
NonTree Dissemination Barrier,
Butterfly Barrier

参考のためにハードウェアのみによるバリア同期機構として Ultimate Barrier についても考えることとする。

[3]40 ビット・アドレスの場合。32 ビット・アドレスの場合には 16MB。

また、性能を評価するための基準として以下の4項目を考えることとする。

- 相互結合網上のトランザクションの回数
- 必要とする共有メモリ量
- 必要とするメモリオペレーションの実現性
- クリティカル・パスの長さ

ここで、クリティカル・パスの長さとは、最後のプロセッサが同期点に到着した時点から全てのプロセッサにそれが伝わるまでの時間であるとする。

以下、Tをバリア同期に参加するスレッドの数、Pをそれらのスレッドを実際に割り付けられた物理プロセッサの総数とする。

定性的性能評価

ECCC に基づいたキャッシュを利用したバリア同期操作と、上に列挙したバリア同期アルゴリズムとの性能の比較を表 2に簡単にまとめた。

ECCC に基づいたキャッシュを利用したバリアと集中型のバリアを比較した場合、ネットワーク上のトランザクション回数について $O(P)$ と $O(P^2)$ という差が生ずる。この差は不必要なコヒーレンス動作をしないというところから生まれる。

また、Tournament Barrier、New-Tree Barrier もトランザクション回数について優位性を持っていると言える。

必要とする共有メモリ量については、ECCC に基づいたキャッシュを利用したバリアと集中型バリアの2つが優位性を持っていると言える。

メモリ・オペレーションについては、非ツリー型と New-Tree barrier が通常の Read/Write だけでよいのであるが、多くの密結合型マルチプロセッサ・システム上で atomic な Read-Modify-Write が実装されていることを考慮すると、これによる優劣の判断は困難である。

クリティカル・パスの長さは、ECCC に基づいたキャッシュを利用したバリアと集中型のバリアがハードウェア・レベルで $O(P)$、それ以外は全てソフトウェア・レベルで $O(\log T)$ である。ハードウェアのレベルとソフトウェアのレベルの差は実装に深く依存するので、一概に比較することはできない。

考察

ECCC に基づいたキャッシュを利用したバリア同期機構の利点は、

- ネットワーク上のトランザクション回数が少ない
- 必要となる共有メモリ量が少ない
- コヒーレンス動作による通信競合が発生しない

ことが挙げられる。

ECCC に基づいたキャッシュを利用したバリア同期はユーザに割り当てられたメモリ空間上で行われるため、システム・コールを必要としない。また、同じ理由から多数のバリアを同時に設置可能である[4]ということも言える。マルチユーザ・マルチタスクを前提としたシステムにおいては、これらをハードウェアのみによるバリア同期機構によって行うことはほぼ不可能である。

ECCC に基づいたキャッシュを利用したバリア同期機構の最大の弱点はクリティカル・パスが $O(P)$ になることである。これについての最良の解決策は $O(\log P)$ でマルチキャストが実現できるネットワークを考案することである。より現実的な解決法は2つ考えられる。第一の方法としては、$O(\log P)$ でブロードキャストできるネットワークを用いることである。第二の方法としては、ツリー型バリアに ECCC を適用することである。概念的には、software combining tree と同様である。この場合には、ツリーの各階層における集中型バリアが効率化されることとなる。

コストの面では、ソフトウェアのみによるバリアが優位性を持っている。しかし、大規模なシステムにおいてキャッシュ・コヒーレンス制御をハードウェアによって行うことを前提とすれば、ECCC の実現に要するコストはプロトコル切り替えのハードウェア・コストだけであり、Elastic Barrier や Ultimate を多数設置する場合よりもはるかに低いと思われる。

5 おわりに

従来型のキャッシュ・コヒーレンス制御方式に対して問題を提起し、イベントという概念を導入とその概念を用いたキャッシュ・コヒーレンス制御の方式についての提案を行った。また、その応用例として、バリア同期操作とその性能についても述べた。

[4] ASURA 実装の場合、バリアカウンタの総数は 2M 個である。

表 2: バリア同期の性能の比較

	通信回数	メモリ量	メモリ操作	最長パス
ECCC	$O(T)$	$O(1)$	Fetch_and_Decrement	$O(P)$
Centralized1	$O(T^2)$	$O(1)$	Fetch_and_Φ	$O(P)$
Centralized2	$\Omega(T)$	$O(1)$	Fetch_and_Φ	$O(P)$
Combining1	$O(nT)$	$O(T/n)$	Fetch_and_Φ	$O(\log T)$
Combining2	$\Omega(T)$	$O(T/n)$	Fetch_and_Φ	$O(\log T)$
Tournament1	$O(T)$	$O(T \log T)$	Fetch_and_Φ	$O(\log T)$
Tournament2	$O(T)$	$O(T)$	Fetch_and_Φ	$O(\log T)$
NewTree	$O(T)$	$O(T)$	通常の Read/Write	$O(\log T)$
NonTree	$O(T \log T)$	$O(T \log T)$	通常の Read/Write	$O(\log T)$
Ultimate Barrier	0	0	なし	電気信号遅延

注) Fetch_and_Φ は fetch_and_decrement などの atomic な read-modify-write のことを言う。

ECCC に関しては、アクセス・パターンについての調査を進め、どのようなコヒーレンス動作が必要とされるのかを考えること、また、バリア同期操作については、シミュレーションによる調査と比較検討によって性能を詳しく評価することが今後の課題である。

謝辞

日頃ご討論頂く名古屋大学 阿草清滋教授、(株) クボタ 山口宗之部長、(株) クボタ ASURA プロジェクト・チーム の諸氏、ならびに京都大学 富田研究室 の諸氏に感謝致します.

参考文献

[1] J. Archibald and J.-L. Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Trans. on Computer Systems*, 4(4):273–298, November 1986.

[2] C. Beckmann and C. Polychronopoulos. Broadcast networks for fast synchronization. In *Proc. of the 1991 Int'l Conf. on Parallel Processing*, pp.I–220–I–224, 1991.

[3] D. Chaiken, C. Fields, K. Kurihara, and A. Agarwal. Directory–based cache coherence in large–scale multiprocessors. *IEEE Computer*, pp.49–58, June, 1990.

[4] D. R. Cheriton, H. A. Goosen, and P. D. Boyle. Paradigm: A highly scalable shared–memory multicomputer architecture. *IEEE Computer*, pp.33–46, February, 1991.

[5] R. Gupta. The fuzzy barrier: A mechanism for high speed synchronization of processors. In *Proc. of the Third Int'l Conf. on ASPLOS*, pp.54–63, 1989.

[6] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The Stanford Dash Multiprocessor. *IEEE Computer*, pp.63–79, March, 1992.

[7] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. on Computer Systems*, 9(1):21–65, 1991.

[8] 高木, 有田, 曽和. 重複可能なバリア型同期のためのスケジューリングアルゴリズムとその性能. 信学技報 CPSY91-15, 電子情報通信学会, 1991.

[9] 松本. Elastic barrier: 一般化されたバリア型同期機構. 情報処理学会論文誌, 32(7):886–896, 1991.

[10] 城, 柳原, D. Fraser, 田中, 新田, 森, 齋藤, 富田. 分散共有メモリ型マルチプロセッサ「ＡＳＵＲＡ」の階層性とその評価. 情処研報 92-ARC-95-1, 情報処理学会, 1992.

[11] 森, 齋藤, 五島, 富田, 田中, D. Fraser, 城, 新田. 分散共有メモリ型マルチプロセッサ「阿修羅」の概要. 情処研報 92-ARC-94-6, 情報処理学会, 1992.